A bytecode compiler and interpreter for FOOL$^+$ language

Course project of ANALISI STATICA DI PROGRAMMI

Chun Tian

Scuola di Scienze, Università di Bologna
chun.tian@studio.unibo.it
Numero di matricola: 0000735539

Abstract. In this course project, we have implemented a bytecode compiler and interpreter for an object-oriented (OO) language called FOOL$^+$, which is based on a toy programming language called FOOL, given by professor. Various types of syntactic and semantic analysis are implemented, including scope and type checkings. The OO extension is single-inheritance, and is based on generic function instead of message-passing model. Multi-methods, polymorphic types, and dynamic method dispatching are supported. Beside the OO extension, tail-recursion optimization of function calls is also implemented. To compile and run source code written in this new language, we have also defined a set of bytecode instructions and implemented a stack-based interpreter. Finally the compiler has ability to save the compiled bytecode into disk files and execute the saved bytecode. The whole project is written in Java programming language, and the only dependent external library is ANTLR (version 4.7 and later).

1 Introduction

The author must admit that, this is a very interesting project and by doing this project he has learnt a lot in Java, ANTLR, and language implementing techniques. Implementing a new programming language is not easy, there're a lot of work to do, no matter how "simple" the language is. And it's still true when the parser code is generated from tools like ANTLR. On the other side, this project is useless in the sense that, the work done in this project (the language design and its compiler and interpreter) has no other use except for passing the exam. The world of programming languages is already massed enough. Instead of putting efforts in new languages, it's much better to focus on real problems and write programs (when necessary) in existing good languages.

This project is an single-person two-weeks coding effort, and it's actually the author's first Java program (except for those helloworld-like small testing code). The highlight of this project, is mainly in the OO feature design and the corresponding implementation at bytecode generation level. It's carefully designed to give object ability to access member variables of its parent classes, and use dynamic method dispatching at runtime to choose the most applicable method for objects of polymorphic types. Many ideas came from the design of Common Lisp Object System (CLOS), but the bytecode-based implementation is totally the credits of the author without looking at the implementation of any other programming languages.

About 2/3 of the code in this project is language-independent (the implementation of symbols and scopes, and the bytecode assembler/interpreter), they can be reused for implementing another better non-toy language. Most of these language-independent code are based on the sample code in the book *Language Implementation Patterns* [1], a few others are from *The Definitive ANTLR 4 Reference* [2], both written by Terence Parr. But the code in the former book was written in ANTLR v3 API but v4. In this project, although code is borrowed from these book, we made the efforts to fully understand those code and translated them into ANTLR v4, and the resulting code becomes much smaller than before and more clear to understand.

On the other side, the language-dependent part is totally based on the advanced interfaces provided by ANTLR v4 (and even the latest 4.7), the author thinks it's a very good code example for demonstrating the power of ANTLR v4 over v3. The benefits of ANTLR v4 over v3 are mainly at two points: 1) the complete separation of grammar definition and Java code, 2) the use of both listener and visitor parser APIs in different situations. (there's only visitor interface in ANTLR v3) In this project, we've made great use of the new listener interface, and the first use is to count the number of nodes and terminals in the parser tree in very few lines of Java code. Professor's sample code has defined and constructed an AST (abstract syntax tree), while using ANTLR v4's great new APIs we have no such needs at all.

The main purpose of this paper is to document all the excellent ideas, designs and choices that the author has made during this project, as precise as possible (when memory is still fresh). Here is the structure of this paper: we first briefly describe the FOOL$^+$ language at syntax and then semantic levels, with each part divided into non-OO and OO subsections. Then we explain the implementation details of each compiler phases at source level, firstly the frontend (syntactic and semantic analysis), then the backend (code generation and optimization), with each part still divided into non-OO and OO subsections. Between the frontend and backend, we'll introduce the bytecode instruction set and the assembler, which can be used independently to write programs. We want to emphasize that fact that, although FOOL$^+$ may not be Turing-complete, the assembly language is! And we'll use this assembly language to write complicated code doing dynamic method dispatching, runtime subclass checking, and many other things.

Finally, the paper is written as brief as possible, without any garbage text and background information filled in, because this is not a thesis paper. Whenever something can be understood easily by looking the code (e.g. the ANTLR grammar definition file), we won't explain it in details. After all, for self-referential purposes, it's only important to document the the ideas with a few words. And professor may not have patience to read too many pages, given the fact that, there're about 70 students in the class and finally there may be at least 20 group projects submitted for the exam. Or professor may directly looks into the project code and bypass the paper, so we'll describe *how to import and open the project in Eclipse immediately after this introduction section.*

2 Open the project in Eclipse

Eclipse 4.6.3 (latest release as June 2017) is confirmed working for opening this project, no special plugin is needed. Two ANTLR 4.7 Jars were shipped with the project, one (`antlr-4.7-complete.jar`) is for building and running the project in Eclipse IDE, the other (`antlr-runtime-4.7.jar`) is for building the standalone version of our FOOL compiler into `FOOL.jar`.

The following steps are for importing and opening the project in Eclipse:

1. Extract project files into any folder on local disk;
2. Start Eclipse, choosing any workspace to work on;
3. Choose `File -> Import...`, then `General -> Existing Projects into Workspace`;
4. Go `Next`, in "Select root directory", choose the folder containing the project files;
5. Select `FOOL-Chun` in the "Projects" list, click `Finish`.

Now the project is correctly imported into Eclipse in any workspace, once opened, the project should be built automatically. To get a standalone version of the FOOL compiler (and also interpreter), it must be done in command line with Apache Ant (version 1.7 and later) with the following steps: (suppose it's already built in Eclipse)

1. Go to the project's root directory (should be named `FOOL`);
2. Execute `ant` command (suppose it's in `PATH` environment variable).

As the result, a `FOOL.jar` file will be created in the root directory. Here is the usage:

- Executing `java -jar FOOL.jar`, the compiler accepts FOOL program code from standard input stream.
- Executing `java -jar FOOL.jar xxx.fool`, it compiles the file `xxx.fool` containing FOOL program code, run the program with bytecode saved into `xxx.byte`;
- Executing `java -jar FOOL.jar xxx.byte`, it loads the bytecode from file `xxx.byte` and execute the program inside.

3 The syntax of FOOL$^+$ language

For the core language, we have slightly modified the top-level structure of the original FOOL language, below is the new ANTLR grammar definition:

```
prog   : (block SEMIC)+ ;

block  : body#codeBody
       | varasm#globalVar
```

```
        ;

body    : exp#singleExp
        | let exp#letInExp
        ;

fun     : type ID LPAR ( vardec ( COMMA vardec)* )? RPAR body ;
```

The grammar for defining functions didn't change, but the now the function body has a new grammar block called *body*, which is the original *prog*. Then we added the ability to define global variables (*varasm*) at the *block* level (thus inside functions it's impossible to defined new variables). And finally, we made *prog* the top-level structure which can contain multiple blocks. Thus it's possible to have multiple global variables and expressions (and let binding) in single FOOL program. Adding these extra things really has no much costs from the view of compiler source code, while the expressivity of the new FOOL language is much better than before.

For the OO extension, there're five minor additions to the grammar. First, at block level, we have added a new branch called *defcls* with its grammar definition:

```
block   : body#codeBody
        | varasm#globalVar
        | defcls#classDef // for OO support (part 1 of 5)
        ;

// BEGIN: additions for OO support (part 2 of 5)
defcls : CLASS ID (LPAR ( vardec (COMMA vardec)* )? RPAR)?
 ASM OBJECT supers? slots ;

supers : INHER ID (LPAR (exp (COMMA exp)*)? RPAR)? SEMIC;

slots  : (slotd SEMIC)* END ;

slotd  : vardec#slotNoInit
        | varasm#slotInit
        ;
// END
```

This grammar has the ability to define new classes with the following features:

- The class must have a name;
- The class can optionally have initialization parameters;
- The class can optionally inherit another class (only single inheritance is supported);
- The class can optionally have one or more slots (member variables);
- Each slot must has its type specified;
- Each slot can have an initialization value (constants or value passed from the class initialization parameters).

Next, we have extended the possible values of types in FOOL, now it can be any ID representing a class:

```
type    : INT
        | BOOL
        | ID // class types for OO support (part 3 of 5)
        ;
```

Then, to actually "use" the object, we have extended the grammar definition of *value*:

```
value   : INTEGER#intVal
        | ( TRUE | FALSE )#boolVal
        | LPAR exp RPAR#baseExp
        | IF cond=exp THEN CLPAR thenBranch=exp CRPAR ELSE CLPAR elseBranch=exp CRPAR #ifExp
        | ID#varExp
        | ID LPAR (exp (COMMA exp)* )? RPAR#funExp
        | PRINT exp#printExp
        // BEGIN: additions for OO support (part 4 of 5)
        | 'new' ID (LPAR (exp (COMMA exp)*)? RPAR)?#classExp
```

```
        | value '.' ID LPAR (exp (COMMA exp)*)? RPAR#methodExp
        | value '.' ID#slotExp
        // END
        ;
```

in which the last three branches has the following corresponding functionalities:

1. Create new object of a class, passing initialization arguments (if the class requires them);
2. Call a method of an object, taking some other arguments (if the method requires them);
3. Access the value of a slot of an object returned by an *value*.

Finally, four lexical tokens used by above definitions (as keywords) were defined here:

```
// BEGIN: additions for OO support (part 5 of 5)
CLASS  : 'class' ;
OBJECT : 'object' ;
INHER  : 'inherit' ;
END    : 'end' ;
// END
```

That's all. But how to define a method? In fact, at syntactic level a method looks exactly the same as a normal function taking several parameters, of either primitive types or classes. Defining methods outside of classes is the central idea of generic function model, and it's more powerful than the usual message-passing model, and there's no need to have special grammar support for defining methods. To see how this happened, let's see the following "classic" sample code:

```
class A = object end;
class B = object inherit A; end;
let int foo(A o1, A o2) 1;
    int foo(A o1, B o2) 2;
    int foo(B o1, A o2) 3;
    int foo(B o1, B o2) 4;
    bool flag = false;
    A obj = (if flag then {new A()} else {new B()});
in obj.foo(obj) + 6;
```

In above code, we have defined two class, A and B, without any slot. Class B inherits A. Then we have four "methods" with the same name `foo`, but the types of parameters are not identical. To call such a method on an object `foo` of class `A` or `B`, with itself as the other argument, actually there're two ways:

1. `obj.foo(obj)`
2. `foo(obj, obj)`

The first way is what we usually see in OO languages like C++ and Java. (I call it the message-passing syntax) But if we consider the leading object as the first argument to the method, we naturally got the second way (I call it the generic-function syntax), which looks exactly the same as a normal function definition. (actually C++ supports it too, but functions defined in this way in C++ are not considered as methods of any class) In our FOOL$^+$ language, we simply treat all functions as method, but if the first argument is not an object, there will be no alternative generic-function syntax.

4 The semantic of FOOL$^+$ language

Professor ever questioned "what's the semantic of a programming language" in class, and concluded that, the semantic of Java is defined by the semantic of its bytecode instructions. We don't agree with this opinion for two reasons:

1. The semantic of Java bytecode is still yet to be answered;
2. Java and JVM bytecode are too irrelevant languages: it's possible to compile Java natively and run independent of JVM (e.g. the GCJ project, part of GNU Compiler Collection), while some other languages can be compiled into JVM bytecode and run on top of JVM.

In logic, the *semantic* of any logic term e, notated as $[\![e]\!]$, has the value as a mathematical set, which in turn can be seen as a type, containing a possibly infinite number of logic terms. Here we have the same idea: the semantic of a programming language is the *type* of programs and all its components

(expressions, blocks, values) in that language. When Robin Milner invented the language Standard ML [3], he defined the semantic[1] of his language by giving the type inference rules in that language. Not all programming languages defined their semantics in this way, while all Standard ML compilers were incredibly consistent because of its remarkable language specification.

Now we describe the type inference rules in a bottom-up order according to the grammar definition, as this is actually the order in which the type information for the entire program is built. In the following contexts, we use notations like $\tau \prec \rho$ or $\rho \succ \tau$ to indicate that type (or class) τ is a subtype (or subclass) of ρ.

Below are type inference rules for *values* in FOOL$^+$ language:

1. Integer constants:

$$\frac{e \text{ is an integer token}}{\Gamma \vdash e \Rightarrow \texttt{int}} \text{ [intVal]}$$

2. Boolean constants:

$$\frac{}{\Gamma \vdash \texttt{true} \Rightarrow \texttt{bool}} \text{ [boolVal}_1\text{]} \qquad \frac{}{\Gamma \vdash \texttt{false} \Rightarrow \texttt{bool}} \text{ [boolVal}_2\text{]}$$

3. Parenthesis (base expressions):

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash (e) \Rightarrow \tau} \text{ [baseExp]}$$

4. If-then-else expressions:

$$\frac{\Gamma \vdash \mathit{cond} \Rightarrow \texttt{bool} \qquad \Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Rightarrow \rho \qquad \tau \prec \rho}{\Gamma \vdash \texttt{if } \mathit{cond} \texttt{ then } \{a\} \texttt{ else } \{b\} \Rightarrow \rho} \text{ [ifExp}_1\text{]}$$

$$\frac{\Gamma \vdash \mathit{cond} \Rightarrow \texttt{bool} \qquad \Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Rightarrow \rho \qquad \tau \succ \rho}{\Gamma \vdash \texttt{if } \mathit{cond} \texttt{ then } \{a\} \texttt{ else } \{b\} \Rightarrow \tau} \text{ [ifExp}_2\text{]}$$

5. Variables:

$$\frac{}{\Gamma[x \mapsto \tau] \vdash x \Rightarrow \tau} \text{ [varExp]}$$

6. Function applications:

$$\frac{\Gamma \vdash f \Rightarrow (\tau_1, \tau_2, \ldots, \tau_n) \to \tau \qquad \Gamma \vdash e_1 \Rightarrow \rho_1 \ldots \Gamma \vdash e_n \Rightarrow \rho_n \qquad \tau_1 \succ \rho_1, \ldots, \tau_n \succ \rho_n}{\Gamma \vdash f(e_1, e_2, \ldots e_n) \Rightarrow \tau} \text{ [funExp]}$$

7. Print expressions:

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \texttt{print } e \Rightarrow \tau} \text{ [printExp]}$$

8. Class expressions (creating new instances):

$$\frac{\Gamma \vdash \texttt{new } C \Rightarrow (\tau_1, \tau_2, \ldots, \tau_n) \to C \qquad \Gamma \vdash e_1 \Rightarrow \rho_1 \ldots \Gamma \vdash e_n \Rightarrow \rho_n \qquad \tau_1 \succ \rho_1, \ldots, \tau_n \succ \rho_n}{\Gamma \vdash \texttt{new } C(e_1, e_2, \ldots e_n) \Rightarrow C} \text{ [classExp]}$$

9. Method applications:

$$\frac{\Gamma \vdash o \Rightarrow C' \quad \Gamma \vdash m \Rightarrow (C, \tau_1, \ldots, \tau_n) \to \tau \quad \Gamma \vdash e_1 \Rightarrow \rho_1 \ldots \Gamma \vdash e_n \Rightarrow \rho_n \quad C \succ C', \tau_1 \succ \rho_1, \ldots, \tau_n \succ \rho_n}{\Gamma \vdash o.m(e_1, e_2, \ldots e_n) \Rightarrow \tau} \text{ [funExp]}$$

10. Class slot access:

$$\frac{\Gamma \vdash o \Rightarrow C \qquad C \vdash x \Rightarrow \tau}{\Gamma \vdash o.x \Rightarrow \tau} \text{ [slotExp]}$$

Below are type inference rules for higher level constructions in FOOL$^+$ language:

[1] Strictly speaking, it's the *static* semantic

1. Factor (equality of values):

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Rightarrow \tau}{\Gamma \vdash e_1 == e_2 \Rightarrow \texttt{bool}} \text{ [factor]}$$

2. Term (multiplication of values):

$$\frac{\Gamma \vdash e_1 \Rightarrow \texttt{int} \qquad \Gamma \vdash e_2 \Rightarrow \texttt{int}}{\Gamma \vdash e_1 \times e_2 \Rightarrow \texttt{int}} \text{ [term]}$$

3. Expressions (sum of values):

$$\frac{\Gamma \vdash e_1 \Rightarrow \texttt{int} \qquad \Gamma \vdash e_2 \Rightarrow \texttt{int}}{\Gamma \vdash e_1 + e_2 \Rightarrow \texttt{int}} \text{ [exp]}$$

4. Variable assignments:

$$\frac{\Gamma[x \mapsto \tau] \vdash e \Rightarrow \rho \qquad \rho \prec \tau}{\Gamma[x \mapsto \tau] \vdash x = e \Rightarrow \tau} \text{ [varasm]}$$

5. Let-in bindings (only one binding variable):

$$\frac{\Gamma \vdash e \Rightarrow T \qquad \Gamma[x \mapsto T''] \vdash e' \Rightarrow T' \qquad T \prec T''}{\Gamma \vdash \texttt{let } T'' \ x = e \ \texttt{ in } e' \Rightarrow T'} \text{ [letInExp}_1\text{]}$$

6. Let-in bindings (general):

$$\frac{\Gamma \vdash dec \Rightarrow E \qquad \Gamma \oplus E \vdash exp \Rightarrow \tau}{\Gamma \vdash \texttt{let } dec \texttt{ in } exp \Rightarrow \tau} \text{ [letInExp}_2\text{]}$$

7. Sequence of expressions (program):

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \ldots \Gamma \vdash e_n \Rightarrow \tau_n}{\Gamma \vdash e_1; e_2; \ldots; e_n \Rightarrow \tau_n} \text{ [prog]}$$

In our framework, function definitions and class definitions doesn't have type inference rules, because they have no contributions when calculating the type of syntax trees in any FOOL program. Each of above mentioned type inference rules has corresponding Java code at the type checking phase of FOOL compiler, and if anything goes wrong, the compiler will throw a runtime exception immediately (or at the end of type checking stage sometimes), and refuse to continue further phases.

5 Implementation in Java

5.1 Overall architecture

The whole program is divided into four packages:

1. `it.unibo.FOOL`, the language-dependent compiler of FOOL$^+$ language.
2. `it.unibo.FOOL.env`, a language-independent environment package, providing scopes and symbol classes.
3. `it.unibo.FOOL.svm`, a simple stack-based bytecode virtual machine (interpreter) and assembler.
4. `it.unibo.FOOL.test` (and two sub-packages), the unit tests based on JUnit framework.

The entry class is `it.unibo.FOOL.Main`, providing a `main()` function as the sole entry of the whole program. Given different arguments (supported input sources: standard input stream, source code files, bytecode files), the entry class has different behaviors.

In package `it.unibo.FOOL`, the FOOL compiler was implemented by different compiling phases, each phase has one single Java file defining one class:

1. `CountPhase.java`, counting the number of nodes and terminals in the parsing tree. This was a homework exercise. Our code is based on ANTLR's listener API and is extremely simple.
2. `DefPhase.java` Symbolic analysis (Part I). In this phase, all scopes and symbols representing variables, functions and classes are defined.

3. `RefPhase.java` Symbolic analysis (Part II). In this phase, all function and variable accesses were checked. Forward references and recursive functions were supported in this project because of the second-pass parser tree walking in this stage.
4. `TypePhase.java` Semantic analysis (mainly type checking). In this phase, the type of each parser tree nodes were decided, and all type errors were reported at this stage.
5. `EmitPhase.java` Code generation. This is the last compiler phase, it use all information collected in previous phases to generate bytecode for the input program.

The following Java code piece shows how each of above mentioned phases connected with each other:

```
FOOLLexer lexer = new FOOLLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
FOOLParser parser = new FOOLParser(tokens);
parser.setErrorHandler(new ErrorStrategy());
ParseTree tree = parser.prog();
ParseTreeWalker walker = new ParseTreeWalker();
DefPhase def = new DefPhase();
walker.walk(def, tree);
RefPhase ref = new RefPhase(def);
walker.walk(ref, tree);
TypePhase typ = new TypePhase(def);
walker.walk(typ, tree);
EmitPhase emit = new EmitPhase(typ);
emit.visit(tree);
```

Also noticed that, the first 4 phases were all based on ANTLR's listener interface, while the last phase (code generation) has to use the traditional visitor interface to gain finest controls on the order of parsing tree walking. There's no much to say about the source code of the compiler front-end (the first 4 phases), except for the architecture.

One of the biggest difference between our project and professor's reference code is that, in our project there's no AST (abstracted syntax tree) constructed from the parsing tree generated by ANTLR. Recall the purpose to define an AST class is to hold extra information that ANTLR is parsing tree nodes have no ability to hold, but in ANTLR v4, such needs can be fulfilled by doing *annotations on the parsing tree*, i.e. we can define any number of hash tables which has parser tree as key and arbitrary values can be connected with specific parser tree nodes. The key class provided by ANTLR is `ParseTreeProperty`. Refer to [2] for details of ANTLR v4's new feature.

The reason we can get rid of the AST is because that, in such a simple compiler we didn't do any kind of optimization in which the "shapes" of AST must be changed. In another words, we never need to rewrite any portion of the parsing tree. If such optimizations were required, an AST must be constructed instead. ANTLR's parsing tree annotations are not enough to handle such needs.

5.2 The environment package – scopes, types and symbols

The environment package, `it.unibo.FOOL.env`, provides classes for holding all semantic components (variables, functions, types, classes and scopes) of the program. We don't have a symbol table here, instead we have chosen to use the "spaghetti" tree of scopes for holding all symbols defined in the user code. Most classes in this package were borrowed from [1] with some modifications, the only class that the author has written from ground is the class for generic functions.

There're two interfaces and 10 classes defined in this package. Roughly speaking, these classes can be divided into scopes and symbols, while some classes belongs to both categories. Below is an overview of the purpose and inheritance relations for each class in this package:

1. `Scope` (interface). Scope is an abstract concept, each scope has a name, the ability to hold some symbols and a link to its "enclosing scope" up to the global scope.

   ```
   public interface Scope {
       public String getScopeName();
       public Scope getEnclosingScope();
       public void define(Symbol sym);
       public Symbol resolve(String name);
   }
   ```

2. `BaseScope` (class). The base class of scopes, implementing the ability to define and resolve symbols.

3. `GlobalScope` (class, extends: `BaseScope`). The single global scope for any program, it has the name "global".
4. `LocalScope` (class, extends: `BaseScope`). It represent the scopes created by anonymous LET bindings, it has the name "local" always.
5. `Type` (interface). Type is an abstract concept, each type must has a name, an index as integer, and a method for testing if it can be assigned to another type:

```
public interface Type {
    public String getName();
    public int getTypeIndex();
    public boolean canAssignTo(Type destType);
}
```

6. `Symbol` (class). The fundamental class for any symbol in the program. A symbol must have its name, scope and type.
7. `VariableSymbol` (class, extends: `Symbol`). The class for variables (including class slots and function parameters).
8. `ScopedSymbol` (abstract class, extends: `Symbol`, implements: `Scope`). A scope symbol is the base class for functions and classes which can hold other symbols.
9. `MethodSymbol` (class, extends: `ScopedSymbol`). The class for functions and methods (they're the same thing in our view). It has an ordered parameters map holding symbols representing all its parameters, and it has a return type (to be explained).
10. `ClassSymbol` (class, extends: `ScopedSymbol`, implements: `Type`). A class is also a type, while it's also a scope symbol because it owns member variables (slots).
11. `BuiltinTypeSymbol` (class, extends: `Symbol`, implements: `Type`). The class for two builtin types (boolean and integer).
12. `GenericFunction` (class, extends: `ScopedSymbol`). CLOS-style generic functions (gf). A gf owns all methods with the same name and number of arguments. Refer to [4] for details about generic functions. In our project, gf has essential roles when generating bytecode for method calls on polymorphic typed arguments. The design and implementation of this class is totally the author's work, although the algorithm for dynamic method dispatching is taken from [4].

5.3 The bytecode and its assembler, interpreter

The package `it.unibo.FOOL.svm` has implemented an language-independent, standalone bytecode instruction set together with its assembler and interpreter in Java. The base code is again from [1], but we have done an important modification.

We call the instructions 'bytecodes' because we can represent each instruction with a unique integer code from 0..255 (a byte?s range). In [1] and also professor's referential implementation, an text-base assembly language is always implemented, and then ANTLR is used again to parse that assembly language and then generate real bytecodes for execution in the interpreter. We think this is really unnecessary, because the assembly language can only be parsed into a flat list of instructions.

In this project, instead, we let the compiler backend (code generation phase) directly generated real bytecodes – instructions filled into Java `byte[]` arrays. There's still an assembly language, but it's not text-based. Instead it has easy-to-use APIs in Java.

In Table 1 the instruction set is listed. Noticed that, all float related instructions were not used in this project, and the last a bunch of so-called "dynamic" instructions were creation by the author for special purposes (but they're not all used too). The difference between dynamic instructions and their corresponding normal instructions is that, dynamic instructions take operands on stack, and the value may be a value loaded from other variables. (thus the operands works like a pointer) The choice for a reasonable instruction set is a balance between performance, expressivity and human readability (for debugging purpose).

This bytecode assembler and interpreter can be used alone, independently with other code in this project. Below is an example taken from our unit test set, showing how to write programs directly in bytecode:

```
Assembler assem = new Assembler();

assem.defineFunction("add", 2, 0);
assem.gen("load", 0); // load 1st argument into stack
assem.gen("load", 1); // load 2nd argument into stack
```

Table 1. the bytecode instruction set

name	code	operand	functionality
iadd	1	-	integer add
isub	2	-	integer sub
imul	3	-	integer mul
ilt	4	-	integer less than
ieq	5	-	integer equal
fadd	6	-	float add
fsub	7	-	float sub
fmul	8	-	float mul
flt	9	-	float less than
feq	10	-	float eq
itof	11	-	int to float
call	12	function	call function
ret	13	-	return with/without value
br	14	label	branch
brt	15	label	branch if true
brf	16	label	branch if false
cconst	17	integer	push constant char
iconst	18	integer	push constant integer
fconst	19	pool	push constant float
sconst	20	pool	push constant string
load	21	integer	load from local context
gload	22	integer	load from global memory
fload	23	integer	field load
store	24	integer	store in local context
gstore	25	integer	store in global memory
fstore	26	integer	field store
print	27	-	print stack top
struct	28	integer	push new struct on stack
null	29	-	push null onto stack
pop	30	-	throw away top of stack
halt	31	-	halt machine
aconst	32	-	push constant address
pconst	33	-	push constant pointer
dcall	34	-	(dynamic) call function
dbr	35	-	(dynamic) branch
dbrt	36	-	(dynamic) branch if true
dbrf	37	-	(dynamic) branch if false
dload	38	-	(dynamic) load from local context
dgload	39	-	(dynamic) load from global memory
dload	40	-	(dynamic) field load store in local context
dstore	41	-	(dynamic) store in local context
dgstore	42	-	(dynamic) store in global memory
dfstore	43	-	(dynamic) field store
dstruct	44	-	(dynamic) push new struct on stack

```
assem.gen("iadd");
assem.gen("ret");

assem.defineFunction("main", 0, 0);
assem.gen("iconst", 1);
assem.gen("iconst", 2);
assem.gen("call", new Function("add"));
assem.gen("halt");
assem.check();

Disassembler disasm = new Disassembler(assem);
disasm.disassemble();
Interpreter vm = new Interpreter(assem);
vm.setTrace(true);
vm.exec();
```

It's also possible to define labels and use them in a forward referenced way:

```
Assembler assem = new Assembler();
Label end = assem.newLabel("end");

assem.gen("br", end);
assem.setLabel(end);
assem.gen("halt");
assem.check();

Disassembler disasm = new Disassembler(assem);
disasm.disassemble();
Interpreter vm = new Interpreter(assem);
vm.exec();
```

Check `it.unibo.FOOL.test.LoopTest` and other classes for more complicated examples. In next section, when talking about code generations for the FOOL+ language, we'll explain more about the assembly interface.

The interpreter (virtual machine) has the following characters:

1. It has no heap management facility. Instead, it depends on the GC of Java language. The instruction `struct` takes an integer parameter and create object of the class `StructSpace`:

   ```
   public class StructSpace {
       Object[] fields;

       public StructSpace(int nfields) {
           fields = new Object[nfields];
       }

       public String toString() {
           return Arrays.toString(fields);
       }
   }
   ```

   To some extents, such objects are universal data types: they can holding any Java object including themselves, just like the List data type in Lisp language family. So it's obviously enough for representing an object of any class in our FOOL language.
2. When doing function calls, instead of having the large array as frame stack, we create a new Java object of class `StackFrame`, holding a link to last stack frame:

   ```
   public class StackFrame {
       Function sym;      // associated with which function?
       int returnAddress; // the instruction following the call
       Object[] locals;   // holds parameters and local variables

       public StackFrame(Function sym, int returnAddress) {
           this.sym = sym;
           this.returnAddress = returnAddress;
           locals = new Object[sym.nargs + sym.nlocals];
   ```

```
        }
    }
```

These designs are based on the fact that, neither heap management (and GC) nor the interpreter itself is the focus in this project (and the course). Thus these parts were designed and implemented as simple as possible. The key part of this project, instead, is to find a way to compile a complex object oriented programming language (with nested functions) into a bytecode which has only very simple instructions.

6 Code generation

In this section, we will talk about several main difficulties when generating bytecode for FOOL programs, and our solution.

6.1 Accessing variables in bytecode

The highlight of this project actually starts here. All previous work (compiler front-end, environment packages, bytecode assembler and interpreter), although they're precise work, is relative easy and straightforward, also because they were well taught by professor. In the part of code generation, we found the idea of "access link" difficult to actually implement, and there's no reference code.

Recall that, we have a simple programming language called FOOL+. It lacks of many useful features, even may not be Turing-complete [2]. But it's still a challenge to compile it into the bytecode instruction set that we have defined. The first difficulty is, how to access local variables at bytecode level? At runtime, there's no symbols at all, there's no variables at all, all memory operations must be converted into store/load instructions on indexes.

For example, consider the following code:

```
let int x = 1;
    int foo(int n)
        let int m = 2; in x + m + n;
in foo(3);
```

First of all, in our setting, the outer variable x is *not* a global variable, because we found that, handling all variables declared in all LET bindings in a unique way could make the compiler source code more reasonable and a litter shorter. That's why we have introduced the ability to define "real" global variables at a higher level in the program, outside of the LET binding in our enhanced core FOOL grammar.

Therefore, we're looking at a local variable x declared in the outer LET binding, and another variable m declared in the inner LET binding, together with the variable n passed in as the parameter of the function foo between the inner and outer LET binding. So how can an expression like $x + m + n$ access to the variable x? Is such an access "read only" or also "writable"? (suppose the FOOL language had ability the do variable assignments, which is not possible in current grammar setting.)

We want to consult with behaviors in other programming languages first. In Common Lisp, above expression has the following version:

```
CL-USER 1 > (let ((x 1))
              (labels ((foo (n)
                         (let ((m 2))
                           (+ x m n))))
                (foo 3)))
6
```

Above expression returns 6 in any Common Lisp platform. Common Lisp supports changing the value of variable by `SETQ` command. We want to know what happens if inside the local function `foo` the value of x is changed:

```
(let ((x 1))
  (labels ((foo (n)
             (let ((m 2))
               (setq x 2)
               (+ x m n))))
    (foo 3)
    x))
```

[2] With recursive function supported, maybe it is Turing-complete already

Unfortunately the return value is 2. This seems confirmed that, there is indeed an "access link" from the function `foo` to its enclosing scope and it has ability to modified local variables defined in outer scopes.

In Standard ML, however, such behavior is impossible, simply because there's no way to modify variables:

```
> let val x = 1;
      fun foo (n) = let val m = 2;
                    in x+m+n end;
   in foo(3) end;
val it = 6: int
```

In Standard ML (and perhaps all ML language family), to be able to modify the value of variables, the variables themselves must be declared in a special way, which makes them in fact a pointer holding values. Then it will be no surprise that such pointers can be passed into any inner scope by value, and the data it pointed can be freely modified as long as the pointer address is known.

In this project, we're going to take the approach from ML language family but Lisp family, by *not using access links*. Instead, we found that, *outer variables are nothing but extra hidden parameters to the inner functions*. In another words, our FOOL compiler will treat above FOOL program code as equivalent to the following version:

```
let int x = 1;
    int foo(int x, int n)
        let int m = 2; in x + m + n;
in foo(x, 3);
```

That is, implicit passing of all outer local variables as leading arguments to any inner function calls. In this way, we're actually making a copy of all these local variables. But such copies doesn't hurt the possibly to modify data structures (objects) holding by outer variables, because what's copied is just the references (pointer addresses) to these heap-allocated data.

The advantage of this design is that, now we can allocate an unique ID to every local variables as their offset in the local data area of stack frames. And there's no need to have access links, no need for any function to access local data of other functions. And that unique ID is valid for every appearance of the variable in any function. Mapping all these to bytecode level, it's nothing but an `load` instruction with that ID as operand.

From the view of source code, this solution consists with the following parts:

1. In the class definition of `VariableSymbol`, we have added a new member variable called `id`:

   ```
   public class VariableSymbol extends Symbol {
       int id; // the ID of symbol is used by assembler
       ...
   }
   ```

2. In the abstract class definition of `ScopedSymbol` (parent of `MethodSymbol` and `ClassSymbol`), we have added a new member variable called `next_id`:

   ```
   public abstract class ScopedSymbol extends Symbol implements Scope {
       Scope enclosingScope;
       int next_id = 0; // the next available ID for new symbols
       ...
       public int getID() { return id; }
   }
   ```

3. In the member function `define` of `ScopeSymbol`, whenever the defined symbol is a variable, we assigned it a new ID according the current last ID allocated in the scope (a function or a class):

   ```
       public void define(Symbol sym) {
           getMembers().put(sym.name, sym);
           sym.scope = this; // track the scope in each symbol
           if (sym instanceof VariableSymbol) {
               VariableSymbol var = (VariableSymbol) sym;
               var.id = next_id++; // allocate a new ID for the var
           }
       }
   ```

4. In code generation of any variable expression, the generated bytecode is just a `load` instruction to local store (or global store if the variable belong to global scope):

```
/* Code generation of variable references */
public ParseTree visitVarExp(FOOLParser.VarExpContext ctx) {
    String name = ctx.ID().getText();
    VariableSymbol sym = (VariableSymbol) currentScope.resolve(name);
    assert (sym != null);

    int index = sym.getID();
    if (sym.getScope().getScopeName() == "global")
        assem.gload(index);
    else
        assem.load(index);
    return ctx;
}
```

5. Whenever a new scope (could be: function, let binding, class) is defined, the new scope must "inherit" the last ID from its enclosing scope:

```
public void enterLetInExp(FOOLParser.LetInExpContext ctx) {
    LocalScope s = new LocalScope(currentScope);
    saveScope(ctx, s);
    // An offset to the IDs of local variables
    if (currentScope.getScopeName() != "global") {
        int next_id = currentScope.getNextID();
        s.setNextID(next_id);
        System.out.println("base␣ID␣of␣" + s + "␣is␣" + next_id);
    }
    currentScope = s;
}
```

   Noticed that, it's possible that the enclosing scope will have new variables defined in contexts after current position, and the final "last ID" is bigger than the current value. However this won't be a problem, because the forward reference is only supported for functions. (Otherwise we could move the ID assigning process from `DefPhase` to `RefPhase` where all variables are known.

With this solution, we're capable to freely access global, local variables together with class slots at bytecode level, using the ID stored in `VariableSymbol` class.

6.2 Initialization of objects

From the view of bytecode, an object is nothing but a structure holding all its slots and parents' slots. Here we have taken the CLOS-like approach: if a local slot has the same name as any parent slot, they're considered as the same slot. (This seems a requirement in the project specification) Consider the following example:

```
class A = object int x = 5; end;
class B = object inherit A; int x = 3; int y = 2; end;
let A a = new B; in print a.x;
```

The expected printing result is 3. Here the slot x in class B is the same slot as in class A, thus when we assign an object of class B into a variable a of class A and then try to access the slot x, we should get the initialization value 3 in the class definition of B. (We'll talk polymorphic types in next section)

From the view of bytecode, a class definition is nothing but an initialization function: it creates the structure and fill the initial values of slots (if there is) as part of the function, and then return the object. Such an initialization function will be called whenever expressions like `new B` or `new B()` appears in the program. Such an initialization function must also call initialization function of the parent class, and this call must be made before the initialization of local slots. But if the initialization function of all parent classes must be called too, obviously the structure creation process must be outside these function calls, and before calling the initialization functions (otherwise each parent class will create their own structure).

With all these issues considered, together with the needs for identifying the class for any object at run time, we have made the following design decisions:

1. Each class definition (say, class A) will result into two functions, `A_init` (*init* function) and `A_new` (*new* function).
2. The *new* function is the one being called whenever a new object is created by calling `new` operator in the program.
3. The *new* function: 1) creates a empty structure representing the object, 2) call *init* function of the current class, 3) insert runtime identification to the object, and 4) return the object.
4. The *init* function: 1) call parent class' *init* function on the input object, 2) initialize all local slots.

Above approach is implemented in the `visitDefcls` method of *EmitPhase*.

6.3 Polymorphic types

Polymorphic types is a phenomenon that the type (or class) cannot be fully determined at compile-ime. Polymorphic types could appear in two situations:

1. Variable assignments. If B is a subclass of A, then assigning an object b of class B to a variable a of class A will cause the variable a having polymorphic types.
2. If-then-else expression: If the type of then branch and else branch is different but has a subclass (or subtype) relation, and if the value of condition expression cannot be decided at compile-time, then the whole If-then-else expression has polymorphic types.

We think Polymorphic is a property that must be assigned to both expressions and variables during the compiler's type analysis phase. Polymorphic expressions and variables work like virus: it will infect all upper expressions and could pass the method calls. Such a property is useful when doing method calls: instead of calling a specific method, now we have to call generic function instead and let the gf to decide which actual method to be called at runtime.

As a classic example, the following Java code demonstrated the limitation of message-passing model: (from Chapter 16 of *Practical Common Lisp* [5]:

```
public class A {
  public void foo(A a) { System.out.println("A/A"); }
  public void foo(B b) { System.out.println("A/B"); }
}
public class B extends A {
  public void foo(A a) { System.out.println("B/A"); }
  public void foo(B b) { System.out.println("B/B"); }
}
public class Main {
  public static void main(String[] argv) {
    A obj = argv[0].equals("A") ? new A() : new B();
    obj.foo(obj);
  }
}
```

In the `main` method of Class `Main`, the actual class of variable `obj` cannot be fully decided at compile-time, because it depends on the command line argument at runtime. Thus the variable `obj` is polymorphic. In Java (and C++), methods belong to their class, and overloaded methods are chosen at compile time: the two `foo()` method in each class must be decided according to the class of their input arguments, and in this case the `foo(A a)` method will be always the choice, because the declared type of `obj` is A, even if it's polymorphic. Thus above Java code has the following behaviors based on different input arguments at runtime:

```
$ java com.gigamonkeys.Main A
A/A
$ java com.gigamonkeys.Main B
B/A
```

This is not perfect, because what we really want in the second case is to call the `foo(B b)` method and output `"B/B"` instead.

We think above issue has no solution in any object-oriented design based on message-passing. But in our project, we do have a solution, which is based on the following ideas:

1. In our OO design, methods are outside of class, and are managed by generic function with the same name.

2. Whenever a method is called on polymorphic arguments, instead of calling specific method, call the generic function instead.
3. In generic function, the precise type of each input arguments are check at runtime, and based on the actual argument types the most applicable method is chosen and called.

It's commonly accepted that, supporting polymorphic types and dynamic method dispatching will result into performance issues. That's again confirm in our project, because the amount of bytecode that must be executed is several times to the similar program without polymorphic typed expressions.

To actually support polymorphic types, we have defined a new parse tree property called `polys` at the type checking phase:

```
public class TypePhase extends FOOLBaseListener {
    // polymorphic annotations to the parsing tree
    ParseTreeProperty<Boolean> polys = new ParseTreeProperty<Boolean>();
    ...
}
```

And we also need to add a new member variable in class `VariableSymbol`:

```
public class VariableSymbol extends Symbol {
    boolean polyp; // Polymorphic type?
    ....
    public void setPoly(boolean b) { polyp = b; }
    public boolean polyp() { return polyp; }
}
```

The first place where polymorphic flag is set to true, is the type checking code for If-then-else expressions:

```
    public void exitIfExp(FOOLParser.IfExpContext ctx) {
        Type cond = getType(ctx.cond);
        Type thenBranch = getType(ctx.thenBranch);
        Type elseBranch = getType(ctx.elseBranch);
        if (cond != tBOOL) {
            System.err.println("[exitIfExp] the condition is not of type boolean.");
            on_error = true;
            return;
        }
        if (thenBranch == elseBranch)
            saveType(ctx, thenBranch);
        else if (thenBranch.canAssignTo(elseBranch)) {
            // thenBranch is more specific, choose the other
            saveType(ctx, elseBranch);
            setPoly(ctx, true);
        } else if (elseBranch.canAssignTo(thenBranch)) {
            // elseBranch is more specific, choose the other
            saveType(ctx, thenBranch);
            setPoly(ctx, true);
        } else { // two irrelevant types
            System.err.println("[exitIfExp] type mismatch in IfExp: " + ctx.getText());
            on_error = true;
        }
    }
```

The second place is the type checking for variable assignment, and not only the expression but also the variable must be marked as polymorphic:

```
    public void exitVarasm(FOOLParser.VarasmContext ctx) {
        ParseTree lhs = ctx.vardec();
        ParseTree rhs = ctx.exp();
        Type lhsType = getType(lhs);
        Type rhsType = getType(rhs);
        if (rhsType.canAssignTo(lhsType)) { // rhs is more specific
            saveType(ctx, lhsType);
            // Variable assignment is the beginning of Polymorphic types, such
            // information will pass upwards and be contagious to upper nodes
```

```
            // and also variable symbols.
            boolean flag = getPoly(rhs);
            if (lhsType != rhsType || flag) {
                setPoly(ctx, true);
                String name = ctx.vardec().ID().getText();
                VariableSymbol var = (VariableSymbol) currentScope.resolve(name);
                var.setPoly(true);
                System.out.println("The␣type␣of␣" + name + "␣is␣polymorphic!");
            }
        } else {
            System.err.println("type␣mismatch␣in␣var␣assignment:␣" + rhsType + "␣to␣" + lhsTy
            on_error = true;
        }
    }
```

Once a variable has polymorphic types, such an information will be transferred to all expression containing it, starting from the variable expression itself:

```
    // value : ID
    public void exitVarExp(FOOLParser.VarExpContext ctx) {
        String name = ctx.ID().getText();
        VariableSymbol var = (VariableSymbol) currentScope.resolve(name);
        Type typ = var.getType();
        if (typ == null) {
            System.err.println("[exitVarExp]␣no␣type␣information␣for␣variable:␣" + name);
            on_error = true;
        }
        saveType(ctx, typ);
        setPoly(ctx, var.polyp());
    }
```

Finally, all these settings are used when generating method and function calls. If any input argument is polymorphic, call generic function instead of specific method:

```
    /* Code generation of function calls */
    public ParseTree visitFunExp(FOOLParser.FunExpContext ctx) {
        String name = ctx.ID().getText();
        GenericFunction gf = (GenericFunction) currentScope.resolve(name);
        List<ExpContext> args = ctx.exp();
        List<Type> argTypes = this.getTypes(args);

        // lookup the most specific method
        MethodSymbol fun = gf.findMethod(argTypes);
        int last_id = fun.getNextID();
        int base_id = last_id - fun.nargs();

        // 1. push scope arguments
        for (int id = 0; id < base_id; id++)
            assem.load(id);
        // 2. push local arguments, boxing args when gf is called
        if (getPoly(ctx)) {
            for (ExpContext e : ctx.exp()) {
                visit(e);
                Type typ = types.get(e);
                // boxing all primitive types before calling gf
                if (typ instanceof BuiltinTypeSymbol) {
                    assem.iconst(typ.getTypeIndex());
                    assem.call(new Function("_box"));
                }
            }
            assem.call(new Function(gf.cname()));
            gf.setCalled(true);
        } else {
            for (ExpContext e : ctx.exp())
                visit(e);
```

```
            assem.call(new Function(fun.cname()));
        }
        return ctx;
    }
```

6.4 Dynamic method dispatching

In our FOOL+ compiler, functions and methods are the same thing. But whenever a function/method is called with polymorphic typed arguments, instead of calling the method itself, we call the corresponding generic function (gf) instead. The bytecode of generic functions is not generated from any FOOL program code, instead it's written directly in assembly language and generated into bytecodes when it's needed.

A generic function has the following business logics (in order):

1. It gets a list of its methods sorted (at compile-time) by the precedence of their parameter types from left to right.
2. It stores the types of all parameters of these methods into a matrix in the local store of gf.
3. It retrieves the types of input arguments at runtime and store them into local store.
4. It starts a loop into each of its potential method, and check if the type of each input argument is a subclass of the types from the method.
5. In case the first matching method is found, call it.
6. If at the end of loop there's no matching method, signal runtime error saying "no applicable method".

About 1/5 of time and Java source code in this project were dedicated for correctly implementing above behavior. Most of these code can be found at the end of `EmitPhase`. And to be able to decide the class for an object at runtime, we must find a way to encode such information into the data structure and also make sure the algorithm for subclass checking can be as easy as possible. And the other important thing is support primitive types (boolean and integer) in this process too, because they can be types of methods' parameters too.

Here is our solution to this important problem, inspired by Common Lisp Object System (CLOS):

1. In the structure created for any object, the first slot is reserved for hold the class information. Then follows by other class slots (i.e. member variables), including local slots and parent slots.
2. The data presenting class information is again a structure, but acts as a list of integers. The last element is always zero, so that the length of list needs not to be mentioned or stored.
3. For all primitive types and classes, when they're created, there's an unique positive integer associated with them. It's called type index.
4. All user-defined class is implicitly a subclass of `standard_object`, which cannot be defined by the user because of the underscore in its name (forbidden by lexer).
5. For each class, there's a *local precedence list* built by putting the type index of itself, followed by the type index of its direct parent, until there's no more direct parents. It's this list being used as the identifier of its class.

For example, in the FOOL language, there're only two primitive types, `fool` and `int`, they have type indexes as 1 and 2. The class `standard_object` was the next, so it has type index as 3. All the rest user-defined classes has distinct type indexes ¿3. Let's say class A has type index 4 and class B has 5. Their class precedence list are the follows:

- Primitive type `bool`: [1 0]
- Primitive type `int`: [2 0]
- Class A: [4 3 0]
- Class B: [5 4 3 0]

And here is the algorithm to decide if any given class (or types) C_1 is a subclass of C_2: *looping over each number in the class precedence list of C_1, return true until it found a number that is the same as the leading number of the class precedence list of C_2, the process terminates until zero is reached, and the result is false.*

Above algorithm is simple enough, and we have implemented it directly in bytecode as the runtime function `_subclassp`:

```
// The function _subclassp accepts the local precedence list of two classes
// (C1 and C2) and return true if C1 is subclass of C2. For example, if C1
// has local precedence list [102; 101; 100; 0] and C2 is [101; 100; 0],
// then C1 is subclass of C2. Only single-inheritance is supported.
//
// Built-in types are supported too, if their local precedence lists are in
// forms like [a; 0], in which a is the type index of built-in types.
protected void emit_subclassp() {
    int nargs = 2;
    int nlocals = 3;
    int C1 = 0, C2 = 1;
    Label restart = assem.newLabel("restart");
    Label success = assem.newLabel("success");
    Label fail = assem.newLabel("fail");
    Label end = assem.newLabel("end_subclassp");

    // 1. initialization
    assem.defineFunction("_subclassp", nargs, nlocals);
    int i = nargs + 0;     // loop variable
    assem.iconst(0);
    assem.store(i);
    int e = nargs + 1;     // end value
    assem.iconst(0);       // type index of root class
    assem.store(e);
    int h = nargs + 2;     // head of C2 (102 here)
    assem.load(C2);
    assem.fload(0);
    assem.store(h);

    // 2. end condition
    assem.setLabel(restart);
    assem.load(C1);
    assem.load(i);         // load i
    assem.dynamic_fload(); // load C1[*i] - API extension!
    assem.load(e);         // load e
    assem.ieq();           // i == e ?
    assem.brt(fail);       // goto end

    // 3. loop body
    assem.load(C1);
    assem.load(i);
    assem.dynamic_fload(); // load C1[*i] - API extension!
    assem.load(h);
    assem.ieq();
    assem.brt(success);

    // 4. increase counter
    assem.load(i);         // load i
    assem.iconst(1);       // 1
    assem.iadd();          // i + 1
    assem.store(i);        // save i

    // 5. restart the loop
    assem.br(restart);

    // 6. return value
    assem.setLabel(success);
    assem.iconst(1);       // return true
    assem.br(end);
    assem.setLabel(fail);
    assem.zero();           // return false
    assem.setLabel(end);
    assem.ret();
```

```
}
```

In the assembly code, we have to use two dynamic data loading instructions, to be able to check the element in structure data at the index stored in another local variable. Such a common facility is usually provided by registers in other machine or bytecode languages.

But there's still one problem: if the method call has arguments of primitive types, to be able to receive class precedence list from these arguments, a "boxing" of arguments must be done before actually calling the generic function. Perhaps this is also the reason why Java also provided the classes for all its primitive types, e.g. `Integer` and `Boolean`. The boxed data is only useful for generic function to retrieve type information at runtime, once the calling method has been decided, all data must be unboxed to get the raw primitive data for sending to the methods. We have written the boxing and unboxing code in bytecode too, they're runtime functions `_box` and `_unbox`, the leading underscore makes sure they can not be overloaded by user-defined functions. Here are the implementations:

```
// _box(value, type) create a boxed object [[type 0] value]
protected void emit_box() {
    int nargs = 2, nlocals = 2;
    // indexes in local stack
    int value = 0, type = 1, outer = 2, inner = 3;

    assem.defineFunction("_box", nargs, nlocals);
    assem.alloca(2);        // [_, _]
    assem.store(inner);
    assem.load(type);
    assem.load(inner);
    assem.fstore(0);        // [type _]
    assem.zero();
    assem.load(inner);
    assem.fstore(1);        // [type 0]
    assem.alloca(2);
    assem.store(outer);     // [_, _]
    assem.load(inner);
    assem.load(outer);
    assem.fstore(0);        // [[type 0] _]
    assem.load(value);
    assem.load(outer);
    assem.fstore(1);        // [[type 0] value]
    assem.load(outer);
    assem.ret();
}

// The system function _unbox() unbox its arguments and returns the
// 2nd struct element back.
// If the input argument is an object [[? non-zero ... 0] ? ? ...],
// then it's returned directly. The key is to see if the "non-zero" position
// is zero or not.
protected void emit_unbox() {
    int nargs = 1;
    int nlocals = 0;
    Label unbox = assem.newLabel("unbox");
    Label end = assem.newLabel("end_of_unbox");

    assem.defineFunction("_unbox", nargs, nlocals);
    assem.load(0);     // load arg
    assem.fload(0);    // load arg[0]
    assem.fload(1);    // load arg[0][1]
    assem.zero();      // push 0
    assem.ieq();       // arg[0][1] == 0 ?
    assem.brt(unbox); // if so, go to unbox
    assem.load(0);     // else load arg
    assem.br(end);
    assem.setLabel(unbox);
    assem.load(0);     // load arg
    assem.fload(1);    // load arg[1]: the boxed value
```

```
        assem.setLabel(end);
        assem.ret();
    }
```

And let's see again how the boxing is done when calling the gf:

```
    public ParseTree visitFunExp(FOOLParser.FunExpContext ctx) {
        ...
                Type typ = types.get(e);
                // boxing all primitive types before calling gf
                if (typ instanceof BuiltinTypeSymbol) {
                    assem.iconst(typ.getTypeIndex());
                    assem.call(new Function("_box"));
                }
        ...
    }
```

With all these work, the following FOOL program (part of the unit test) finally worked (`foo(B, B)` is expected to call):

```
class A = object end;
class B = object inherit A; end;
let int foo(A o1, A o2, int x) 1+x;
    int foo(A o1, B o2, int x) 2+x;
    int foo(B o1, A o2, int x) 3+x;
    int foo(B o1, B o2, int x) 4+x;
    A obj = new B();
in foo(obj, obj, 6);

// result: 10
```

6.5 Tail-recursion optimization

This is really not related to object-oriented features. But it's excellent compiler optimization examples that can be only done at bytecode (or machine code) level.

It's well known that, recursive functions can be carefully written to shift their recursive calls to the end of function. For instance, the following function calculated the factorial number 10!:

```
let int fact(int n)
     if (n==1) then {1} else {n*fact(n + -1)};
 in print fact(10);

// result: 3628800
```

When we trace the execution process of above code, near the end we see the the increasing and then decreasing process of both operands and frame stacks:

```
  0055:ret      stack=[ 10 9 8 7 6 5 4 3 2 1 ], calls=[ _main fact(int) fact(int) fact(int) f
  0054:imul     stack=[ 10 9 8 7 6 5 4 3 2 1 ], calls=[ _main fact(int) fact(int) fact(int) f
  0055:ret      stack=[ 10 9 8 7 6 5 4 3 2 ], calls=[ _main fact(int) fact(int) fact(int) fac
  0054:imul     stack=[ 10 9 8 7 6 5 4 3 2 ], calls=[ _main fact(int) fact(int) fact(int) fac
  0055:ret      stack=[ 10 9 8 7 6 5 4 6 ], calls=[ _main fact(int) fact(int) fact(int) fact(
  0054:imul     stack=[ 10 9 8 7 6 5 4 6 ], calls=[ _main fact(int) fact(int) fact(int) fact(
  0055:ret      stack=[ 10 9 8 7 6 5 24 ], calls=[ _main fact(int) fact(int) fact(int) fact(i
  0054:imul     stack=[ 10 9 8 7 6 5 24 ], calls=[ _main fact(int) fact(int) fact(int) fact(i
  0055:ret      stack=[ 10 9 8 7 6 120 ], calls=[ _main fact(int) fact(int) fact(int) fact(in
  0054:imul     stack=[ 10 9 8 7 6 120 ], calls=[ _main fact(int) fact(int) fact(int) fact(in
  0055:ret      stack=[ 10 9 8 7 720 ], calls=[ _main fact(int) fact(int) fact(int) fact(int)
  0054:imul     stack=[ 10 9 8 7 720 ], calls=[ _main fact(int) fact(int) fact(int) fact(int)
  0055:ret      stack=[ 10 9 8 5040 ], calls=[ _main fact(int) fact(int) fact(int) fact(int)
  0054:imul     stack=[ 10 9 8 5040 ], calls=[ _main fact(int) fact(int) fact(int) ]
  0055:ret      stack=[ 10 9 40320 ], calls=[ _main fact(int) fact(int) fact(int) ]
  0054:imul     stack=[ 10 9 40320 ], calls=[ _main fact(int) fact(int) ]
  0055:ret      stack=[ 10 362880 ], calls=[ _main fact(int) fact(int) ]
  0054:imul     stack=[ 10 362880 ], calls=[ _main fact(int) ]
  0055:ret      stack=[ 3628800 ], calls=[ _main fact(int) ]
```

It's also well known that, the need for the big size of data stacks can be superseded, if the program can be rewritten in the following way:

```
let int fact(int n, int acc)
     if (n==1) then {acc} else {fact(n + -1, n*acc)};
 in print fact(10, 1);

// result: 3628800
```

The fact that new function need one more argument is not of issues in practice, because we can easily use another single-parameter function to wrap it. This new function has the property of "tail recursion", and it has a bounded use for data stacks which is irrelevant to the value of input arguments:

```
  0005:load       0stack=[ ], calls=[ _main fact(int,int) fact(int,int) fact(int,int) fact(in
  0010:iconst     1stack=[ 1 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int) fact(
  0015:ieq          stack=[ 1 1 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int) fa
  0016:null         stack=[ 1 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int) fact
  0017:ieq          stack=[ 1 0 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int) fa
  0018:brt       33stack=[ 0 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int) fact
  0023:load       1stack=[ ], calls=[ _main fact(int,int) fact(int,int) fact(int,int) fact(in
  0028:br        60stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) fact(int,int
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) fact(int,int) ]
  0060:ret          stack=[ 3628800 ], calls=[ _main fact(int,int) ]
```

As we can see, now the data (operand) stack is stable, but the stack frame is still unbounded, and the last 10 instruction were all about returning from previous functions.

If we took a look at the generated bytecode (in disassembled text form), we found that the "call" instruction is the last second instruction in the function:

```
.fact(int,int) nargs=2 nlocals=0
begin_fact(int,int):
  0005: load       0
  0010: iconst     1
  0015: ieq
  0016: null
  0017: ieq
  0018: brt        33 [else]
  0023: load       1
  0028: br         60 [endif]
else:
  0033: load       0
  0038: iconst     -1
  0043: iadd
  0044: load       0
  0049: load       1
  0054: imul
  0055: call       #0:fact(int,int)@5
endif:
  0060: ret
```

In this project, we have done an well-known optimization, before generating the "ret" instruction: we look back the last generated instruction and if it's a "call" into current function, instead of calling itself, we put all arguments back the local store and jump directly back to the beginning of current function. So the above assembly code now becomes this:

```
.fact(int,int) nargs=2 nlocals=0
begin_fact(int,int):
  0005: load       0
```

```
  0010: iconst     1
  0015: ieq
  0016: null
  0017: ieq
  0018: brt        33 [else]
  0023: load       1
  0028: br         70 [endif]
else:
  0033: load       0
  0038: iconst     -1
  0043: iadd
  0044: load       0
  0049: load       1
  0054: imul
  0055: store      1 /// HERE !!!
  0060: store      0 /// HERE !!!
  0065: br         5 [begin_fact(int,int)] /// and HERE !!!
endif:
  0070: ret
```

With such an optimization, now the trace of execution shows that, either data stack or frame stack are bounded:

```
.fact(int,int) nargs=2 nlocals=0
  0005: load       0            stack=[ ], calls=[ _main fact(int,int) ]
  0010: iconst     1            stack=[ 2 ], calls=[ _main fact(int,int) ]
  0015: ieq                     stack=[ 2 1 ], calls=[ _main fact(int,int) ]
  0016: null                    stack=[ 0 ], calls=[ _main fact(int,int) ]
  0017: ieq                     stack=[ 0 0 ], calls=[ _main fact(int,int) ]
  0018: brt        33           stack=[ 1 ], calls=[ _main fact(int,int) ]
  0033: load       0            stack=[ ], calls=[ _main fact(int,int) ]
  0038: iconst     -1           stack=[ 2 ], calls=[ _main fact(int,int) ]
  0043: iadd                    stack=[ 2 -1 ], calls=[ _main fact(int,int) ]
  0044: load       0            stack=[ 1 ], calls=[ _main fact(int,int) ]
  0049: load       1            stack=[ 1 2 ], calls=[ _main fact(int,int) ]
  0054: imul                    stack=[ 1 2 1814400 ], calls=[ _main fact(int,int) ]
  0055: store      1            stack=[ 1 3628800 ], calls=[ _main fact(int,int) ]
  0060: store      0            stack=[ 1 ], calls=[ _main fact(int,int) ]
  0065: br         5            stack=[ ], calls=[ _main fact(int,int) ]
.fact(int,int) nargs=2 nlocals=0
  0005: load       0            stack=[ ], calls=[ _main fact(int,int) ]
  0010: iconst     1            stack=[ 1 ], calls=[ _main fact(int,int) ]
  0015: ieq                     stack=[ 1 1 ], calls=[ _main fact(int,int) ]
  0016: null                    stack=[ 1 ], calls=[ _main fact(int,int) ]
  0017: ieq                     stack=[ 1 0 ], calls=[ _main fact(int,int) ]
  0018: brt        33           stack=[ 0 ], calls=[ _main fact(int,int) ]
  0023: load       1            stack=[ ], calls=[ _main fact(int,int) ]
  0028: br         70           stack=[ 3628800 ], calls=[ _main fact(int,int) ]
  0070: ret                     stack=[ 3628800 ], calls=[ _main fact(int,int) ]
```

The cost is very low to implement tail-recursion optimization, in our existing project framework. The only thing we did, is to insert one more function call during the code generation of any function:

```
assem.br(end_of_fun); // jump directly to the end of function
assem.defineFunction(cname, nargs, nlocals);
assem.setLabel(begin_of_fun);
System.out.printf("emiting␣function␣%s␣[nargs:␣%d(%d+%d),␣nlocals:␣%d]\n", cname, nar
        nlocals);
ParseTree body = visit(ctx.body()); // generate function body
// do tail recursion optimization before return
if (tail_rec) {
    tail_recursion_optimization(cname, nargs, begin_of_fun); // HERE !!!
}
assem.ret();
assem.setLabel(end_of_fun);  // jumped here
```

And the inserted function is also not long:

```
/**
 * Tail recursion optimization: if the last instruct is a call to current
 * function, then instead of calling itself, just jump to the beginning of
 * the function (before storing stack args back to frame).
 */
public void tail_recursion_optimization(String name, int nargs, Label begin_of_fun) {
    byte[] code = assem.getMachineCode();
    int last_ip = assem.getLastIP();
    int opcode = code[last_ip];
    Bytecode.Instruction I = Bytecode.instructions[opcode];
    if (I.name == "call") {
        int f = Assembler.getInt(code, last_ip + 1);
        int g = assem.getConstantPoolIndex(new Function(name));
        if (f == g) { // if it's the same function
            assem.resetIP(last_ip);
            // 1. store stack args back to frame
            for (int i = nargs - 1; i >= 0; i--)
                assem.store(i);
            // 2. do branch() instead of call()
            assem.br(begin_of_fun);
            // 3. update all labels at next ip (code[last_ip+5])
            List<Label> labels = assem.getLabels();
            for (Label l : labels) {
                if (l.address == last_ip + 5) {
                    l.address += nargs * 5;
                    l.resolveForwardReferences(code);
                }
            }
        }
    }
}
```

The only tricky thing here, is to make sure all assembly labels still pointing to the correct places after the replacement of old instructions with new instructions inserted before it. And to do this, we also have to access to some internal interface of the assembler. Therefore the whole work must be seen as a hacking in existing framework. After all, optimization is not a focus in this project, so we have no infrastructure to support optimization in a non-hacking way.

Perhaps the only programming language in which tail-recursion optimization is a requirement in the language specification, is the algorithm language Scheme. This is partly because Scheme has no direct support for looping, and if tail-recursion optimization is not supported, there's basically no way to do large loop without exposing the execution stack.

There's another higher level of optimization: tail-call optimization, in which we can actually replace any function call before returning by jumping directly to that function instead. However, in our project framework this is impossible, because our stack frame is Java object holding fix-sized local storage, it's quite hard to reuse the current stack frame. Also it's quite hard to retrieve the starting label of arbitrary functions. It's not impossible but we chose to stop here.

7 Conclusions

In this project, we (the author alone, actually) have built a simple bytecode compiler and interpreter for an object-oriented programming language called FOOL$^+$ in Java. We have slightly extended the core FOOL language specification given by professor as the working basis, then added object-oriented on top of it, with minimized changes at grammar level. The resulting OO extension supports single-inheritance, multi-methods, polymorphic types, and dynamic method dispatching based on generic functions instead of message-passing. We have shown that, a classical multi-method example in Java now has better running result when expressed in our new language with our new compiler. The idea mainly comes from Common Lisp Object System (CLOS) and the related algorithms were from the de-facto Common Lisp language specification [4].

The reference code provided by professor is not used, because we want to follow the recommended API provided by ANTLR v4. Instead we have use sample code provided with two ANTLR books ([2],

[1]) with essential modifications. The biggest difference is that, we don't need an AST, instead we do annotations directly on the original parsing tree generated by ANTLR. We used ANTLR's new listener APIs to implemented the first 4 phases (Count, Def, Ref, Type) of the compiler, and then use the traditional visitor API to implement the code generation phase (compiler backend). This architecture makes sure that source code related to the same compiling phases were written together in the same Java class. The project is bit-perfect although it's just a two-week work (plus 2 days report writing).

The bytecode is stack-based, but the assembler doesn't read assembly code as another text-based language, instead we have designed easy-to-use interface for user to write bytecode programs directly in Java. And to implement dynamic method dispatching, we have written long bytecode programs in hands, and make them part of the runtime library of the OO-based FOOL language.

As a side effect, we have also implemented tail-recursion optimization as a low cost addition when generating bytecode for function definitions.

The project has followed good practices in software engineering. We have provided unit tests written in JUnit framework, and the final "program" can be built into single JAR file containing only the ANTLR runtime but the full parser. The author sincerely hopes that this time professor could successfully open the project.

References

1. Parr, T.: Language Implementation Patterns. Create Your Own Domain-specific and General Programming Languages. The Pragmatic Bookshelf (2010)
2. Parr, T.: The Definitive ANTLR 4 Reference. Building Domain-specific Languages. The Pragmatic Bookshelf (2012)
3. Milner, R.: The Definition of Standard ML. Revised. MIT Press (1997)
4. Steele Jr, G.L.: Common Lisp: The Language, Second Edition. Digitool Press (1990)
5. Seibel, P.: Practical Common Lisp. Apress (2005)