CS164 Programming Languages and Compilers **Spring 2019**

Programming Assignment 3

Assigned: April 10, 2019 **Checkpoint**: April 29 2019 at 11:59pm **Due:** May 7, 2019 at 11:59pm

1 Overview

The three programming assignments in this course will direct you to develop a compiler for ChocoPy, a statically typed dialect of Python. The assignments will cover (1) lexing and parsing of ChocoPy into an abstract syntax tree (AST), (2) semantic analysis of the AST, and (3) code generation.

For this assignment, you are to implement a RISC-V code generator for ChocoPy. This phase of the compiler takes as input the type-annotated AST of a semantically valid and well-typed ChocoPy program, and produces as output RISC-V assembly code. Section 6 describes the version of RISC-V that we will be using, as well as the execution environment used for grading.

This assignment is also accompanied by the ChocoPy RISC-V implementation guide, which is a document that describes in detail the design decisions taken by the reference compiler. Unlike previous assignments, the starter code provided for this assignment is quite extensive. We encourage you to make full use of this code, since it will save you about half the development effort of building a code generator. Reading the accompanying implementation guide is essential to understanding the provided starter code. This assignment can get a bit tedious, so **start early**. However, implementing a code generator can be a very rewarding task, since you will (finally) be able to execute ChocoPy programs and observe their behavior.

2 Getting started

We are going to use the Github Classroom platform for managing programming assignments and submissions.

- Visit `https://classroom.github.com/g/RaVpIcx8` for the assignment. You will need a GitHub account to join.
- It seems that Github classroom is not terribly flexible when it comes to changing teams. Therefore, for this assignment, you'll form new ones, as for PA1. The first team member accepting the assignment should create a new team with some reasonable team name. Team names should start with a capital letter and contain only letters, digits, hyphens, and underscores (no blanks). The second team member can then find the team in the list of open teams and join it when accepting the assignment. A private GitHub repository will be created for your team. It should be of the form `https://github.com/cs164spring2019/pa3-chocopy-code-generation-<team>` where `<team>` is the name of your team.
- Ensure you have Git, Apache Maven and JDK 8+ installed. See Section 3 for more information regarding software.
- If your team name is `<team>`, then clone the git repository:

`https://github.com/cs164spring2019/pa3-chocopy-code-generation-<team>.git`.

It will contain all the files required for the assignment. Your repository must remain private; otherwise, you will get 0 points in this assignment.

- Add the upstream repository in order to receive future updates to this repository. This must be done only once per local clone of your repository. Run

```
git remote add upstream \
    https://github.com/cs164spring2019/pa3-chocopy-code-generation.git
```

- Run `mvn clean package`. This will compile the starter code, which analyzes all declarations in a ChocoPy and emits everything that is needed in the data segment, as well as a skeleton text segment for the top-level statements. Your goal is to emit code for top-level statements as well as for every function/method defined in the ChocoPy program.

- Run the following command to test your analysis against sample inputs and expected outputs—only one test will pass with the starter code:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=..s \
     --run --dir src/test/data/pa3/sample --test
```

Windows users should replace the colon between the JAR names in the classpath with a semicolon: `java -cp "chocopy-ref.jar;target/assignment.jar" ...`. This applies to all `java` commands listed in this document.

3 Software dependencies

The software required for this assignment is as follows:

- Git, version 2.5 or newer: `https://git-scm.com/downloads`
- Java Development Kit (JDK), version 8 or newer: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
- Apache Maven, version 3.3.9 or newer: `https://maven.apache.org/download.cgi`
- (optional) An IDE such as IntelliJ IDEA (free community editor or ultimate edition for students): `https://www.jetbrains.com/idea`.
- (optional) Python, version 3.6 or newer, for running ChocoPy programs in a Python interpreter: `https://www.python.org/downloads`

If you are using Linux or MacOS, we recommend using a package manager such as `apt` or `homebrew`. Otherwise, you can simply download and install the software from the websites listed above. We also recommend using an IDE to develop and debug your code. In IntelliJ, you should be able to import the repository as a Maven project.

4 External Documentation

- RISC-V specification: `https://riscv.org/specifications`
- Venus wiki: `https://github.com/kvakil/venus/wiki`. We are using a modified version of Venus for this course. Section 6 describes our simulator and its differences from the original.

5 Files and directories

The assignment repository contains a number of files that provide a skeleton for the project. Some of these files should not be modified, as they are essential for the assignment to compile correctly. Other files must be modified in order to complete the assignment. You may also have to create some new files in this directory structure. The list below summarizes each file or directory in the provided skeleton.

- `pom.xml`: The Apache Maven build configuration. You do not need to modify this as it is set up to compile the entire pipeline. We will overwrite this file with the original `pom.xml` while autograding.
- `src/`: The `src` directory contains manually editable source files, some of which you must modify for this assignment. Classes in the `chocopy.common` package may not be modified, because they are common to your assignment and the reference implementation / test framework. However, you are free to duplicate/extend these classes in the `chocopy.pa3` package or elsewhere. Section 8 describes in detail how the provided starter code is meant to be extended without requiring any duplication.
  - `src/main/java/chocopy/pa3/StudentCodeGen.java`: This class is the entry point to the code generation phase of your compiler. It contains a single method: `public static String process(String input, boolean debug)`. The first argument to this method will be the typed AST produced by the semantic analyis stage in JSON format, and the return value should be the RISC-V assembly program. The second argument to this method is `true` if the `--debug` flag is provided on the command line when invoking the compiler.
  - `src/main/java/chocopy/pa3/CodeGenImpl.java`: This class contains a skeleton implementation of the abstract class `chocopy.common.CodeGenBase`. You will have to modify this file to emit assembly code for top-level statements and function bodies. Section 8 describes these classes in detail.
  - `src/main/java/chocopy/common/astnodes/*.java`: This package contains one class for every AST-node kind that appears in the input JSON. These are the same classes that were provided in previous assignments.
  - `src/main/java/chocopy/common/analysis/NodeAnalyzer.java`: An interface containing method overloads for every node class in the AST hierarchy. This is the same class that was provided in the previous assignment.
  - `src/main/java/chocopy/common/analysis/AbstractNodeAnalyzer.java`: A dummy implementation of the `NodeAnlyzer` interface. This is the same class that was provided in the previous assignment.

- `src/main/java/chocopy/common/analysis/SymbolTable.java`: This class contains a sample implementation of a symbol table, which is a essentially a map from strings to values of a generic type `T`. This is the same class that was provided in the previous assignment.
- `src/main/java/chocopy/common/analysis/types/*.java`: This package contains a hierarchy of classes for representing types in the typed AST. These are the same classes that were provided in the previous assignment.
- `src/main/java/chocopy/common/codegen/*.java`: These classes contain all the support classes for the extensive starter code provided to you. Section 8 describes these classes in detail, including how you can extend some of them.
- `src/main/java/chocopy/common/codegen/asm/*.s`: These classes contain assembly-language implementations of built-in functions, which `CodeGenBase` copies into the output program. You can use the same technique for adding additional runtime support routines (for things such as string concatenation). Just put such routines in a directory `src/main/java/chocopy/pa3/codegen/asm` and look to see how `CodeGenBase` uses the `emitStdFunc` routines.
- `src/test/data/pa3`: This directory contains ChocoPy programs for testing your code generator.
  - `/sample/*.py` - Sample test programs covering a variety of semantics that you will need to implement in this assignment. Each sample program is designed to test a small number of language features.
  - `/sample/*.py.out.typed` - Typed ASTs corresponding to the test programs. These will be the inputs to your code generator.
  - `/sample/*.py.out.typed.s.result` - The results of executing the test programs. The assembly programs generated by your compiler should produce exactly these results when executed in order for the corresponding tests to pass.
  - `/benchmarks/*.py` - Non-trivial benchmark programs, meant to test the overall working of your compiler. The testing for these programs will be done in the same manner as done for the tests in the `sample` directory, but these tests will have higher weight during grading.
  - `/benchmarks/*.py.out.typed` - Typed ASTs corresponding to the benchmark test programs. These will be the inputs to your code generator.
  - `/benchmarks/*.py.out.typed.s.result` - The results of executing the benchmark programs.

- `target/`: The `target` directory will be created and populated after running `mvn clean package`. It contains automatically generated files that you should not modify by hand. This directory will be deleted before your submission.
- `chocopy-ref.jar`: A reference implementation of the ChocoPy compiler, provided by the instructors.
- `README.md`: You will have to modify this file with a writeup.
- `checkpoint_tests.txt`: List of tests used for grading at the checkpoint (ref. Section 9). This list is same as Appendix A of this document.

6 Execution Environment

The target architecture for this code generation assignment is RV32IM, which is the 32-bit version of RISC-V that supports basic integer arithmetic plus the multiplication (and division) extensions.

In order to execute RISC-V code in a platform-independent manner, we will be using a version of the Venus simulator, which was originally developed by Keyhan Vakil. Venus dictates the execution environment, which includes the initial values of registers, the addresses of the various memory segments, and the set of supported system calls. Section 4 points to some documentation for Venus.

6.1 Venus 164

To support the goals of this project, our version of Venus has been modified—we refer to this variant as Venus 164. The modifications mainly try to make the assembly language conform to the one supported by the official GNU-based RISC-V toolchain.

- `.word` directive: We have added support for emitting addresses in the data segment using the syntax `.word <label>`. Originally, Venus only allowed emitting integer literals.
- `.align` directive: We have added limited support for specifying byte alignment in the data segment. The supported syntax is `.align <n>`, which inserts zero-valued bytes as padding such that the next available address is a multiple of 2^n. Originally, Venus did not support alignment.
- `.string` directive: We have added support for emitting ASCII strings using the syntax `.string <string_in_quotes>`. Originally, Venus supported a directive called `.asciiz` for emitting strings; this still works, but it is not supported by the GNU toolchain. We like to use the standardized version instead.
- `.space` directive: `.space <n>` inserts n 0-bytes into the data segment.
- `.equiv` directive: `.equiv <sym>, <value>` defines the label `<sym>` to have the value `<value>`. Here, `<value>` may be a numeral or another symbol (possibly defined by `.equiv`). As for ordinary labels, the directive may appear after uses of `<sym>`, allowing you to include a value in an assembler instruction before figuring out precisely what that value will be.
- The original Venus supports some extra non-standard pseudo-instructions (such as `seq` and `sgt`). We enforce strict mode in Venus 164: the pseudo-instructions we support include only the ones listed on page 110 of the RISC-V specifications, version 2.2 (ref. Section 4).

The simulator is distributed both as a JAR in our instructional maven repo (for use with the auto-grader) and in web form to enable interactive debugging. The web version of Venus 164 is hosted at the following URL:

`https://cs164spring2019.github.io/venus-simulator`

You can enter RISC-V assembly code in the editor and then switch to the simulator tab to run the program. You can add break-points and step through instructions one at a time to observe changes to the registers and to the memory. The CS164 staff cannot provide support on using the web UI.

7 Assignment goals

The objective of this assignment is to build a code generator for ChocoPy, which takes as input a typed AST corresponding to a ChocoPy program in JSON format, and produces as output a RISC-V assembly program that can execute in the Venus 164 execution environment.

7.1 Running the compiler

7.1.1 Four-step process

The process of executing a ChocoPy program consists of four basic steps:

```
1. java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
       --pass=r <chocopy_input_file> --out <ast_json_file>

2. java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
       --pass=.r <ast_json_file> --out <typed_ast_json_file>

3. java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
       --pass=..s <typed_ast_json_file> --out <assembly_file>

4. java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
       --run <assembly_file>
```

where `<chocopy_input_file>` is a ChocoPy program (usually with a `.py` extension), `<ast_json_file>` is the parsed AST in JSON format (usually with a `.out` extension), `<typed_ast_json_file>` is the type-annotated AST in JSON format (usually with a `.out.typed` extension) and `<assembly_file>` is the compiled RISC-V assembly program (usually with a `.out.typed.s` extension).

7.1.2 Reference implementation

To observe the assembly program produced by the reference implementation, replace step 3 above with the following command:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
     --pass=..r <typed_ast_json_file> --out <assembly_file>
```

7.1.3 Shortcuts: chained commands

To simplify development, you can also club the above commands into a single command that pipes the output of the each phase to the input of the next phase. The combined command to produce an assembly file from an input ChocoPy program (which is equivalent to running steps 1–3) is as follows:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
    --pass=rrs <chocopy_input_file> --out <assembly_file>
```

You can also add `--run` at the end of this chain to actually execute an input ChocoPy program in one step.

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
    --pass=rrs --run <chocopy_input_file>
```

Finally, the command

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
    --pass=rrr --run <chocopy_input_file>
```

will run the combined command with the reference implementation.

In any command, you can omit the `--out <file>` argument to have the result be printed to standard output instead of a file.

7.2 Input/output specification

The interface to your code generation assignment will be the static method `StudentCodeGen.process()`. The input to this method is a typed AST in JSON format, corresponding to a semantically valid and well-typed ChocoPy program. The typed AST will be in the same format that was used as the output format for the previous assignment. The field `inferredType` will be non-`null` for every expression in the AST that evaluates to a value. The output is expected to be a RISC-V assembly program, which is executed in the Venus 164 environment. **The assembly program that your compiler generates need not match the program generated by the reference compiler**.

7.3 Memory Management

In this assignment, all compiled ChocoPy programs will have 32MB of memory to work with. The register `gp` will point to the beginning of the heap before the first top-level statement is executed. Garbage collection (GC) has not yet been implemented in the reference implementation; therefore, newly allocated objects block space for the entire remaining duration of the program. You are not expected to implement GC in this assignment, though the heap and object layouts have been designed in such a way that GC can be easily integrated. The tests used by the auto-grader require far less than 32MB of memory to execute.

7.4 Validation

Testing is performed by executing the generated RISC-V program in Venus 164 and comparing the contents of the output stream with that produced by the reference-implementation-generated program. The program is expected to behave as per the operational semantics defined in the ChocoPy language manual: `chocopy_language_reference.pdf`. The output should contain a sequence of lines, where the ith line corresponds to the string representation of the `str`, `int`, or `bool` object provided as argument to the ith dynamic invocation of the predefined `print` function.

7.5 Error handling

In case of run-time errors, your program is expected to print an appropriate error message and exit with an appropriate exit code. The error messages and exit codes used by the reference implementation are described in `chocopy_implementation_guide.pdf`. Fortunately, you do not have to hand-code the error messages or corresponding exit codes. The errors corresponding to invalid arguments to predefined functions and out-of-memory are generated by the code that has already been provided to you. For errors corresponding to operations on `None`, division by zero, and index out-of-bounds, we have provided you built-in routines that are emitted in the method `CodeGenImpl.emitCustomCode()`. Your generated programs can simply jump to one of these labels when the appropriate condition is met and the error message will be printed for you before aborting the program with an appropriate exit code. You do need to jump to these error handlers exactly *when the appropriate condition is met*. A run-time error is raised when one of the pre-conditions in the operational semantics fails to be true. For example, in the operational rule [DISPATCH], if the object on which a method is dispatched turns out to be the value `None`, then the second line fails to be true; therefore, the run-time error is reported after evaluating the object expression but before evaluating any of the arguments of the method call. These rules have been designed to conform to the error-reporting logic used by Python.

You will not be tested on program executions that lead to arithmetic integer overflow or out-of-memory. You will also not be tested on programs that use the predefined function `input`, which has not currently been implemented by the reference compiler.

7.6 README

Before submitting your completed assignment, you must edit the README.md and provide the following information: (1) names of the team members who completed the assignment, (2) acknowledgements for any collaboration or outside help received, and (3) how many late hours have been consumed (refer to the course website for grading policy).

8 Implementation Notes

In this assignment, you are provided a significant amount of skeleton code. You are not strictly required to use this code; however, we strongly recommend that you do, since it performs about half of the work required to implement a code generator for ChocoPy. This section describes the design of the skeleton code. The code itself is also heavily documented using Javadoc-style comments.

Although the entry point for this assignment is the static `StudentCodeGen.process()` method, this method does little more than handle input/output. Most of the heavy lifting is done within the `CodeGenImpl` class in the `chocopy.pa3` package, which itself is a sub-class of `CodeGenBase` from the `chocopy.common` package. The `CodeGenImpl` class contains skeletons for emitting RISC-V code corresponding to top level statements and function bodies. You are expected to edit this skeleton and emit code corresponding to all types of program statements and expressions. In doing so, you will most likely want to use inherited fields and methods from the base class, `CodeGenBase`, which you cannot modify (but can override if needed).

8.1 Code generation base

The following tasks have already been performed by `CodeGenBase`:

1. Analysis of the entire program to create descriptors for classes, functions/methods, variables, and attributes. These descriptors, whose class names end with `Info`, are placed in appropriate symbol tables. The symbol tables and `Info` objects are described in Section 8.2. The `globalSymbols` field in `CodeGenBase` references the global symbol table. Every `FuncInfo` object references its corresponding function's symbol table, which takes into account local definitions, implicitly inherited names, as well as explicit `nonlocal`/`global` declarations. You likely do not need to modify the symbol tables in this assignment.

2. Code generation for prototypes of every class (refer to `chocopy_implementation_guide.pdf` to understand what *prototype* means). The `ClassInfo` objects contain labels pointing to their corresponding prototypes in memory.

3. Code generation for method dispatch tables for every class. The `ClassInfo` objects contain labels pointing to their corresponding dispatch tables in memory.

4. Code generation for global variables. For every global variable in the program, there exists exactly one `GlobalVarInfo` object in the global symbol table (these may be inherited by a function's symbol table). A `GlobalVarInfo` object contains a label pointing to the global variable allocated in memory. Global variables are emitted in the data segment using their initially defined values from the ChocoPy program.

5. Management of and code generation for constants. The `constants` field in `CodeGenBase` references a manager for constant integers, booleans, and strings encountered in the program. The method `constants.getIntConstant(int x)` method returns a label that points to a globally-allocated ChocoPy `int` object having the same value as the Java integer `x`. Similar methods are available for booleans and strings. The constants' manager performs caching, so that every distinct constant label references a unique constant. Once code is emitted for all program statements, the `CodeGenBase` emits all encountered constants to the global data segment.

6. Code generation for predefined functions and built-in routines. The `CodeGenBase` class emits bodies of predefined functions such as `len`, `print`, and `object.__init__`, as well as built-in routines such as `abort` and `alloc`. Although you do not need to modify this logic, you may want to read through the code that emits these functions/routines in order to get some inspiration for how to emit code in your own `CodeGenImpl` for user-defined functions.

7. Initialization of the heap and clean exit. The `CodeGenBase` class emits some start-up code that should execute before the first top-level statement is executed. The start-up code includes logic for initializing the heap and setting the initial value of `fp`. The `CodeGenBase` class also emits some tear-down code that should execute after the last top-level statement has been executed. The tear-down code performs a successful exit from the execution environment. The code that you will emit in the method `CodeGenImpl.emitTopLevel()` will be placed in-between the start-up and tear-down logic.

To summarize, the `CodeGenBase` takes care of populating symbol tables, emitting everything that needs to be emitted to the global data segment, as well as emitting boilerplate code to the text segment. Your task in this assignment is to leverage the symbol tables and other available utilities to emit code in the text segment by filling in the `CodeGenImpl`.

Although you probably do not need to do so, it is possible to override virtually every single task that `CodeGenBase` performs, since all of its fields and methods are defined with `protected` or `public` access.

8.2 Symbol table

A symbol table maps identifiers to their corresponding symbol descriptors. This mapping changes depending on the current scope. The starter code creates the following types of symbol descriptors in its analysis (you likely do not need to add to this hierarchy):

- `FuncInfo`: A descriptor for functions and methods. A function has an associated *depth*: global functions and methods have a depth of 0, whereas nested functions that are defined within a function of depth d have a depth of $d + 1$. A `FuncInfo` object contains the function's depth, its symbol table, its parameter list (a list of names), its local variables (a list of `StackVarInfo` objects), a label corresponding to its entry point, and a reference to the `FuncInfo` of its enclosing function (if applicable). The `FuncInfo` class also contains a utility method, `getVarIndex()`, to retrieve the index of a parameter or local variable in order of definition.

- `ClassInfo`: A descriptor for classes. A `ClassInfo` object corresponding to a class contains its type tag, its attributes (a list of `AttrInfo` objects), its methods (a list of `FuncInfo` objects), a label corresponding to its prototype and a label corresponding to its dispatch table. This class also contains utility methods to get the index of an attribute or method in order of their original definitions.

- `GlobalVarInfo`: A descriptor for a global variable. A `GlobalVarInfo` object simply contains the label of its corresponding global variable.

- `AttrVarInfo`: A descriptor for class attributes. A `AttrVarInfo` object contains the initial value of its corresponding attribute, represented as a label that points to a constant allocated in the data segment; the label may be `null` in case of an initial value of `None`.

- `StackVarInfo`: A descriptor for variables allocated on the stack, such as parameters and local variables. A `StackVarInfo` object contains the initial value of its corresponding variable, represented as a label that points to a constant allocated in the data segment; the label may be `null` in case of an initial value of `None`. A `StackVarInfo` object also references the `FuncInfo` object corresponding to the function which defines the stack variable; this pointer is useful for determining the static depth of a stack-allocated variable, which may be necessary when emitting code for accessing non-local variables.

8.3 RISC-V backend

The class `RiscVBackend` contains a large number of methods for emitting RISC-V assembly instructions to an output stream. The field `backend` defined within `CodeGenBase` references the backend whose output stream will be returned by the static method `StudentCodeGen.process()` as the assembly program produced by your ChocoPy compiler. The methods within `RiscVBackend` usually take the form of `emitXYZ`, where `XYZ` is a RISC-V instruction in uppercase. These methods are strongly typed: the arguments to these methods are expected to be objects of type `Register` (an enum defined within `RiscVBackend`), type `Label` (for addresses), or type `Integer` (for immediates). Each such method also expects a comment string as the last argument. For example, to generate the RISC-V instruction `lw a0, 4(fp)`, you might execute the following Java code in `CodeGenImpl`:

```
backend.emitLW(A0, FP, 4, "Load something");
```

Similarly, to invoke a function whose descriptor is available in a variable say `funcInfo`, you might execute the following Java code in `CodeGenImpl`:

```
backend.emitJAL(funcInfo.getCodeLabel(), "Invoke function");
```

8.4 Labels

The class `Label` is heavily used throughout the provided code framework to represent labels in the generated assembly. A `Label` object simply encapsulates the name of a label as a string. Several instruction-emitting methods of the `RiscVBackend` expect a `Label` as an argument.

Labels can be created in two ways: either by directly instantiating a new `Label` object with a specific string provided as an argument to its constructor, or by invoking the utility method `generateLocalLabel()` defined in `CodeGenBase`. The utility method generates a fresh label named `label_<n>`, where `<n>` is a unique integer. This method is quite useful when generating labels for use in local control structures such as conditional branches or loops. The method `RiscVBackend.emitLocalLabel(Label)` is typically used to emit such a label to assembly. By convention, the code generated for a given function should not contain jumps to a local label in a different function. On the other hand, the method `RiscVBackend.emitGlobalLabel(Label)` is used to emit labels which are meant to be referenced across function boundaries; this method also creates a global symbol for the emitted label using the `.globl` assembly directive. Global labels are used for function entry, global variables, constants, object prototypes, dispatch tables, and built-in routines. Almost all of the global labels that you will need to refer to have already been created by `CodeGenBase`.

You should only jump to global labels using unconditional jumps such as `jr` or `jal`. If you want to conditionally branch to a global label (e.g. with `beqz`), then first conditionally branch to a local label, and then jump from there to the target global label. This is because in RISC-V, conditional branch instructions require some bits to encode the registers to test; therefore, the jump target cannot be very far (the offset has to fit within 12 bits). Unconditional jump instructions can jump to targets that are further away.

8.5 Anticipated FAQ

This section answers some common questions that we anticipate may arise when working with the skeleton code.

Where can I find the label corresponding to entity X? Labels for built-in routines are present in fields of `CodeGenBase`. For example, the field `allocLabel` points to the label for the built-in routine `alloc`. Labels for class prototypes and dispatch tables are contained in the corresponding `ClassInfo` objects. Labels for function entry are contained in the corresponding `FuncInfo` objects. Labels for global variables are contained in the corresponding `GlobalInfo` objects.

How do I get a `ClassInfo`/`FuncInfo` object corresponding to X? The `CodeGenBase` has fields that reference `ClassInfo` objects corresponding to predefined classes. For example, the field `objectClass` references the class descriptor for class `object`, the field `intClass` references the descriptor for `int`, and so on. Similar fields are present for predefined functions, such as `printFunc` and `lenFunc`. In general, you can query the current symbol table to retrieve the descriptor for a class or a function that is currently in scope. One exception is the `ClassInfo` object for lists.

The field `listClass` in `CodeGenBase` references a pseudo-class descriptor for lists, which is useful for getting a label that points to the prototype empty list object. There is no real `list` class in ChocoPy, and therefore there is no entry in any symbol table that references this descriptor.

How do I emit instruction XYZ? There isn't an emitXYZ() defined in `RiscVBackend`. There are two ways to handle this. First, you could call the `emitInsn()` method, which emits a raw instruction given as a string. This allows you to emit virtually any line of code to assembly, but it is not strongly typed. Alternatively, you can create a custom strongly typed `emitXYZ` method for an instruction `XYZ` by sub-classing `RiscVBackend` in the `chocopy.pa3` package. Add the required method in the sub-class and then use an instance of this custom sub-class in `StudentCodeGen` instead.

How do I add functionality to one of the Info classes (e.g. `FuncInfo`)? If you feel the need to modify any of the Info classes, simply create sub-classes in the `chocopy.pa3` package. Let's say you create a subclass `MyFuncInfo extends FuncInfo` with some custom methods. Now, override the factory method `makeFuncInfo`, which is originally defined in `CodeGenBase`, in your `CodeGenImpl` class. In this factory method, you can create instances of `MyFuncInfo` instead and the symbol table will now contain instances of this sub-class throughout the program. There is one factory method corresponding to every type of Info class whose instances are inserted into the symbol table. That said, you probably do not need to do this at all.

8.6 Recommendations

This assignment can get quite tricky if you are not comfortable with assembly code. We strongly recommend that you emit useful comments with your assembly code, so that you know what your code is doing when you have to debug it. The Venus Web UI can be a useful tool for interactive debugging.

You may also want to decide on a strategy in terms of which language features to implement first, and in what order to proceed from there on. We recommend trying to tackle code generation for function calls from the get go: this will enable you to actually invoke `print` and observe output. Other easy features to implement include global variables, function prologues and epilogues, local variables, and basic arithmetic. Code generation for `if-else` and `while` loops is also straightforward. Of medium difficulty would probably be code generation for object attribute access, method dispatch, nested functions (including nonlocal variable access), list instantiation and list-element access. The hardest features to implement would likely be string/list concatenation and `for` loops—make sure to allocate sufficient time to tackle these once you are comfortable with the basics.

9 Checkpoint

Part way through the assignment, on **29 April 2019**, we will have a checkpoint to evaluate your progress. At that time, we want your code generation to work correctly on a small set of ChocoPy language features. In particular, we want your code to work on programs with only functions, variable access (global + local + nonlocal), integer and boolean operations, and simple control-structures: `if-else` and `while`. For the checkpoint, you are not required to have implemented

object instantiation, attribute access, method dispatch, or operations on lists and strings. Appendix A lists the tests which you are expected to pass by the checkpoint. The same tests will also be included as part of the final submission, where you are expected to generate code for the full ChocoPy language. The checkpoint accounts for a substantial part of your overall grade for this assignment, so do not take it lightly. Section 10.1 describes how to submit your checkpoint. Section 11.1 describes the grading rubric. For the checkpoint, you do not need to create custom tests in the `student_contributed` directory and do not need to write anything in the README apart from your names. There are no slip hours available for the checkpoint.

10 Submission

10.1 Checkpoint

Submitting your checkpoint requires the following steps:

- Run `mvn clean` to rid your directory of any unnecessary files.
- Add and commit all your progress and push changes to the repository. Run `git commit` followed by `git push origin` to achieve this.
- Tag the desired commit with `pa3checkpoint`. If the desired commit is the latest one, run `git tag pa3checkpoint`. Otherwise, run `git tag pa3checkpoint <commit-id>` where `<commit-id>` is the commit you want to tag as your final submission.
- Push the tag using `git push origin pa3checkpoint`.

10.2 Final submission

Submitting your final assignment requires the following steps:

- Run `mvn clean` to rid your directory of any unnecessary files.
- Add and commit all your progress and push changes to the repository. Run `git commit` followed by `git push origin` to achieve this.
- Tag the desired commit with `pa3final`. If the desired commit is the latest one, run `git tag pa3final`. Otherwise, run `git tag pa3final <commit-id>` where `<commit-id>` is the commit you want to tag as your final submission.
- Push the tag using `git push origin pa3final`.

11 Grading

The project as a whole is worth 25 points. Of these, 5 points will come from the checkpoint and 20 from the final submission.

11.1 Checkpoint

We count the number of successful simple tests from the filesrc/test/pa3/data/samples and benchmarks from the `src/test/pa3/data/benchmarks` directory, which exercise only the features required for checkpoints. Each benchmark program tests a combination of a *subset* of ChocoPy features to perform a non-trivial task. No hidden tests will be used for the checkpoint. Appendix A lists the tests that we will run.

11.2 Final submission

- We count the number of successful sample, benchmark, and hidden tests, including the original checkpoint tests. The maximum score is 15 points.
- 2 points for test `src/test/data/pa3/student_contributed/good.py`, which should cover a range of ChocoPy features such as arithmetic, objects, lists, and string operations. Only exercise the semantics that your implementation handles!
- 3 points code cleanliness and structure. These include Clear naming for variables and other symbols, consistent spacing and punctuation conventions, reasonable modularization of functions and other components, code comments explaining non-obvious logic.

11.3 Extra credit: Bug reports

The reference implementation possibly contains some bugs. If you find a bug, report it by making a post on Piazza with a sample input program and describe how the expected output should differ. The first student/team to report a bug gets extra credit (5 points per unique bug with a maximum of 20 extra credits per team).

Bugs in the reference implementation are defined as (1) unexpected exceptions being reported or (2) violations of the specifications of the assignment or the specifications of the ChocoPy manual, which would lead to incorrect results. Minor mistakes in the ChocoPy manual or this document itself are not considered bugs in the reference implementation, though we would appreciate any such feedback.

The decision on whether to accept a bug report as valid and distinct from previous bug reports is at the discretion of the instructors.

A Checkpoint Tests

At the checkpoint, your code generator will be evaluated on the following 23 tests in `src/test/data/pa3`:

```
sample/literal_bool.py          sample/op_cmp_int.py
sample/literal_int.py           sample/op_div_mod.py
sample/literal_str.py           sample/op_logical.py
sample/id_global.py             sample/op_mul.py
sample/id_local.py              sample/op_negate.py
sample/var_assign.py            sample/op_sub.py
sample/call.py                  sample/stmt_if.py
sample/call_with_args.py        sample/stmt_while.py
```

```
sample/nested.py
sample/nested2.py
sample/op_add.py
sample/op_cmp_bool.py
sample/stmt_return_early.py
benchmarks/exp.py
benchmarks/prime.py
```