Testing Commands and Notes

java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=..s \

--run --dir src/test/data/pa3/sample –test

Stand up from chair every 20-40 min

You will not be tested on program executions that lead to arithmetic integer overflow or out-

of-memory

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 the function’s activation record.

• 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 in the object layout or the index

of a method in the dispatch table.

• GlobalVarInfo: A descriptor for a global variable. A GlobalVarInfo object simply contains

the label of its corresponding global variable.

• AttrInfo: A descriptor for class attributes. An AttrInfo 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, rep-

resented 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 instruc-

tions 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 as-

sembly 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");

**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 generat-

ing 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.

**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

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

**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.

**To run the web-based IDE: open a terminal, cd to the web directory of the assignment package,**

**and run the following command:**

**python -m WebCompiler 8000**

Now, go to a Web browser and navigate to http://localhost:8000 (replace 8000 if you used

a different port). You should see a web page with a code editor, and options to select your own

compiler stages (i.e., the student’s version) or the reference compiler’s stages. Enter a ChocoPy

program and click the button to compile to RISC-V. If the program is valid, then a new window

will open up with the compiled RISC-V code pre-filled within the Venus Web-based simulator. In

case of static errors, this window will close and you will be taken back to the code editor; hover

your mouse over the red cross in the left margin to read the corresponding error message. The

source locations associated with compiler errors in the JSON are used to highlight the errornous

fragments of code in the editor.

For the Web IDE to work, you must have the web server up and running, and you must have

built the JARs using mvn clean package.

java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=..s --run src/test/data/pa3/sample/literal\_int.py.ast.typed --test

java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=..s --run src/test/data/pa3/sample/literal\_int.py.ast.typed --test

Total:

1/80 passing

Passing tests:

src/test/data/pa3/sample/pass.py.ast.typed

src/test/data/pa3/sample/literal\_int.py.ast.typed

src/test/data/pa3/sample/literal\_bool.py.ast.typed

src/test/data/pa3/sample/literal\_str.py.ast.typed

Failed tests:

src/test/data/pa3/sample/stmt\_while.py.ast.typed

src/test/data/pa3/sample/op\_div\_mod.py.ast.typed

src/test/data/pa3/sample/object\_attr\_get\_none.py.ast.typed

src/test/data/pa3/sample/call.py.ast.typed

src/test/data/pa3/sample/id\_local.py.ast.typed

src/test/data/pa3/sample/list\_get\_element.py.ast.typed

src/test/data/pa3/sample/list\_set\_element\_none.py.ast.typed

src/test/data/pa3/sample/object\_method\_none.py.ast.typed

src/test/data/pa3/sample/nested.py.ast.typed

src/test/data/pa3/sample/str\_cat.py.ast.typed

src/test/data/pa3/sample/list\_set\_element\_oob\_1.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list.py.ast.typed

src/test/data/pa3/sample/object\_attr\_set\_none.py.ast.typed

src/test/data/pa3/sample/error\_invalid\_print.py.ast.typed

src/test/data/pa3/sample/id\_global.py.ast.typed

src/test/data/pa3/sample/len\_invalid\_1.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_str\_eval.py.ast.typed

src/test/data/pa3/sample/op\_cmp\_int.py.ast.typed

src/test/data/pa3/sample/str\_cat\_2.py.ast.typed

src/test/data/pa3/sample/op\_sub.py.ast.typed

src/test/data/pa3/sample/object\_method\_nested.py.ast.typed

src/test/data/pa3/sample/list\_len\_empty.py.ast.typed

src/test/data/pa3/sample/str\_get\_element\_oob\_1.py.ast.typed

src/test/data/pa3/sample/op\_negate.py.ast.typed

src/test/data/pa3/sample/list\_get\_element\_oob\_1.py.ast.typed

src/test/data/pa3/sample/var\_assign.py.ast.typed

src/test/data/pa3/sample/object\_attr\_get.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_nested.py.ast.typed

src/test/data/pa3/sample/list\_get\_element\_complex.py.ast.typed

src/test/data/pa3/sample/stmt\_return\_early.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_nested\_same\_var.py.ast.typed

src/test/data/pa3/sample/op\_logical.py.ast.typed

src/test/data/pa3/sample/expr\_if.py.ast.typed

src/test/data/pa3/sample/error\_mod\_zero.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_str\_same\_var.py.ast.typed

src/test/data/pa3/sample/str\_cmp.py.ast.typed

src/test/data/pa3/sample/op\_mul.py.ast.typed

src/test/data/pa3/sample/list\_concat\_2.py.ast.typed

src/test/data/pa3/sample/list\_get\_element\_oob\_2.py.ast.typed

src/test/data/pa3/sample/list\_len.py.ast.typed

src/test/data/pa3/sample/input.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_str.py.ast.typed

src/test/data/pa3/sample/str\_get\_element\_oob\_2.py.ast.typed

src/test/data/pa3/sample/object\_method.py.ast.typed

src/test/data/pa3/sample/call\_with\_args.py.ast.typed

src/test/data/pa3/sample/nested2.py.ast.typed

src/test/data/pa3/sample/str\_get\_element\_oob\_3.py.ast.typed

src/test/data/pa3/sample/op\_add.py.ast.typed

src/test/data/pa3/sample/op\_cmp\_bool.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_return.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_str\_empty.py.ast.typed

src/test/data/pa3/sample/object\_attr\_set.py.ast.typed

src/test/data/pa3/sample/list\_concat\_none.py.ast.typed

src/test/data/pa3/sample/predef\_constructors.py.ast.typed

src/test/data/pa3/sample/list\_get\_element\_oob\_3.py.ast.typed

src/test/data/pa3/sample/stmt\_if.py.ast.typed

src/test/data/pa3/sample/list\_set\_element.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_modify.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_nonlocal.py.ast.typed

src/test/data/pa3/sample/object\_attr\_set\_eval\_order.py.ast.typed

src/test/data/pa3/sample/error\_div\_zero.py.ast.typed

src/test/data/pa3/sample/op\_is.py.ast.typed

src/test/data/pa3/sample/list\_set\_element\_oob\_3.py.ast.typed

src/test/data/pa3/sample/object\_init.py.ast.typed

src/test/data/pa3/sample/list\_concat.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_eval.py.ast.typed

src/test/data/pa3/sample/len\_invalid\_2.py.ast.typed

src/test/data/pa3/sample/object\_method\_override.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_none.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_list\_empty.py.ast.typed

src/test/data/pa3/sample/str\_get\_element.py.ast.typed

src/test/data/pa3/sample/stmt\_for\_str\_nested.py.ast.typed

src/test/data/pa3/sample/object\_method\_complex\_call.py.ast.typed

src/test/data/pa3/sample/list\_set\_element\_oob\_2.py.ast.typed

src/test/data/pa3/sample/str\_len.py.ast.typed

src/test/data/pa3/sample/list\_get\_element\_none.py.ast.typed