Compiler Construction
CSCI-GA.2130-001/Spring 2021
Programming Assignment 3: Code Generation

This assignment is due by 11:59 pm ET, May 2, 2022.

Contents

1 Acknowledgments

This programming project was adapted, with the original authors' permission, from the project developed at the University of California, Berkeley for CS 164: Programming Languages and Compilers.

ChocoPy was designed by Rohan Padhye and Koushik Sen, with substantial contributions from Paul Hilfinger:

Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In Proceedings of the 2019 ACM SIGPLAN SPLASH-E Symposium (SPLASH-E '19), October 25, 2019, Athens, Greece. ACM, New York, NY, USA, 5 pages. `https://doi.org/10.1145/3358711.3361627`

2 Overview

The three programming assignments in this course will direct you in developing 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) RISC-V code generation.

In this assignment, we implement code generation for ChocoPy. This phase of the compiler takes as input the type-annotated AST of a semantically valid and well-typed ChocoPy program, and outputs the corresponding RISC-V assembly code.

3 Tools and Resources

In addition to the tools and resources introduced in assignments 1 and 2, we will use:

- RISC-V specification. See Brightspace.
- Venus, a RISC-V simulator:
  - Original version: `https://github.com/kvakil/venus`.
  - Venus 164: `https://github.com/chocopy/venus`.

4 Workflow

4.1 Complete Pipeline

All compiler phases are in place at this point, so we can outline the entire process:

1. The parsing pass takes a ChocoPy source file and produces an AST:

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

2. The semantic analysis pass takes an AST and produces a type-annotated AST:

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

3. The code generation pass takes a type-annotated AST and produces a corresponding RISC-V assembly program:

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

4. The bundled Venus 164 simulator takes a RISC-V assembly program and executes it:

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

As usual, replacing `r` with `s` will make that pass use your rather than reference code. Omitting the `--out` option will print the results to standard output.

4.2 Submitting

To submit your team's assignment:

- Pick a commit to represent your final submission.
- Confirm that a clean clone at that commit compiles and runs as expected.
- Git tag the commit with `pa3final`.
- Push your changes.
- Push your tag using `git push origin pa3final`.

4.3 Grading

The grading rubric allocates 100 points as follows:

- 70 points for the standard tests (includes tests not in the starter code).
- 10 points for the benchmark tests.
- 10 points for the extensions and optimizations.
- 10 points for the report.

5 Project Structure

As before, you are free to add, edit, and remove anything in the starter code, but be aware that changing some code may stop the project from working correctly; use version control and run the tests often.

The list below highlights the most relevant files and directories, while section 8 gives more details.

- `src/`: Source files, some of which you may need to modify.
  - Do not modify classes in `chocopy.common` directly, as they are used in both your assignment and the reference implementation. If you want to modify these classes, duplicate or extend them in a different package.
  - `main/java/chocopy/pa3/StudentCodeGen.java`: The entry point for the code generator. `String process(Program program, boolean debug)`, takes the annotated AST produced by the semantic analyzer and returns the corresponding RISC-V assembly program.
  - `main/java/chocopy/common/codegen/CodeGenBase.java`: An abstract class providing the infrastructure for code generation. Do not edit this class directly; subclass it if you would like to make changes.
  - `main/java/chocopy/pa3/CodeGenImpl.java`: Skeleton implementation of the abstract `CodeGenBase`. Modify this file to emit assembly code for top-level statements and function bodies.
  - `main/java/chocopy/common/codegen/*.java`: Support classes for code generation. See section 8 for more details.
  - `main/asm/chocopy/common/*.s`: Assembly-language implementations of built-in functions, which `CodeGenBase` copies into the output program. You can use the same technique for adding runtime support routines (such as string concatenation): put them into the `codegen/asm` directory and see `emitStdFunc` in `CodeGenBase`.
  - `test/data/pa3/`: Test ChocoPy programs:
    - `sample/*.py`: Sample test programs.
    - `sample/*.py.out.typed`: Annotated ASTs corresponding to the test programs. Inputs to the 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.
    - `benchmarks/*.py`: Non-trivial benchmark programs, meant to test the overall working of your compiler. These tests are included in grading.
    - `benchmarks/*.py.out.typed`: Annotated ASTs for benchmarks.
    - `benchmarks/*.py.out.typed.s.result`: The results of executing the benchmarks.

6 Execution Environment

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

6.1 Venus

In order to execute RISC-V code in a platform-independent manner, we use a version of the Venus simulator, originally developed by Keyhan Vakil: `https://github.com/kvakil/venus`. 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. See `https://github.com/kvakil/venus/wiki` for documentation.

This assignment uses a modification of Venus created at the University of California, Berkeley for *CS 164 Programming Languages and Compilers* and referred to as Venus 164: `https://github.com/chocopy/venus`. The modifications mainly aim to improve conformance with the GNU RISC-V toolchain:

- `.word` directive: Support for emitting addresses in the data segment using the syntax `.word <label>`.
- `.align` directive: Limited support for specifying byte alignment in the data segment. `.align <n>` inserts zero-valued bytes as padding such that the next available address is a multiple of 2^n.
- `.string` directive: Support for emitting ASCII strings using the syntax `.string <string in quotes>`.
- `.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 to include a value in an assembler instruction before figuring out precisely what that value will be.
- The original Venus supports non-standard pseudo-instructions, such as `seq` and `sgt`. Venus 164 enforces strict mode: the only supported pseudo-instructions are those listed in the RISC-V specification.

6.2 Venus Tooling

Venus 164 is automatically added to your project on build (see `pom.xml`). This is how your RISC-V programs are executed when you supply the `--run` option.

Venus 164 is also available online: `https://chocopy.org/venus.html`.

Finally, two untested but potentially useful resources:

- Venus web frontend: `https://cs61c.org/sp22/resources/venus-reference/`

- VS Code extension: `https://marketplace.visualstudio.com/items?itemName=hm.riscv-venus`

7 Assignment Specification

The objective of this assignment is to build a code generator takes as input the type-annotated AST of a semantically valid and well-typed ChocoPy program, and outputs the corresponding RISC-V assembly code.

7.1 Input and Output

The interface to your code generator is the static method `StudentCodeGen#process`. The method accepts a typed AST in JSON format, corresponding to a semantically valid and well-typed ChocoPy program. The typed AST will have the same format as the output of the previous assignment. The field `inferredType` will be non-null for every expression in the AST that evaluates to a value.

The expected output is a RISC-V assembly program to be executed in the Venus 164 environment. **The assembly program your compiler generates need not match the program generated by the reference compiler.** Instead, your goal is to independently produce assembly code that implements the source ChocoPy program and follows the operational semantics given in the language reference.

7.2 Validation

Testing is done by executing the generated RISC-V program in Venus 164 and comparing the output streams of your program and that generated by the reference compiler. The program is expected to follow the operational semantics defined in the ChocoPy language reference. The output should contain a sequence of lines, where the i-th line corresponds to the string representation of the `str`, `int`, or `bool` object provided as argument to the i-th dynamic invocation of the predefined `print` function.

7.3 Memory Management

All compiled ChocoPy programs will have 32MB of memory to work with. (The standard tests require far less.)

The register gp will point to the beginning of the heap before the first top-level statement is executed. The reference compiler does not implement garbage collection (GC); newly allocated objects block space for the entire remaining duration of the program. You are not expected to implement GC either, though the heap and object layouts are designed to facilitate GC.

7.4 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 compiler are described in the implementation guide. 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, there are built-in routines emitted in the method `CodeGenImpl#emitCustomCode`. Your generated programs can simply jump to one of these existing 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.

Your compiler will not be tested on program executions that lead to arithmetic integer overflow or out-of-memory.

7.5 Extensions

As part of this project, you need to design and implement your own extensions to ChocoPy. They can be inspired by Python or another language, or invented completely by you. You need to introduce either two "medium" or one "complex" extension. Refer to assignment 1 for examples.

In this assignment, you need to implement code generation for your extensions. You are not locked into your initial choices and can change your mind at any point, as long as your implementation remains consistent across the entire compiler.

Showcase your extensions in `src/test/data/pa3/extra/ext.py`, preferably in a variety of situations. Your new features should be backward compatible and not break any of the existing tests.

7.6 Optimizations

Required

As part of this assignment, you need to design and implement at least two different optimizations. Consult chapters 8 and 9 in the textbook for ideas.

You can choose to implement straightforward optimizations that do not require specialized algorithms or data structures. For example:

- Constant folding (for example, replace `2 * 3 * 5` with `30`).
- Algebraic identities (for example, replace `x * 1` with `x`).
- Peephole optimizations (for example, erase redundant `sw t0, mem` after `lw t0, mem`).

If you choose to implement the more advanced optimizations, such as constant propagation or dead code elimination, consult the textbook for the implementation details. (You may find *Engineering a Compiler* by Keith Cooper and Linda Torczon a useful complement.)

Showcase your optimizations in `src/test/data/pa3/extra/opt.py`. Your optimizations should not change the semantic meaning of programs and should not break any of the existing tests.

Optional

You are welcome to implement more that two optimizations. We will evaluate the performance of your compiler (that is, of the code it generates) on five non-trivial programs provided in the `benchmarks` directory.

For simplicity and reproducibility, we will measure performance as *the total number of RISC-V instructions executed* rather than time it takes to execute them. You can measure this value by adding the `--profile` option immediately after `--run`:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy \
--pass=rrr --run --profile ./src/test/data/pa3/sample/op_add.py

Reading ./src/test/data/pa3/sample/op_add.py
101
Cycles executed = 91
```

We will recognize the top three compilers with a shout-out and extra credit.

7.7 Report

As part of your submission, create `PA3-REPORT.md` with at least the following sections (feel free to add more). As before, link to `PA1-REPORT.md` or `PA2-REPORT.md` where appropriate rather than repeat the same material.

- `Acknowledgments`: Provide attribution to any collaborations, external resources, or outside help.
- `Organization`: Describe how you organized work and handled software engineering aspects, such as distributing work, doing code reviews, adding more tests, using continuous integration, and so on.
- `Design`: Describe any design decisions you made or problems you solved. Include the approaches that did not work, if any. Cover at least the following:
  - How did you handle boxing?

  - How did you handle stack allocation?
  - How did you handle nested functions?

- `Extensions`: Describe your language extensions. Concentrate on the challenges of code generation. If you changed you extensions since PA2, note that and explain why.
- `Optimizations`: Describe your optimizations. Explain what makes them semantically correct (that is, why they don't change the meaning of the program) and how you implemented them.

We will grade the report based on clarity, structure, and technical insight. Treat it as a white paper about your code generator.

8 Implementation Notes

As before, you are given skeleton code to build upon, and, as before, you are free to ignore it and start from scratch. If you choose to use the starter code, it is heavily documented, and this section provides a high-level overview.

Most of the heavy lifting is done within the `CodeGenImpl` class, a subclass of `CodeGenBase`. The `CodeGenImpl` class contains templates for emitting RISC-V code corresponding to top-level statements and function bodies. Edit this template to 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 should not modify but can override if needed.

8.1 Code Generation Base

`CodeGenBase` already performs the following tasks:

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 the next section. 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 the implementation guide to understand prototypes). 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)` 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`, `input`, 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 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, `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 `CodeGenImpl`.

Although you probably do not need to do so, it is possible to override almost every task that `CodeGenBase` performs, since all of its members are protected or public.

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, 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` class 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 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 Miscellaneous

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 entries 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? `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 is no `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 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 subclassing `RiscVBackend` in the `chocopy.pa3` package. Add the required method in the subclass and then use an instance of this custom subclass in `StudentCodeGen` instead.

How do I add functionality to one of the `Info` classes, such as `FuncInfo`? First, you probably do not need to do this. If you still have the need to modify any of the `Info` classes, create subclasses in the `chocopy.pa3` package. For example, you can create class `MyFuncInfo` that extends `FuncInfo` with some custom methods. Then, override the factory method `makeFuncInfo`, 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 subclass throughout the program. There is one factory method corresponding to every type of `Info` class whose instances are inserted into the symbol table.

Why does the reference compiler emit code for X in this way? The reference compiler performs several optimizations, including:

- Using `fp`-relative indexing of temporaries instead of moving `sp` around all the time.
- Unboxing `int` and `bool` values unless they are used as an object.
- Aligning the stack to 64-bit boundaries at call sites.

We cannot help you reverse-engineer the code generated by the reference compiler, nor should you have the need to do so. The ChocoPy reference manual and implementation guide are sufficient for this assignment.

8.6 Recommendations

Learn RISC-V assembly. This assignment can be tricky if you are not comfortable with assembly code. Writing RISC-V assembly programs by hand is a good way to practice.

Generate comments. Emit comments with your assembly code. Knowing what the generated code does and why it was generated that way will help in debugging.

Decide what to implement first. This is a large assignment; have a plan for tackling different features. Here is one possible approach:

- Start by generating code for function calls. This will enable you to actually invoke `print` and observe output.

- Global variables, function prologues and epilogues, local variables, and basic arithmetic are straightforward to implement. Tackling them next would give you the necessary practice and confidence.
- `if-else` and `while` loops are also straightforward and can be implemented next.
- The following features are likely to require more effort: object attribute access, method dispatch, nested functions (including non-local variable access), list instantiation and list-element access.
- String/list concatenation and `for` loops are likely to be the hardest to implement. Approach them last.

This is the end of the assignment.