CS164 Programming Languages and Compilers **Spring 2024**

Programming Assignment 4

Assigned: Apr 15, 2024 **Due:** Apr 26, 2024 at 11:59pm

1 Overview

The four 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, (3) code generation, and (4) optimization.

For this assignment, you will be extending your compile from PA3 to emit more efficient RISC-V assembly. Specifically, we are interested in applying optimizations that reduce the total number of instructions executed. We will focus on a subset of ChocoPy for this project, consisting of the constructs supported in the PA3 Checkpoint along with list operations.

2 Getting started

For this assignment you will be using your **existing** PA3 repository. We recommend creating a branch for your work on PA4:

```
git checkout -b pa4
```

3 Dependencies, files and directories, execution environment

Please see the PA3 specification for details. Note that you will **need to modify README.md** as part of this project to describe your optimization techniques.

4 Assignment goals

The objective of this assignment is to implement optimizations in your ChocoPy compiler to improve the performance of the generated RISC-V code. For simplicity and reproducibility, we will measure performance not in terms of how much *time* it takes for your generated code to execute a program, but instead we will count the *total number of RISC-V instructions* executed. To measure this value, simply add the `--profile` argument immediately after `--run`. For example:

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

You can of course provide `--pass=rrr` to measure the performance of the reference compiler.

For this assignment, we will focus on a **subset of ChocoPy** to avoid cascading failures if you were unable to achieve full correctness in PA3. Specifically, we will only test your compiler on programs that use the constructs from the PA3 **Checkpoint** as well as some simple list operations

(if you were unable to complete the checkpoint, see Section 4.5 for an alternative assignment). Specifically, programs are restricted to the following AST node types:

- Program
- FuncDef
- VarDef
- TypedVar
- ClassType
- **ListType**
- ExprStmt
- ReturnStmt
- AssignStmt
- IfStmt
- WhileStmt
- Identifier
- BinaryExpr (+ **only on lists**, not strings)
- UnaryExpr
- CallExpr
- **IndexExpr** (on lists **and** strings)
- **ListExpr**
- NoneLiteral
- StringLiteral
- IntegerLiteral
- BooleanLiteral

4.1 List Operations

Note that you will need to implement a few constructs that are not required on the checkpoint:

- `len` on strings and lists
- `IndexExpr` to look up elements in strings and lists
- `ListExpr` nodes for constructing lists
- `BinaryExpr` + nodes for concatenating lists

The latter two of these are particularly challenging. If you have been unable to implement these yourself, you are allowed to use subroutines from the reference compiler for **this project only, not PA3**. To add these subroutines, modify the `emitCustomCode` function in your compiler and add the following lines:

```
emitStdFunc(noconvLabel, "chocopy/reference/");
emitStdFunc(concatLabel, "chocopy/reference/");
emitStdFunc(consListLabel, "chocopy/reference/");
```

And override the `initAsmConstants` to add:

```
@Override
protected void initAsmConstants() {
    super.initAsmConstants();
    backend.defineSym("listHeaderWords", 4);
}
```

Then, add the label definitions at the top of your class:

```
/** Label for identity conversion. */
private final Label noconvLabel = new Label("noconv");
/** Label for list concatenation routine. */
private final Label concatLabel = new Label("concat");
/** Label for list construction  routine. */
private final Label consListLabel = new Label("conslist");
```

Finally, you'll need to add some custom code for loading the reference subroutines:

```
@Override
protected String getStandardLibraryCode(String name, String lib) {
    String orig = super.getStandardLibraryCode(name, lib);
    if (orig != null) {
        return orig;
    }

    String os = chocopy.common.Utils.getResourceFileAsString(
        lib + name.replace("$", "") + ".os"
    );
    if (os == null) {
        return null;
    }

    StringBuilder result = new StringBuilder();
    for (int p = 0; p < os.length(); p += 1) {
        result.append((char) ((os.charAt(p) ^ ((p + 1) % 127 + 1)) - 128));
    }
    return result.toString();
}
```

With these subroutines available, you can implement the last two using the standard calling conventions. For list expressions, push the elements in order to the stack with the length pushed last (at the lowest address), and invoke `consListLabel`. The return value is the list.

For list concatenation, first push the address of `noconv` twice (the address can be loaded by an LA instruction), then push the first list, then push the second list, and invoke `concatLabel`. The return value is the concatenated list.

4.2 Optimizations to Implement

For this project, you are free to implement whichever optimizations you would like, but we have a list of recommended optimizations to get you started:

- Boxing elimination (don't box integers and booleans unless necessary)
- Constant propagation (when there are expressions involving values that are known at compile-time, pre-calculate them)

- Common subexpression elimination (use existing computation results rather than re-computing the same expression later)
- Dead code elimination (if a value is not used, don't compute it)
- Register allocation (instead of pushing temporaries to the stack, use registers)

4.3 Input/output specification

The student-generated benchmarks will be available to everyone working on PA4, and will be posted on Ed shortly after the release of the project. These benchmarks will be available as a flat directory consisting of triples of Python programs, an input, and an output. We will provide you with two out of the three inputs submitted during the Pre-PA4 assignment, and to make testing easier the repository is organized so that each Python program is duplicated for each input.

To run your compiler on all the benchmarks, you can run:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=rrs
--run --test --dir path/to/your/downloaded/benchmarks
```

To get the cycle count for an individual benchmark, you can run:

```
java -cp "chocopy-ref.jar:target/assignment.jar" chocopy.ChocoPy --pass=rrs
--run --profile the_benchmark.py < the_benchmark.py.py.in
```

To compute the overall speedup, we also include a Python script `bench_all.py` which simulates the logic run on the autograder. Within your ChocoPy folder, run:

```
python ../path/to/your/downloaded/benchmarks/bench_all.py
```

4.4 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).

Further, you must answer the following questions in your write-up by editing the README.md file (one or two paragraphs per question is fine):

- Which optimizations did you implement? Cite line numbers in your implementation.
- Which optimization had the most significant impact, why do you think this is the case?

4.5 Fallback Assignment

If your compiler is unable to produce code that emits the correct outputs for each benchmark (i.e. you were unable to complete the PA3 Checkpoint), there is an alternative assignment available to avoid cascading failures. In your README, answer the following questions (we are expecting much longer responses in this case, a few paragraphs for each will be sufficient):

- What bugs in your compiler prevented you from achieving correctness? Provide *minimal* examples of code your compiler is unable to handle for each of the bugs.

- Pick three optimizations from the list earlier, and for each one:
  - Explain in English with pseudocode how you would implement this on top of a working PA3.
  - Take a specific benchmark program from the class submissions, and provide examples of how the optimization would affect the RISC-V (please show both before and after)

Note that if you choose this alternative assignment, course staff reserves the option to ask you additional questions to ensure that you meet the expected understanding required for this project.

5 Submission

Submit your implementation to the PA4 assignment on Gradescope. The autograder will maintain a leaderboard of performance compared to the reference compiler, measured as a **geometric mean** of speedup over all the benchmark programs.

6 Grading (40 points, up to 10 extra credit)

The project as a whole is worth 40 points. Of these, 12 points come from the pre-PA4 benchmark program submission and 28 points come from the final submission. In the final submission, you also have the opportunity to earn up to 10 extra credit points which will be applied to the projects category.

6.1 Pre-PA4 (12 points)

You will **individually** submit four benchmark programs with three input/output examples each, as specified in the pre-PA4 announcement. Each input/output example that matches the reference compiler will give you one point. You will not receive points for a benchmark program if it uses constructs outside of the ChocoPy subset for this project.

6.2 Final submission (28 points, up to 10 extra credit)

As in PA3, submit your compiler implementation to Gradescope. For PA4, your code will be benchmarked and placed on a live leaderboard. Based on your geomean performance compared to the reference compiler, you are eligible for different numbers of points. If your compiler is unable to produce code that emits the correct result for $\geq 80\%$ of the benchmarks, you are only eligible for the following categories:

- Passes $< 30\%$ of tests: **0 points**
- Passes $< 60\%$ of tests: **8 points**
- Passes $[60\%, 80\%)$ of tests: **12 points**

If your compiler produces correct code for $\geq 80\%$ of the benchmarks, your submission will instead be graded by computing the **geometric mean** of the speedup relative to the reference compiler for each benchmark on the **top 80% of speedups**. The categories for this are:

- $< 30\%$ the performance: **16 points**
- $[30\%, 50\%)$: **20 points**
- $[50\%, 70\%)$: **24 points**
- $[70\%, 90\%)$: **26 points**
- $[90\%, 100\%)$: **27 points**
- $\geq 100\%$: **28 points**

On top of this, if your team scores in the top 5 places on the leaderboard, each member of your team will receive **10 points** of extra credit.