CS4120/4121/5120/5121—Spring 2023
Programming Assignment 5
Assembly Code Generation
Due: Tuesday, April 18, 11:59PM

For this programming assignment, you will implement an *assembly-code generator* for the Eta programming language. Assembly code is generated from the intermediate representation, making your compiler fully functional. The assembly code should be processable by the GNU assembler and linkable with the runtime library we provide in order to produce working executables.

0 Changes

- None yet; watch this space.

1 Instructions

1.1 Grading

Solutions will be graded on documentation, completeness, correctness, and style. 5% of the score is allocated to whether bugs in past assignments have been fixed.

1.2 Partners

You will work in a group of 3–4 students for this assignment. This should be the same group as in the last assignment. If not, please discuss with the course staff.

Remember that the course staff is happy to help with problems you run into. For help, read all Ed posts and ask questions (that have not already been addressed), attend office hours, or schedule a meeting with course staff.

1.3 Package names

Please ensure that all Java code you submit is contained within a package whose name contains the NetID of at least one of your group members. Subpackages under this package are allowed; they can be named however you would like.

2 Building on previous programming assignments

Use your lexer from PA1, your parser from PA2, your type checker from PA3, and your IR generator from PA4. Part of your task for this assignment is to fix any problems that you had in the previous assignments. Discuss these problems in your overview document, and explain briefly how you fixed them.

3 Runtime library

We require the code you produce to be able to interface with the runtime we provide, and to interoperate with other functions we may create for testing. For this reason we require you to follow the ABI specification, and in particular to implement System V calling conventions. You've already done most of the work required to meet the ABI spec in PA4. In this assignment, you'll take care of the details that were kept abstract in the IR. In particular, you will need to generate code that respects ABI rules about caller- and callee-saved registers.

4 Quality of assembly code

We do not expect you to implement optimizations or high-quality register allocation for this assignment; the goal here is to produce working programs. It's fine to spill every `TEMP` in a function to the stack. However, we do expect you to implement nontrivial *instruction selection*. Your tiles should make use of x86-64 instruction set features like complicated addressing modes and in-memory operands.

5 Assembling your code

It may help you to take a look at the assembly lab and its corresponding example code to get a better idea of how to assemble and link your generated assembly code.

In particular, in the released runtime/ there is a script linketa.sh that should be useful for assembling your code as follows:

```
./linketa.sh -o binary foo.s
```

Running that command in the VM generates a binary file called `binary` which you can run by running

```
./binary
```

6 Command-line interface

A general form for the command-line interface is as follows:

```
etac [options] <source files>
```

Unless noted below, the expected behaviors of previously available options are as defined in the previous assignment. `etac` should support any reasonable combination of options. For this assignment, the following options are possible:

- `--help`: Print a synopsis of options.
- `--lex`: Generate output from lexical analysis.
- `--parse`: Generate output from syntactic analysis.

- `--typecheck`: Generate output from semantic analysis.
- `--irgen`: Generate intermediate code.
- `--irrun`: Generate and interpret intermediate code.
- `-sourcepath` <path>: Specify where to find input source files.
- `-libpath` <path>: Specify where to find library interface files.
- `-D` <path>: Specify where to place generated diagnostic files.
- `-d` <path>: Specify where to place generated assembly output files.

  For each source file given as `path/to/file.eta` in the command line, an output file named `path/to/file.s` is generated to contain the assembly output of the source file. If `path` is given, the compiler should place generated assembly output files in the directory relative to this path. The default is the current directory in which `etac` is run.

  For example, if this path is `o/u/t` and the file to be generated is `path/to/file.s`, the compiler should place this file at `o/u/t/path/to/file.s`.
- `-O`: Disable optimizations.
- `-target` <OS>: Specify the operating system for which to generate code.

  `OS` may be one of `linux`, `windows`, and `macos`. Your compiler is only required to support the `linux` option. You may support additional operating systems at your discretion, and you may define the default operating system for your compiler in a way that is convenient to you.

7 Build script

Your build script `etac-build` from previous programming assignments should remain available. The expected behaviors of the build script are as defined in the previous assignment. The build script must be in the root directory your submission `zip` file. Problems within the test script from previous submissions should be fixed.

8 Test harness

`eth` has been updated to contain test cases for this assignment and to support testing assembly code generation. While we've added a few code generation tests, you will need to develop your own test cases to properly test your compiler.

You should ensure that you are using the latest version of the Docker virtual machine for this assignment. To update your Docker image, run `docker pull charlessherk/cs4120-vm`. The runtime should already be installed on the VM. Further updates to the runtime can be pulled in by running the `update` script in the `runtime` directory. To update `eth`, run the `update` script in the `eth` directory on the VM.

Note that the runtime will not work properly in the ARM version of the VM, so ARM users will need to assemble and run code in the official course VM.

A general form for the `eth` command-line invocation is as follows:

```
eth [options] <test-script>
```

The following options are of particular interest:

- `-compilerpath <path>`: Specify where to find the compiler
- `-testpath <path>`: Specify where to find the test files
- `-workpath <path>`: Specify the working directory for the compiler

For the full list of currently available options, invoke `eth`.

An `eth` test script specifies a number of test cases to run. Once the updated `eth` is released, directory `eth/tests/pa5` will contain a sample test script (`ethScript`), along with several test cases. `ethScript` also lists the syntax of an `eth` test script.

9 Submission

You should submit these files on CMS:

- `overview.txt/pdf`: Your overview document for the assignment. This file should contain your names, your NetIDs, all known issues you have with your implementation, and the names of anyone you have discussed the homework with. It should also include descriptions of any extensions you implemented. The Overview Document Specification outlines our expectations.
- A `zip` file containing these items:
  - *Source code*: You should include all source code required to compile and run the project. Please ensure that the directory structure of your source files is maintained within the archive so that your code can be compiled upon extraction. If your code depends on any third-party libraries, please include compilation instructions in your overview document.
    Include your parser and lexer generator input files, e.g., `*.cup` and `*.flex`, as well as any generated code.
  - *Tests*: You should include all your test cases and test code that you used to test your program. Be sure to mention where these files are and to describe your testing strategy in your overview document.

  Do not include any non-source files or directories such as `.class`, `.classpath`, `.project`, `.git`, and `.gitignore`.
- `pa5.log`: A dump of your commit log since your last submission from the version control system of your choice.

10 Tips

You should complete your implementation of assembly-code generator as you see fit, but we offer the following suggestions.

First, download and compile the runtime. Read `README.txt`. Take a look at the `.s` files inside the `examples` directory, and try assembling and linking them by hand. If you can do it for the examples, you will be able to do it for your compiler's output.

As part of your implementation, you will be specifying many different tiles and their mapping from the IR to the assembly code. Plan out how you will represent and organize these tiles.

Once your compiler is producing runnable binaries, you can test it by compiling an Eta program to a binary and then checking the output of the binary. But be careful—a bug in instruction selection is hard to uncover using only end-to-end tests. You will need tests that exercise your instruction selection pass by giving it all kinds of valid IR as input.

10.1 Debugging your compiled code with GDB

If your binary is not working correctly, you can debug it using `gdb` to understand its behavior. We talked about how to use `gdb` in the assembly lab (see the link below). You can set the display mode to Intel syntax with the command `set disassembly-flavor intel` (even better, put it in the file `~/.gdbinit` so you don't have to bother in the future). Set a breakpoint in your main function with `break _Imain_paai` and then run the program with `run`. You can execute the program one instruction at a time with the `ni` instruction. The command `x/20i $rip` will display the next 20 instructions in the instruction stream, or you can use the `disas` command to disassemble code from a symbol. You can print the values of all registers with the `info registers` command. The gdb reference manual has many more useful commands.

10.2 Assembler Directives

While looking at compiled code, you might run into instructions that look something like

```
.text
```

here .text is a assembler directive. Assembler directives are commands that are part of the assembler syntax but are not related to the x86 processor instruction set. To distinguish directives from assembly instructions, directive names begin with a period. You will find it helpful to take a look into assembler directives. Running an existing compiler like gcc is a good way to see what directives are needed by the assembler.

Asssembler directives were covered in the assembly lab, so consult the slides from the lab for more information. Here are some of the useful directives:

- `.intel_syntax noprefix`
  Make the assembler use Intel syntax for instructions.
- `.globl` *name*
  Declare a global name that will be visible to other program modules. Functions declared in Eta interfaces should be introduced this way.
- `.file` ⟨*filenum*⟩ ⟨*file*⟩
  This directive allows connecting assembly code to the source code that generated it, allowing you to (optionally) debug Eta code at the source level. The number ⟨*filenum*⟩ is used to name the file in the `.loc` directive. Use of this
- `.loc` ⟨*filenum*⟩ ⟨*linenum*⟩ ⟨*column*⟩
  This directive specifies a precise location of the next instruction in a source file.

11 External links

The following resources may be useful:

- Assembly lab slides
- WikiBook: x86 assembly
- Intel® 64 and IA-32 Architectures Software Developer Manuals
- GNU Assembler manual
- Assembler Directives

Unfortunately, these documents use different assembly syntax: Intel and AT&T syntax respectively. You may use either syntax with your compiler. To use Intel syntax, however, you will need to use the `.intel_syntax` directive.