CS4120/4121/5120/5121—Spring 2026
Programming Assignment 4
Intermediate Code Generation
Due: Friday, March 13, 11:59PM

This programming assignment requires you to implement an *IR generator* for the Eta programming language. As discussed in class, it is difficult to translate high-level source code directly into assembly code. A solution to this problem is to perform simpler translations using *intermediate representations*. Compilers often make use of multiple IRs, but your compiler should only use one.

0 Changes

- 3/12: Fix example for mangled naming of globals
- 3/8: AI policy and submission instruction update

1 Instructions

1.1 Grading

Solutions will be graded on documentation and design, completeness of the implementation, 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. Read all Ed posts and ask questions that have not been addressed, attend office hours, or set up meetings with any course staff member for help.

1.3 Package names

Please ensure that all Java code you submit is contained within a package (or similar, for other languages) 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 like.

1.4 Tips

You should complete your implementation of an IR generator as you see fit, but we offer the following suggestions.

Incrementally implement the translations you will apply to your source nodes to produce the IR nodes. As you write these translations, test them out using simple expressions.

Incremental testing can be helpful. For example, a test for the correctness of a translation of an expression `e` need not test translations of subexpressions of `e`.

2 Design overview document

We expect your group to submit an overview document. The Overview Document Specification outlines our expectations.

3 Building on previous programming assignments

Use your lexer from PA1, your parser from PA2, and your type checker from PA3. 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.

4 Version control

The instructor and TAs have access to your repository and will look at your commit history during grading to ensure that everyone in the group contributed. If your commit history does not clearly reflect individual contributions, briefly mention it in your overview document. (For example, if you employ pair programming for elements of the assignment (not a bad idea!), you may wish to clarify this in your overview document, as only one member would appear on the commit history for that work.)

5 IR model

We expect you to use the tree-based IR presented in class. We generally encourage you to stick with the IR from class. You may add nodes to this IR or adjust the meaning or syntax of current nodes, but if you choose to do so, you should carefully describe and justify your changes in your overview document. Additionally, note that if you make changes to the IR then you must also support `--irrun` flag in a meaningful way (see Section 8). That is, if you provide a modified IR, then you must also provide a way to interpret that IR. We recommend (but do not require) that you use our IR and interpreter as a base as it lowers well, but you are free to extend it or create your own if you wish.

You are required to output canonical (lowered) IR. This means:

- All side effects (including function calls) must be at the top level in their own statements.
- Basic blocks should be reordered so that the "false" target of all conditional jumps is a fall-though to the appropriate label. You should also remove any unnecessary jumps that go to the very next statement, and add jumps as necessary to ensure that the control flow graph is unchanged.

We will interpret your IR code for grading. To avoid grading issues, please use the following calling conventions. All arguments to a function call should be given as children to the **CALL_STMT** node. All returns from a function should be placed in the dedicated return temps, i.e. the first

Content of input file	Content of output file

`use io`
`use conv`
`g : int = 3;`
`main (a : int[][]) {`
`  g = 5;`
`  println("Hello World!")`
`}`
	`(COMPUNIT example`
	`  (DATA _Vg (3))`
	`  (DATA string_const1`
	`  (12 72 101 108 108 111 32 87 111 114 108 100 33))`
	`  (FUNC _Imain_paai`
	`  (SEQ`
	`  (MOVE (TEMP a) (TEMP _ARG1))`
	`  (MOVE (MEM (NAME _Vg)) (CONST 5))`
	`  (MOVE (TEMP t2) (NAME string_const1))`
	`  (MOVE (TEMP t0) (ADD (TEMP t2) (CONST 8)))`
	`  (CALL_STMT 0 (NAME _Iprintln_pai) (TEMP t0))`
	`  (RETURN))))`

Table 1: Example IR Translation

returned value in `_RV1`, the second in `_RV2`, and so on. On the receiving side (the callee function, or the code after the function call), these arguments will be stored in registers with the prefix `_ARG` and indices starting from 1 (e.g. `_ARG1`, `_ARG2`, etc).

Globals are expected to be implemented using the `IRData` node. The IR simulator also has support for ctors (constructors) which provide the ability to run some code to automatically run at the beginning of the program. This feature is not needed for this assignment and we advise you to ignore it. Table 1 shows one example translation which involves global variables.

5.1 Provided code

An initial Java implementation of this IR that you may build on or even use unmodified, along with an interpreter for this IR implementation, is provided in a released `zip` file `pa4-release.zip` on CMS. This release includes some example of correctly structured IR code.

6 Constant folding

For this assignment, you are also required to implement *constant folding* at the IR level. This optimization will improve the performance of your generated code by computing the values of side-effect-free expressions at compile time. You may also implement some constant folding at the AST level.

7 Eta ABI

Although your compiler will not be able to generate runnable code until after the next assignment, you will need to begin ensuring that your code will be compliant with the Eta *application binary interface* (ABI) for run-time support. You will need this run-time support for program bootstrapping, for memory management, and for I/O.

To help you with this, we are providing you with an ABI specification for your reference.

8 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.
  For each source file given as `path/to/file.eta` in the command line, an output file named `path/to/file.ir` will be generated with the intermediate representation of the source file. The generated code should be pretty-printed as S-expressions.
- `-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.
- `-O`: Disable optimizations.
  If specified, optimizations such as constant folding should not be performed.

You are also required to provide the following option as part of your compiler.

- `--irrun`: Generate and interpret intermediate code. We encourage but do not require you to provide a non-trivial implementation of `--irrun` for your compiler. It will be a big help to you when debugging code generation, and it should be straightforward since we have already provided you a Java implementation of an IR interpreter.
  For each source file, the intermediate code is generated, as with the `--irgen` option. The compiler then interprets this expression, ideally

  ```
  CALL(NAME(_Imain_paai), CONST(0))
  ```

  Per the ABI specification, this statement invokes the procedure `main()` with a dummy argument. Any call to the standard library functions should be simulated by the interpreter. For example, the interpreter should handle a call to `print()` by simply printing output to `System.out`.
  You have two options for implementing this compiler option. In the case that you stick with the class IR representation, you do not need to actually provide functionality to interpret your IR, but the flag should return exit code 2 to indicate that this flag is not fully supported by your compiler. If you are using your own IR representation, this flag must interpret your IR and execute the code. In this case, the compiler must not return exit code 2. We recommend implementing such functionality by using a modified version of the IR interpreter that we have provided, either as a library or as a standalone program run as a separate process.

If you are building your compiler in Java, you will get almost all of the implementation of the `--irrun` option in the `IRSimulator` class that is part of the released code. If you are building your compiler in another language, it should not be difficult to start the Java IR interpreter as a separate process, to port the IR interpreter to that language, or to build one from scratch.

9 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 of your submission `zip` file.

If your `etac-build` uses Gradle, it is now required that you do not use the Gradle daemon (at least when invoked via `etac-build`). This can be done by passing the CLI argument `--no-daemon` to `gradle`.

10 Test harness

`eth` has been updated to contain test cases for this assignment and to support testing IR generation. To update `eth`, run the `update` script in the `eth` directory on the 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`.

The best way to run `eth` with the provided test cases is from the home directory of the VM, using this command:

```
eth -compilerpath <etacpath> -testpath <tp> -workpath <wp> <ethScript>
```

where

- `<etacpath>` is the path to the directory containing your build script and command-line interface.
- `<tp>` is of the form `eth/tests/pa#/`, where # is the programming assignment number.
- `<wp>` is preferably a fresh, nonexistent compiler such as `shared/ethout`.
- `<ethScript>` is of the form `eth/tests/pa#/ethScript`, where # is the programming assignment number.

An `eth` test script specifies a number of test cases to run. Once the updated `eth` is released, directory `eth/tests/pa4` will contain a sample test script (`ethScript`), along with several test cases. `ethScript` also lists the syntax of an `eth` test script.

`eth` was used successfully in the last iteration of the course, but bugs are always possible. Please report errors, request additional features, and give feedback on Ed.

11 Submission

Before submitting, you should first prepare your repository. Make sure that your repository is complete and ready to be graded. In particular, it should include all of the following:

- All source code needed to compile and run your project, including any required third-party libraries.
- All test cases and any scripts used to run them. Please also include the EthScript you use for the testing harness.
- A design overview document, as described in Section **??**, placed in the root of your repository.
- A `README` file explaining how to navigate, build, run, and test your code (this may overlap with the design overview).
- No unnecessary files, such as large binaries, generated files, IDE metadata, or OS metadata.

Once your repository is ready, go to your repository on the Cornell GitHub website and create a new release (found in the right sidebar). Use the programming assignment title—e.g., `Programming Assignment 3: Implementing Semantic Analysis` as the title of your release, and set the tag to `submissions`. Publish your release and download the release ZIP file. Finally, submit the release ZIP file on CMSX before the deadline of Friday, March 13, 11:59PM. **All of these steps must be completed on time.**

Your repository should be self-contained, meaning that graders should be able to clone it and build your project without additional setup. Using Git submodules is allowed; if you do so, ensure that `git clone --recursive your_repo` retrieves everything needed to build and run your project.

For smaller libraries, it is often easy and effective to include the source code directly, but be sure to make clear what is library code, e.g. by package name. JAR files also work well, but sometimes do not include Javadoc and are less well integrated with your IDE. Your mileage may vary.

If you use a lexer or parser generator, please include the input files, e.g., `*.flex`. Your `etac-build` should use these files to generate source code, and you should not submit the corresponding generated source code file (e.g. `.java`).

Finally, remember that clear and well-written documentation—including commit messages, the design overview, and the `README`—does not just help your graders; they also help you and your partners.

12 AI Policy

You may use AI tools such as ChatGPT to assist with small, well-scoped pieces of code. However, you should not use AI tools to generate large portions of your code or core functionality. Using AI too much is a common pitfall and tends to backfire in this course.

In your design overview document, please write down where and how you used AI tools, if at

all.

The Vibe Mirage.

Who needs to write code anymore—don't we just ask generative AI to do the work for us? This approach may sound like it will save time and effort, but it rarely does. AI-generated code often looks convincing until it is tested carefully and reveals subtle bugs that cost far more time than they save. Because each stage of the project builds on your existing code, your foundations must be solid. Relying on AI can undermine your understanding of your own code, and that understanding is critical for success.