CS4120/4121/5120/5121—Spring 2026
Programming Assignment 3
Implementing Semantic Analysis

Due: Friday, February 27, 11:59PM

This programming assignment requires you to implement a *type checker* for the Eta programming language. Given an AST of a syntactically valid Eta program, the type checker will see if it makes sense according to the static semantics of the language. If there are any type errors, it should produce descriptive error messages. If there are not, it should annotate the AST with information computed during its work, and also construct symbol tables describing all variables. We have provided a formalization of the Eta type system to help you get started.

0 Changes

- None yet; watch this space.

1 Instructions

1.1 Grading

Solutions will be graded on design, correctness, and style. A good design makes the implementation easy to understand and maximizes code sharing. A correct program compiles without errors or warnings, and behaves according to the requirements given here. A program with good style is clear, concise, and easy to read. Anything ambiguous about the spec will be graded more leniently if you mention all your considerations and confusions in the design overview document.

A few suggestions regarding good style may be helpful. You should use brief but mnemonic variable names and proper indentation. Keep your code within an 80-character width. Methods should be accompanied by Javadoc-compliant specifications, and class invariants should be documented. Other comments may be included to explain nonobvious implementation details. Other languages may follow a certain convention or style guide, such as CS 3110's OCaml style guide or the Google C++ style guide.

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.

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 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.

1.4 Tips

The typing rules for Eta contain most of the information you need to implement this assignment. The type information you compute will be useful in code generation, not just for catching errors. Similarly, symbol tables are useful when allocating memory for local variables. Therefore, try to preserve information you will need in later phases.

As the project goes on, it will be increasingly important that your project group is functioning effectively as a team. Everyone should be contributing significantly. If this is not happening, your group should talk about how to be more effective.

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 and your parser from PA2. 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 Type checker

Each source file should be checked for lexical, syntactic, and semantic errors. Your compiler should behave as follows:

- If there is a lexical, syntax, or semantic error within the source, the compiler should indicate this by printing to standard output (`System.out`) an error message that includes the kind (lexical, syntax, or semantic) and the position of the error, in the following format:

  ```
  <kind> error beginning at <filename>:<line>:<column>: <description>
  ```

  where <`kind`> is one of `Lexical`, `Syntax`, and `Semantic`.
- If the program is semantically valid, the compiler should terminate normally (exit code `0`) without generating any standard output, unless certain options are specified on the command line. (See Section 7 for details.)

6 Interface files

In previous assignments, the compiler only needed to read the specified input file(s). To type-check a source file, however, it will be necessary in general to read interface files specified with the `use` statement. Any functions declared in these interface files may be used in the source file, with the signatures declared in the interface file.

7 Command-line interface

A command-line interface is the primary channel for users to interact with your compiler. As your compiler matures, your command-line interface will support a growing number of possible options.

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.

  For each source file given as `path/to/file.eta` in the command line, an output file named `path/to/file.typed` is generated to provide the result of type checking the source file. The compiler should be able to handle both relative and absolute paths for all filename arguments.

  If the source file has a parse error or is a semantically invalid Eta program, the content of the `.typed` file should contain only the following line:

  ```
  <line>:<column> error:<description>
  ```

  where <`line`> and <`column`> indicate the beginning position of the error, and <`description`> details the error.

  If the source file is a semantically valid Eta program, the content of the `.typed` file should contain only the following line:

```
Valid Eta Program
```

Table 1 shows a few examples of expected results.

- `-sourcepath <path>`: Specify where to find input source files.
- `-libpath <path>`: Specify where to find library interface files.

  If given, the compiler should find library interface files in the directory relative to this path. The default is the current directory in which `etac` is run.
- `-D <path>`: Specify where to place generated diagnostic files.

8 Build script

Your compiler implementation should provide a build script called `etac-build` in the compiler path that can be run on the command-line interface. The build script must be in the root directory your submission `zip` file. This script should compile your implementation and produce files required to run `etac` properly. Your build script should terminate with exit code `0` if your implementation successfully compiles, or 1 otherwise.

Please try to avoid downloading third-party libraries from the internet when building your compiler. Either include these with your submission, or request an installation on the virtual machine.

The test harness will assume the availability of your build script and fail grading if the build script fails to build your compiler.

9 Test harness

`eth` has been updated to contain test cases for this assignment and to support testing semantic analysis.

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`.

An `eth` test script specifies a number of test cases to run. Once the updated `eth` is released, directory `eth/tests/pa3` will contain a sample test script (`ethScript`), along with several test cases. `ethScript` also lists the syntax of an `eth` test script.

If any errors occur in `eth` or you wish to request additional features, please reach out to the course staff on Ed.

Content of input file	Content of output file

`use io`
 	
`main(args: int[][]) {`
`  print("Hello, Worl\x{64}!\n")`
`  c3po: int = 'x' + 47;`
`  r2d2: int = c3po   //  No Han Solo`
`}`
	`Valid Eta Program`

`foo(): bool, int {`
`  expr: int = 1 - 2 * 3 * -4 *`
`  5pred: bool = true & true | false;`
`  if (expr <= 47) { }`
`  else pred = !pred`
`  if (pred) { expr = 59 }`
`  return pred, expr;`
`}`
 	
`bar() {`
`  _, i: int = foo()`
`  b: int[i][]`
`  b[0] = {1, 0}`
`}`
	`Valid Eta Program`

`valid(): int[] {`
`  return "Valid Eta Program";`
`}`
	`Valid Eta Program`

`foo(x: int): bool {`
`  b:bool = x + 47`
`  return b`
`}`
	`2:12 error:Cannot assign int to bool`

`foo(x: int): bool {`
`  return 47 + (((false & (((x))))))`
`}`
	`2:29 error:Operands of & must be bool`

`foo(): bool { return baz() }`	`1:22 error:Name baz cannot be resolved`

`foo(a: bool[]) {`
`}`
`bar() {`
`  foo({25 + 47})`
`}`
	`4:7 error:Expected bool[], but found int[]`

`foo(): bool {`
`  x:int = 2`
`  b:bool = x != 3`
`}`
	`1:13 error:Missing return`

`foo(): int, int, int { return 0, 1, 2 }`
`bar() { x:int, _ = foo() }`
	`2:9 error:Mismatched number of values`

`foo(): int, int, int { return 0, 1, 2 }`
`bar() { x:int, b:bool, _ = foo() }`
	`2:16 error:Expected int, but found bool`

`foo() { }`
`bar() { x:int = foo() }`
	`2:17 error:foo is not a function`

Table 1: Examples of running `etac` with `--typecheck` option

10 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 2, 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, February 27, 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.

11 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.