CS4120/4121/5120/5121—Spring 2026
Programming Assignment 2
Implementing Syntactic Analysis
Due: Friday, February 13th, 11:59PM

This programming assignment requires you to implement a *parser* for the Eta programming language. This includes devising a grammar to describe the language's syntax. The end result will be a program that reads a Eta source file and produces a pretty-printed version of the AST representing the program.

0 Changes

- Update version control requirements.
- Add AI policy.
- Update submission instructions.
- Update policies for third-party libraries in section 7.
- Update CUP/Jflex conflicts in section 5.4.

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.

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.

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 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 and strongly encouraged. They can be named however you would like.

1.4 Tips

The key to success on the project is for all group members to contribute effectively. Working with partners, however, may add challenges. Some tips:

- Meet with your teammates as early as possible to work out the design and to discuss the responsibilities for the assignment. Keep meeting and talking as the project progresses. Be prepared for your meetings. Be ready to present proposals to your partners for what to do, and to explain the work you have done. Good communication is essential.
- A key design step for this assignment is the design of the AST. A good design will also yield benefits in later assignments, so be sure to get your group agreeing on the AST early.
- A good way to partition an assignment into parts that can be worked on separately is to agree on, first, what the different modules will be, and further, exactly what their interfaces are, including detailed specifications. The key is to find the right interfaces that effectively cleave the assignment into largely independent work.
  For this assignment, the abstract syntax tree (AST) offers a natural way to divide the work. Getting a good AST design from the start will allow your team to work in parallel on parsing to produce the AST, and on pretty-printing the AST even without a working parser. You may be tempted to work on pretty-printing only after parsing is complete. That sequential strategy will likely reduce the effectiveness of your group.
- Drop by office hours and explain your design to a member of the course staff as early as possible. This will help you avoid big design errors that will cost you as you try to implement.
- This project is a great opportunity to try out *pair programming*, in which you program in a pilot/copilot mode. It can be more fun and tends to result in fewer bugs. A key ingredient is to have the pilot/typist convince the other person that the code meets the predefined spec. It might be tempting to let the pilot/typist be the person who is more confident on how to implement the code, but you will probably be more successful if you do the reverse.
- This project is also a great time for *code reviews* with your group members. Walk through your code and explain to your partners what you have done, and convince your partners your design is good. Be ready to give and to accept constructive criticism!
- Sometimes people feel that they are working much harder than their partners. Remember that when you go to implement something, it tends to take about twice as long as you thought it would. So what your partners are doing is also twice as hard as it looks. If you think you are working twice as hard as your partners, you two are probably about even!

2 Design overview document

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

3 Building on PA1

Use your lexer from PA1. Part of your task for this assignment is to fix any problems that you had in PA1.

If we discovered a problem with your lexer, you must devise one or more test cases that *clearly* expose the bug. After you have done this and confirmed that your PA1 implementation indeed fails these tests, fix the bug. Discuss these tests in your overview document, and explain briefly what the problem was.

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 Parser

The job of your parser is to parse an Eta source file. Note that Eta source file may be either a program file (extension `.eta`) or an interface file (extension `.eti`). Your parser should require the appropriate syntax for the kind of file it is parsing. It should output `.parsed` for both `.eta` and `.eti` files. *Hint:* if your lexer provides the right input to the parser, you can get away with having just one grammar for the whole language.

Your parser must be implemented using an LALR(1) parser generator, such as CUP for Java. If you are using some language other than Java, consult the course staff for the appropriate parser generator to use.

Your compiler should behave as follows:

- If there is a lexical or syntax error within the source, the compiler should indicate this by printing to standard output (`System.out`) an error message that includes the position of the error.
- If the program is syntactically 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 6 for details.)

5.1 Provided code

Code and libraries that might help with your implementation are provided in a released `zip` file.

- The `CodeWriterSExpPrinter` class supports pretty-printing of an S-expression. This output will help you debug your parser.
- In addition, we are providing you with a stub CUP specification (`eta.cup`) from which to start.

```
Warning : *** Shift/Reduce conflict found in state #8
  between reduction on stmt ::= IF expr THEN stmt •
  and shift on         stmt ::= IF expr THEN stmt • ELSE stmt
  under symbol ELSE
  Ambiguity detected for nonterminal stmt
  Example: IF expr THEN IF expr THEN stmt • ELSE stmt
  Derivation using reduction:
    stmt ::= [IF expr THEN stmt ::= [IF expr THEN stmt •] ELSE stmt]
  Derivation using shift    :
    stmt ::= [IF expr THEN stmt ::= [IF expr THEN stmt • ELSE stmt]]
```

Figure 1: A sample error message reported by the CUP extension. The first four lines are in the standard version of CUP.

5.2 A version of CUP that generates counterexamples

Debugging conflicts reported by a parser generator can be challenging, especially when the parser generator only reports conflict items and lookahead symbol. In Spring 2016, the course staff implemented an extended version of CUP [1] that looks for counterexamples to better explain parsing conflicts. Figure 1 shows an error message reported by our implementation for the dangling-else shift/reduce conflict.

We are providing you with this extension of CUP (`java_cup.jar` in the released `zip` file) to help you with diagnosing potential conflicts in your grammar. In a presence of conflicts, counterexamples are constructed by default. To turn off counterexample generation, pass the flag `-noexamples`. For the full list of options, invoke `cup --help` from your command line.

If you are programming in Java, we strongly suggest you employ this version of CUP in your project; it has been helpful to students in previous years. If you are programming in C or C++, bison version 3.7 and later has a similar feature (added by a Cornell MEng student). In OCaml, the Menhir parser generator also generates useful though less explanatory counterexamples.

5.3 A note on JFlex and CUP

The authors of JFlex have provided good support for interfacing with CUP. You can modify your JFlex specification to generate a lexer that your CUP-generated parser is able to understand without an adapter. This likely requires some minor changes, but the heart of your lexer will be the same.

5.4 Potential Version Mismatch

Your parser might fail in the VM because of a version mismatch. The old "charlessherk/cs4120-vm" VM has JFlex 1.6.1 installed, which is compatible to CUP runtime up to 11a. The counterexample implementation uses CUP 11b. If you use "new" 11b features, it is likely that JFlex and CUP generate mismatching Java code.

[1] See [Isradisaikul and Myers 2015] for more information.

There are two ways to resolve this. First, you may use the new VM that runs the newest CUP and JFlex. Alternatively, you can download the latest JFlex and use that in the old VM (make sure your build script points to this JFlex instead of the old one from the VM!).

6 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.
  For each source file given as `path/to/file.eta` in the command line, an output file named `path/to/file.lexed` is generated to provide the result of lexing the source file.
- `--parse`: Generate output from syntactic analysis.
  For each source file given as `path/to/file.eta` in the command line, an output file named `path/to/file.parsed` is generated to provide the result of parsing the source file.
  If the source file is a syntactically invalid Eta program, the content of the `.parsed` 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 syntactically valid Eta program, the content of the `.parsed` file should contain an S-expression visualization of the AST representing the program.
  Recall the syntax of *symbolic expressions* (S-expressions):

$$S \quad ::= \texttt{(}\ L^*\ \texttt{)} \mid \langle x \rangle \mid \varepsilon$$

  In the grammar above, $\langle x \rangle$ is an *atom* and L is a *list*. For Eta, possible atoms are as follows:

  - keywords, operators, and identifiers
  - types `int` and `bool`
  - integer and boolean literal constants
  - character literal constants, enclosed in single quotes
  - string literal constants, enclosed in double quotes

  S-expression lists represent all other syntactic constructs. The general syntax is as follows:

$$
\begin{aligned}
\textit{program} &::= \texttt{((}\textit{use}^*\texttt{)}\ \texttt{(}\textit{definition}^*\texttt{))} \\
\textit{use} &::= \texttt{(use}\ \langle \textit{id} \rangle\texttt{)} \\
\textit{definition} &::= \textit{method} \mid \textit{globdecl} \\
\textit{method} &::= \texttt{(}\langle \textit{id} \rangle\ \texttt{(}\textit{decl}^*\texttt{)}\ \texttt{(}\textit{type}^*\texttt{)}\ \textit{block}\texttt{)} \\
\textit{globdecl} &::= \texttt{(:global}\ \langle \textit{id} \rangle\ \textit{type}\texttt{)} \mid \texttt{(:global}\ \langle \textit{id} \rangle\ \textit{type}\ \langle \textit{value} \rangle\texttt{)} \\
\textit{decl} &::= \texttt{(}\langle \textit{id} \rangle\ \textit{type}\texttt{)} \\
\textit{block} &::= \texttt{(}\textit{stmt}^*\texttt{)} \\
\textit{op} &::= \texttt{(}\langle \textit{op} \rangle\ \textit{arg}^*\texttt{)}
\end{aligned}
$$

For Eta interface (`eti`) files, the syntax is very similar, with `id`, `decl`, `type` the same as above:

$$
\begin{aligned}
\textit{interface} &::= \texttt{((}\textit{method\_interface}^*\texttt{))} \\
\textit{method\_interface} &::= \texttt{(}\langle \textit{id} \rangle\ \texttt{(}\textit{decl}^*\texttt{)}\ \texttt{(}\textit{type}^*\texttt{))}
\end{aligned}
$$

Newline characters and additional spaces may be inserted between tokens for readability.

Table 1 shows a few examples of expected results.

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

  If given, the compiler should find given input source 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 `p` and the given source file is `a/r/se.eta`, the compiler should find this file at `p/a/r/se.eta`. When determining the output path for generated diagnostic files, the value of `-sourcepath` should be ignored. For example, the parser diagnostic file for the previous example should be placed at `a/r/se.parsed`.

- `-D <path>`: Specify where to place generated diagnostic files.

  If given, the compiler should place generated diagnostic files, e.g., via `--lex` or `--parse` option, in the directory relative to this path. The default is the current directory in which `etac` is run.

  For example, if this path is `p` and the file to be generated is `a/r/se.lexed`, the compiler should place this file at `p/a/r/se.lexed`.

7 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. 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.

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`
`}`
	`( ((use io))`
	`  ( (main ((args ([] ([] int)))) ()`
	`  ( (print "Hello, World!\n")`
	`  (= (c3po int) (+ 'x' 47))`
	`  (= (r2d2 int) c3po)`
	`  )`
	`  )`
	`  )`
	`)`

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

`+-----------------------------------+`
`| What a beautiful, invalid program! |`
`+-----------------------------------+`
	`1:1 error:Unexpected token +`

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

If the compiler uses large third-party libraries, it is recommended that you include precompiled versions of these libraries. If the size of compiler still exeeeds the limit for submission, you may include a build script that fetches libraries online.

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

8 Test harness

A general form for `eth` command-line interface 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. Directory `eth/tests/pa2` contains 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, or give feedback on Ed.

9 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.
- A design overview document, as described in §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 2: Parser` 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 13th, 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.

10 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.