CS4120/4121/5120/5121—Spring 2019
Programming Assignment 2
Implementing Syntactic Analysis
Due: Thursday, February 13, 11:59PM

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

0 Changes

- 2/10: Clarified behavior of `-sourcepath` when generating diagnostic files.

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 Piazza 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.
- One 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.
- 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

As in the last assignment, you must submit file `pa2.log` that lists the commit history from your group since your last submission.

5 Parser

The job of your parser is to parse a Xi source file. Note that Xi source file may be either a program file (extension `.xi`) or an interface file (extension `.ixi`). Your parser should require the appropriate syntax for the kind of file it is parsing. It should output `.parsed` for both `.xi` and `.ixi` 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 (`xi.cup`) from which to start.

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.

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

```
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 original to CUP.

We strongly suggest you employ this version of CUP in your projects; it was used quite successfully in the previous iteration of the course.

If you choose to use this version of CUP, please indicate and briefly discuss your experience in your overview document. We would like to hear whether counterexamples help you debug your grammar!

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.

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:

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

Unless noted below, the expected behaviors of previously available options are as defined in the previous assignment. `xic` 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.xi` 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.xi` 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 Xi 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 Xi 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 Xi, 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:

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

For Xi interface (`ixi`) 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 `xic` is run.

  For example, if this path is `p` and the given source file is `a/r/se.xi`, the compiler should find this file at `p/a/r/se.xi`. 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`.

<table>
<tr><th>Content of input file</th><th>Content of output file</th></tr>
<tr><td><pre>
use io

main(args: int[][]) {
 print("Hello, Worl\x64!\n")
 c3po: int = 'x' + 47;
 r2d2: int = c3po   // No Han Solo
}
</pre></td><td><pre>
(((use io))
 ((main ((args ([] ([] int)))) ()
   ((print "Hello, World!\n")
    (= (c3po int) (+ 'x' 47))
    (= (r2d2 int) c3po)
)
)
)
)
</pre></td></tr>
<tr><td><pre>
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}
}
</pre></td><td><pre>
(()
 ((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))
)
)
)
)
</pre></td></tr>
<tr><td><pre>
+-----------------------------------+
| What a beautiful, invalid program! |
+-----------------------------------+
</pre></td><td><pre>
1:1 error:Unexpected token +
</pre></td></tr>
</table>

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

- `-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 `xic` 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 `xic-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 `xic` properly. Your build script should terminate with exit code `0` if your implementation successfully compiles, or `1` otherwise.

Please refrain from 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.

8 Test harness

`xth` (`xic` *t*est *h*arness) has been updated to contain test cases for this assignment and to support testing syntactic analysis. To update `xth`, run the `update` script in the `xth` directory on the VM.

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

```
xth [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 `xth`.

An `xth` test script specifies a number of test cases to run. Directory `xth/tests/pa2` contains a sample test script (`xthScript`), along with several test cases. `xthScript` also lists the syntax of an `xth` test script.

`xth` 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 Piazza.

9 Submission

Important: Your submission must work on a clean copy of the VM, without making global environment changes.

You should submit these items 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.
- A `zip` file containing these items:
  - *Source code*: You should include everything required to compile and run the project. *We require that* `xic` *and* `xic-build` *are at the root of the* `zip` *file.*

    If you use a lexer generator, please include the lexer input file, e.g., `*.flex`. Please include your parser generator input file, e.g., `*.cup`.

    Your `xic-build` should use these files to generate source code, and you should not submit the corresponding generated source code files (e.g. `*.java`). Do not submit compiled versions of your own code (submitting precompiled libraries is OK).
  - *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 in your overview document. Do not submit instructor tests or xth.
  - *Libraries*: Your build process must not download anything from the internet. If your code depends on any third-party libraries, they must be included in the submission.

    Include precompiled libraries (e.g. JAR files) when feasible, especially for large libraries. For smaller libraries, it often makes sense to include the source code directly, but be sure to make clear what is library code, e.g. by package name.

    Do not make global environment changes in your xic-build script.

  Do not include any derived, IDE, or SCM-related files or directories such as `.class`, `.jar` `.classpath`, `.project`, `.git`, and `.gitignore`, unless they are precompiled versions of libraries. Pay particular attention to the `.git` folder - this makes your submission needlessly large.

  It is strongly encouraged that you use the `zip` CLI tool on a *nix platform, such as the course VM. Also, it is suggested that you write a small (shell) script to pack your submission zip file, since you will be using it repeatedly throughout the course.

  Your `zip` command should look something like:

  ```
  zip -r submission.zip xic xic-build src tests ...
  ```

  Do not use Archive Utility or Finder on macOS as they include extraneous dotfiles, and do not use a Windows tool which does not maintain the executable bit of your `xic` and `xic-build`.

  We reserve the right to deduct points for submissions not meeting these requirements.
- `pa2.log`: A dump of your commit log since your last submission from the version control system of your choice.