CS4120/4121/5120/5121—Spring 2020
Programming Assignment 3
Implementing Semantic Analysis
Due: Friday, February 28, 11:59PM

This programming assignment requires you to implement a *type checker* for the Xi programming language. Given an AST of a syntactically valid Xi 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 Xi 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.

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 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 Xi 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

As in the last assignment, you must submit file `pa3.log` that lists the commit history from your group since your last submission.

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:

```
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.
- `--parse`: Generate output from syntactic analysis.
- `--typecheck`: Generate output from semantic analysis.

  For each source file given as `path/to/file.xi` in the command line, an output file named `path/to/file.typed` is generated to provide the result of type checking the source file.

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

  ```
  Valid Xi 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 `xic` is run.
- `-D <path>`: Specify where to place generated diagnostic files.

Content of input file	Content of output file

`use io`
 	
`main(args: int[][]) {`
`  print("Hello, Worl\x64!\n")`
`  c3po: int = 'x' + 47;`
`  r2d2: int = c3po   //  No Han Solo`
`}`
	`Valid Xi 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 Xi Program`

`valid(): int[] {`
`  return "Valid Xi Program";`
`}`
	`Valid Xi 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() { _ = 2 }`	`1:13 error:Expected function call`
`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 `xic` with `--typecheck` option

8 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. 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 `xic` 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

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

To update `xth`, run the `update` script in the `xth` directory on the VM.

A general form for the `xth` command-line invocation 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`.

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

```
xth -compilerpath <xicpath> -testpath <tp> -workpath <wp> <xthScript>
```

where

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

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

Despite additional testing, `xth` is still in the development phase. Many features have been added since the last release. Please report errors, request additional features, or give feedback on Piazza.

10 Submission

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 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.
    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 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`.
- `pa3.log`: A dump of your commit log since your last submission from the version control system of your choice.