

# CS4120/4121/5120/5121—Spring 2016

## Programming Assignment 4

### Intermediate Code Generation

Due: Friday March 18, 11:59PM

This programming assignment requires you to implement an *IR generator* for the [Xi programming language](#). As discussed in class, it is difficult to translate from high-level source directly into assembly code. A solution to this problem is to perform simpler translations through *intermediate representations*. Often, compilers make use of multiple IRs, but your compiler should only use one.

## 0 Changes

- 3/16: Updated xth.
- 3/15: Clarified the basic block reordering requirements.
- 3/9: Updated the released IR implementation.

## 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. For help, read all Piazza posts and ask questions (that have not already been addressed), attend office hours, or meet with any course staff member either at the prearranged office hour time or at a mutually satisfactory time you arrange.

### 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

You should complete your implementation of 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 development and testing are helpful. For example, a test for the correctness of a translation of an expression *e* need not (and should not) test for the correctness of the 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

As in the last assignment, you must submit file `pa4.log` that lists the commit history from your group since your last submission.

## 5 Changes from previous programming assignments

Filenames are now required as part of error messages. In particular, when 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 at <filename>:<line>:<column>: <description>
```

where `<kind>` is one of `Lexical`, `Syntax`, and `Semantic`.

Error reporting for interface files is similar.

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 9 for details.)

## 6 IR model

We expect you to use the tree-based IR presented in 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. Note that we will be able to support you best if you use the IR presented in class.

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

- All side effects (including function calls) must be at the top level as statements.
- Basic blocks should be reordered so that the “false” target of all conditional jumps is a fall-through 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.

## 6.1 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](#).

Since the intermediate representation is machine-dependent, all arguments to a function call can be passed via temporary registers. The name of these registers have prefix `_ARG` and indices starting from 0. Also, due to the possibility of multiple results from function calls, the return values of a `CALL` instruction are stored in temporary registers with prefix `_RET` and indices starting from 0. These registers are shared across function calls, so any code wishing to use the values of these registers must save them before evaluating a `CALL` instruction.

The provided interpreter assumes the calling convention above. Please ensure that your translation uses this convention to avoid problems with grading.

## 7 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.

## 8 Xi 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 Xi *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.

## 9 Command-line interface

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.
- `--irgen`: Generate intermediate code.

For each source file given as `path/to/file.xi` in the command line, an output file named `path/to/file.ir` is generated to contain 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.

We encourage but do not require you to provide the following option as part of your compiler. It will be a big help to you when debugging code generation.

- `--irrun`: Generate and interpret intermediate code.

For each source file, the intermediate code is generated, as with the `--irgen` option. The compiler then uses the provided IR interpreter to simulate execution of this expression:

```
CALL(NAME(_Imain_pai), 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`.

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 port the IR interpreter to that language or to build one from scratch.

## 10 Build script

Your build script `xic-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 your submission zip file. Problems within the test script from previous submissions should be fixed.

## 11 Test harness

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

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/pa4` 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.

## 12 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. Include your parser generator input file, e.g., `*.cup`, as well as the generated code. If you use a lexer generator, please include the lexer input file, e.g., `*.flex`, as well as the generated code.

- *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`.

- `pa4.log`: A dump of your commit log since your last submission from the version control system of your choice.