Appendix A
The Compiler Project

There are several general approaches for a compiler project in an academic course on compiler construction. One approach is to give detailed explanations and a complete implementation for a compiler for one source language but then concentrate the project around writing a compiler for a different source language. Another approach is to give explanations and a **partial** implementation for a compiler for a particular source language and then to concentrate the project around finishing the incomplete work and possibly extending the source language or targeting a different computer architecture. This book uses the latter approach.

The overall project of developing a compiler for CPRL is divided into 11 smaller projects (numbered 0-10) as described below. For most of the projects, the book GitHub repository has lots of both complete and skeletal code to help you get started plus CPRL test programs that can be used to check your work. For each project you should test your compiler with both correct and incorrect CPRL programs as explained in the project descriptions.

Organizational Structure of the Compiler Project

The compiler project is structured into four separate organizational modules, with explicit dependencies among the modules.

- The compiler module is independent of the other three.
- The CVM module depends only on the compiler module.
- The assembler module depends on both the compiler and CVM modules.
- The CPRL module depends on both the compiler and CVM modules, but it has no dependency on the assembler module.

These four organizational modules correspond to four Java modules as implemented in Java version 9 or later, and the Java modules are defined in four `module-info.java` files to make the dependencies explicit. Note that you are not required to use Java modules for the compiler project. Traditional `CLASSPATH` dependencies or IDE-specific dependencies among the organizational units will work just fine, but you will need to remove the four `module-info.java` files if you are not using Java modules.

For the Eclipse IDE, the four modules can be structured as four projects within a single workspace. Similarly, for IntelliJ IDEA, the four modules can be structured as four modules within a single IDEA project. While it is possible to place all packages in a single module, I recommend the four-module structure since it separates the architecture in a way that makes the dependencies transparent. The GitHub repository contains handouts that can assist you in creating the four modules using either Eclipse or IntelliJ IDEA. If you are using an IDE other than Eclipse or IntelliJ IDEA, there should be a similar way to organize the compiler project into different organizational units within that IDE.

Note that package names and module names follow recommended practices.

- Names use a reverse Internet domain-name convention.
- Each module name corresponds to the name of its principal exported package.

Following is a description of the four modules for the compiler project. Since we are programming in Kotlin, all four module definitions require `kotlin.stdlib` in addition to other module requirements.

1. **The Compiler Module**
   Module `edu.citadel.compiler` (a.k.a. the compiler module) contains classes that are not directly tied to the CPRL programming language and therefore are useful on any compiler-related project. Examples include classes such as `Position` and `Source` defined in package `edu.citadel.compiler` plus utility classes/functions such as `ByteUtil` defined in package `edu.citadel.compiler.util`. There is also a package named `test.compiler` that contains a couple of test programs for the principal classes in this module. This module has no dependencies on the other modules. Its Java module definition is as follows.

```
module edu.citadel.compiler
  {
    exports edu.citadel.compiler;
    exports edu.citadel.compiler.util;
    requires kotlin.stdlib;
  }
```

2. **The CVM Module**
   Module `edu.citadel.cvm` (a.k.a. the CVM module) contains classes that implement the CVM, the CPRL virtual machine. It also implements a disassembler that converts object code into CVM assembly language. This module has a dependency only on the compiler module as described in item 1 above. Its Java module definition is as follows.

```
module edu.citadel.cvm
  {
    exports edu.citadel.cvm;
    requires kotlin.stdlib;
    requires edu.citadel.compiler;
  }
```

3. **The Assembler Module**
   Module `edu.citadel.assembler` (a.k.a. the assembler module) contains classes that implement an assembler for CVM assembly language. The assembler optionally performs several standard optimizations, such as converting `ADD 1` to `INC`. This module has a dependency on both the compiler and CVM modules as described in items 1 and 2 above. Its Java module definition is as follows.

```
module edu.citadel.assembler
  {
    exports edu.citadel.assembler;
    exports edu.citadel.assembler.ast;
```

```
    requires kotlin.stdlib;
    requires edu.citadel.cvm;
    requires transitive edu.citadel.compiler;
  }
```

4. **The CPRL Module**
   Module `edu.citadel.cprl` (a.k.a. the CPRL module) contains the classes that implement the CPRL compiler. Complete source code is provided for the other three modules described above, but only portions of the source code are provided for this module. **Although students will need to refer occasionally to the other three modules in order to understand the roles their classes play in developing the CPRL compiler, all new development will take place only in this module.** This module has a dependency on the Compiler and CVM modules as described in items 1 and 2 above. Its Java module definition is as follows.

```
module edu.citadel.cprl
  {
    exports edu.citadel.cprl;
    exports edu.citadel.cprl.ast;
    requires kotlin.stdlib;
    requires edu.citadel.cvm;
    requires transitive edu.citadel.compiler;
  }
```

   Since abstract syntax trees are not introduced until Project 4, the line

```
exports edu.citadel.cprl.ast;
```

   can be commented out or ignored out for Projects 0-3 below.

Overview of the Book's GitHub Repository

In addition to the standard license and readme files, the top level of the GitHub Repository at https://github.com/SoftMoore/CPRL-Kt-2nd contains four directories (a.k.a. folders). The Book directory contains PDF documents of several sections from the book including the Table of Contents, the Preface, and this Appendix. The Handouts directory contains several useful handouts. The PowerPoint directory contains PowerPoint slides that could be useful for college instructors that adopt this book as a course textbook. The slides correspond directly to the book chapters. The Project directory contains all files used for the compiler projects, as described below.

Warning: Several of the source files in the Project directory for latter projects are replacements or enhancements for the same files in earlier projects. For example, there are three versions of class `Parser` and two versions of class `IdTable`. Don't try to download the source files and import all of them at the same time into the IDE for your compiler project. Follow the project instructions for each project in the order presented.

Project 0: Getting Started

- This is not a real project but more of an initialization of your working environment for the remaining compiler projects. Expand the green "Code" button on the GitHub Repository page, download the repository as a zip file, and unzip it into a directory on your computer. Under the Project directory you will see 9 subdirectories as follows.

```
Project
  - bin
  - examples
  - src-Assembler
  - src-CVM
  - src-Compiler
  - src-ParserV1
  - src-ParserV2
  - src-ParserV3
  - src-Scanner
```

  Directories with names beginning "src..." contain Kotlin source files for the compiler project.

- Directory `bin` contains sample Bash shell scripts and Windows command scripts for running and testing various stages of the compiler. There are two subdirectories named "`bash`" and "`windows`" that contain the Bash and Windows script files, respectively. For each Windows "`.cmd`" script file there is a corresponding Bash script file without the "`.cmd`" suffix; e.g., `cprlc.cmd` and `cprlc`. Pick the collection of script files for your operating system and programming environment.

  As an example of the contents of `bin` for Windows, there is a script `cprlc.cmd` that will run the CPRL compiler on a source file whose name is entered via standard input. Similarly, there is a script `assemble.cmd` for running the assembler on "`.asm`" files, a script `disassemble.cmd` for running the disassembler on "`.obj`" files, and a script `cprl.cmd` for executing a single compiled/assembled CPRL program on the CVM.

```
cprlc Hello.cprl               // creates Hello.asm
assemble Hello.asm             // creates Hello.obj
cprl Hello.obj                 // executes Hello.obj on the CVM
disassemble Hello.obj          // creates Hello.dis.txt
```

  To compile all CPRL source files in the current working directory use `cprlc *.cprl`.

  There are two scripts for testing correct programs. Script `testCorrect.cmd` can be used to test a single CPRL program, and script `testCorrect_all.cmd` can be used to test all CPRL programs in the current directory. Both test scripts execute ".obj" files on the CVM and compare the output with expected output. Starting with Project 6 you can also use script `testIncorrect_all.cmd` to compile all incorrect CPRL programs in the current directory and compare the results to the expected results.

  Additionally, there are script files for testing the scanner and parser in the earlier projects.

You will need to edit the file `cprl_config.cmd` (or `cprl_config` for bash) so that the `COMPILER_PROJECT_PATH` variable "points to" the directories containing the Kotlin class files for your project.

Most of the other script files use `cprl_config.cmd` to set the module path appropriately. For example, my personal setup for Kotlin uses an IntelliJ IDEA project with four Java modules in four separate directories named `edu.citadel.assembler`, `edu.citadel.compiler`, `edu.citadel.cprl`, and `edu.citadel.cvm`, and all Kotlin class files are in corresponding subdirectories of `out\production`, the default for IntelliJ IDEA. When `cprl_config.cmd` is executed, it sets `COMPILER_PROJECT_PATH` to include the four subdirectories of `out\production` containing the classes. It also sets `COMPILER_PROJECT_PATH` to include the jar file `kotlin-stdlib.jar` installed by IntelliJ.

Note: The script files are written to use `MODULEPATH`, so you will need to edit all of the script files if you want to use `CLASSPATH` instead.

Also, you will need to ensure that the directory containing the script files is included in your `PATH` environment variable so that the operating system can find them when you enter their names on the command line.

You can set the global `PATH` environment variable permanently for your computer, or you can either execute `setPath.cmd` (Windows) or source `setPath` (bash) as appropriate to set the `PATH` environment variable temporarily for the current terminal session. For windows, simply change to the `Windows` subdirectory of `bin` and execute `setPath.cmd`. For Linux, change to the `Bash` subdirectory of `bin` and execute either "`source setPath`" or "`. setPath`".

- Directory `examples` contains examples of correct and incorrect CPRL programs that can be used to test various parts of your compiler. There are three subdirectories of `examples` as follows.
  - `Correct` contains numerous correct CPRL programs. The programs are organized into five subdirectories for testing different projects as you progress though the compiler implementation. For example, there is a subdirectory containing only test programs for CPRL/`0`, the zero subset of CPRL (no subprograms or composite types) as outlined in Project 6 below. **Testing should be performed cumulatively; i.e., you should always retest the CPRL/`0` example programs when working on later projects.**
  - `Incorrect` contains numerous incorrect CPRL programs that will be used for testing error detection and recovery. Similar to the correct programs, these programs are organized into five subdirectories for testing different projects as you progress through the compiler implementation.
  - `ScannerTests` contains both correct and incorrect files that can be used for testing your scanner as described in Project `1` below. These are not necessarily complete CPRL programs. For example, one of the files contains every valid symbol in CPRL including all reserved words, all operators, and numerous programmer-defined identifiers and literals.

You are strongly encouraged to develop additional test programs as you work though the remaining projects described below.

- Directory `src-Compiler` contains the source code for the compiler module as described above. These classes are used by the other three modules, and they are potentially reusable on other compiler projects. All classes in this directory are complete and require no additional work for use on the compiler project. **Import the Kotlin source code for this module into your preferred IDE for Kotlin (e.g., IntelliJ IDEA).** All code should compile without any errors.
- Directory `src-CVM` contains classes that implement the `CVM`, the virtual machine that will be used on subsequent projects to run CPRL programs. These classes are described in the CVM module above. All classes in this directory are complete and require no additional work for use on the compiler project. **Import the Kotlin source code for this module into your preferred IDE for Kotlin (e.g., IntelliJ IDEA).** All code should compile without any errors.
- Directory `src-Assembler` contains classes that implement an assembler for the `CVM`. You will run the assembler on assembly language files generated by your compiler to create machine code files that can be executed on the `CVM`. These classes are described in the assembler module above. All classes in this directory are complete and require no additional work for use on the compiler project. **Import the Java source code for this module into your preferred IDE for Java (e.g., Eclipse).** All code should compile without any errors.

Project 1: Scanner

- Using the concepts from Chapter 5, implement a scanner for CPRL.
- Import the source files from directory `src-Scanner` into your preferred IDE. Directory `src-Scanner` contains several classes in package `edu.citadel.cprl`. It has complete implementations for classes `Symbol`, `Token`, and `TokenBuffer` plus a partial implementation of class `Scanner`. Directory `src-Scanner` also contains a test driver named `TestScanner` in package `test.cprl` that can be used together with the `testScanner.cmd` (or `testScanner` for bash) script file to "wrap" your scanner and run it against the scanner example test files.
- **Complete the implementation for class `Scanner`.**
- Test your scanner with the scanner-specific test examples and multiple correct CPRL examples.

Project 2: Language Recognition

- Using the concepts from Chapter 6, implement a parser that performs language recognition for the full CPRL language (not just the zero subset) based on the language definition and context-free grammar given in Chapter 4, Appendix C, and Appendix D.

- Import the source files from directory `src-ParserV1` into your preferred IDE. Directory `src-ParserV1` contains complete implementations of classes `IdType`, `IdTable`, `ScopeLevel`, and `Scope` in package `edu.citadel.cprl` as described in Chapter 6, and it contains a partial implementation of class `Parser` that performs only language recognition. Directory `src-ParserV1` also contains a test driver named `TestParser` in package `test.cprl` that can be used together with the script file `testParser.cmd` (or `testParser` for bash) to "wrap" both your scanner and parser together and run them against the example test files. The "incorrect" example subdirectories contain text files showing the results that you should expect when running this version of the parser against the incorrect test examples.
- **Complete the implementation for class `Parser`.**
- Do not implement error recovery for this project; i.e., when an error is encountered, simply report the error and exit compilation as described in Chapter 6. Some of the parser methods are fully implemented; study those as examples for the incomplete parser methods.
- Test with all correct and incorrect examples. At this point the parser should accept all correct programs and reject all incorrect programs **except** those with type errors and/or miscellaneous errors. Detection of type errors and miscellaneous errors will be implemented in subsequent projects. Use the text files showing expected results as guides.

Project 3: Error Recovery

- Using the concepts from Chapter 7, add error recovery to your parser.
- Directory `src-ParserV2` contains only one class in package `edu.citadel.cprl`, a partial implementation of class `Parser` that demonstrates how to add error recovery to the parser methods. Do **not** import this class into your IDE since you already have an implementation for class `Parser`. Instead, use the class provided in this directory as a guide to manually edit your existing parser in order to add error recovery. Use the test driver and script files from the previous project to run your parser against the example test programs. The "incorrect" example subdirectories contain text files showing the results that you should expect when running this version of the parser against the incorrect test examples.
- **Edit your parser from the previous project to implement error recovery as described in Chapter 7.**
- Test with all correct and incorrect examples. At this point the parser should accept and reject exactly the same example programs as for the previous project, but this time your parser should report more than one error for some of the incorrect programs. Use the text files showing expected results as guides.

Project 4: Abstract Syntax Trees

- Using the concepts from Chapter 8, add generation of abstract syntax trees to your parser. Most parsing methods should return AST objects or lists of AST objects. From now on we will start referring to our implementation as a "compiler" even though it doesn't yet generate code.
- Directory `src-ParserV3` contains full or partial implementations of more than 40 AST classes in package `edu.citadel.cprl.ast`. Approximately half of the AST classes are implemented completely, while the remaining AST classes have only partial implementations. You should import the AST classes into your IDE. For now there are empty bodies or partially incomplete bodies for methods `checkConstraints()` and `emit()` in the AST classes that are not fully implemented.

  Directory `src-ParserV3` also contains full or partial implementations of several classes in package `edu.citadel.cprl` as follows.

  - A complete implementation for class `Compiler`. For this project we will continue to use `TestParser`, but in the next project you will start using class `Compiler` instead of `TestParser` to test your work.
  - Complete implementations for classes `LoopContext` and `SubprogramContext`. As described in Chapter 8, these classes are used to track entry to and exit from loops and subprograms. Use `LoopContext` when parsing loop statements and exit statements. Use `SubprogramContext` when parsing procedure declarations, function declarations, and return statements.
  - Complete implementations for classes `Scope` and `IdTable`. As described in Chapter 8, `IdTable` now stores scopes with references to an identifier's declaration. These two classes replace the implementations of `Scope` and `IdTable` that we have been using in the previous two projects. Use the complete version of `IdTable` to check for declaration and scope errors. With this new implementation for `IdTable` your parser will now be able to detect additional scope errors when we implement subprograms in Project 7.
  - A complete implementation for class `Type` and implementations for its subclasses `ArrayType`, `StringType`, and `RecordType`. The implementations for these three subclasses will compile, but at this point they are incorrect. You will correct the implementations for these three classes as part of projects 8, 9, and 10. The correct implementations for `ArrayType`, `StringType`, and `RecordType` are straightforward after studying the relevant chapters in the book; i.e., Chapters 14, 15, and 16.
  - A partial implementation for class `Parser`. As described in Chapter 8, most parsing methods now return AST objects or a lists of AST objects. Do **not** import this class into your IDE since you already have an implementation for class `Parser`. Instead, use the class provided in this directory as a guide to manually edit your existing parser in order to add generation of AST classes.
- **Edit your parser from the previous project to add generation of AST classes or lists of AST classes.**

- Test your parser with all correct and incorrect examples. The "incorrect" example subdirectories contain text files showing the results that you should expect when running this version of the parser against the incorrect test examples. The major differences in test results between versions 2 and 3 of your parser are that version 3 should also detect `exit` statements that are not nested within loops.

Project 5: Constraint Analysis for CPRL/0

- **Using the concepts from Chapter 9, implement `checkConstraints()` methods in the AST classes to perform full constraint analysis for the CPRL/0 subset (everything except subprograms and composite types).** In addition to syntax and scope errors previously detected by your compiler, your compiler should also detect all type and miscellaneous errors for the CPRL/0 subset. Do **not** implement any constraint checks for class `Variable`. All constraint checks for class `Variable` involve arrays, strings, or records, which will be covered in subsequent chapters.
- Test with all correct and incorrect CPRL/0 examples. Henceforth use `Compiler` rather than `TestParser` to test your implementation with the CPRL examples. To invoke the compiler, use script `cprlc.cmd` or `cprlc` as appropriate. You can use "`cprlc *.cprl`" to compile all files in the current directory. The compiler should accept all correct CPRL/0 examples and reject all incorrect CPRL/0 examples. The "incorrect" example subdirectories contain text files showing the results that you should expect when running this version of the parser against the incorrect test examples.

Here are some tips for working on the remaining projects.

You will eventually want to run the scripts `testCorrect_all` and `testIncorrect_all` in the appropriated `examples` subdirectory, but start by running `cprlc *.cprl`.

For correct programs, if you don't get a clean compile, then work on each problem example separately. Compile individual examples using something similar to `cprlc Correct_101.cprl`.

Once you get a clean compile on every file, do the same thing with script `assemble`; i.e., start with `assemble *.asm` and use `assemble Correct_101.asm` (or similar) as needed. If there are problems, compare the `.asm` files you are generating to the `.asm` files downloaded from the book's GitHub repository. You can re-download the examples in the event that you overwrote yours when testing your compiler and didn't save copies in a different location.

Once you get a clean compile and a clean assemble, you can run the example's `.obj` file using something like `cprl Correct_101.obj`, or you can simply run `testCorrect Correct_101`.

Project 6: Code Generation for CPRL/0

- **Using the concepts from Chapter 11, implement `emit()` methods in the AST classes to perform code generation for CPRL/0.** At this point you are actually generating assembly language for the CVM.

- Use the assembler provided in the Assemble module (see Project 0 above) and script file `assemble.cmd` (`assemble` for bash) to generate machine code for all correct CPRL/`0` examples.
- Use script files `cprl.cmd`, `testCorrect.cmd`, and `testCorrect_all.cmd` (or bash equivalents) to run and test all correct CPRL/`0` examples. Use script file `testIncorrect_all.cmd` (or bash equivalent) to test all incorrect CPRL/`0` examples.

Project 7: Subprograms

- **Using the concepts from Chapter 13, add constraint analysis and code generation for subprograms.**
- Test all correct and incorrect Subprogram examples. Retest all correct and incorrect examples from CPRL/`0`.

Project 8: Arrays

- **Using the concepts from Chapter 14, add constraint analysis and code generation for arrays.**
- Correct the implementation for class `ArrayType` plus any remaining errors.
- Test all correct and incorrect Arrays examples. Retest all correct and incorrect examples from CPRL/`0` and Subprograms.

Project 9: Strings

- **Using the concepts from Chapter 15, add constraint analysis and code generation for strings.** Most of the work involved in implementing strings is already provided in the GitHub repository, so this project is relatively simple. Depending on student backgrounds and course prerequisites, instructors might elect to make this the last project and assign project 10 as a reading assignment.
- Correct the implementation for class `StringType` plus any remaining errors.
- Test all correct and incorrect Strings examples. Retest all correct and incorrect examples from CPRL/`0`, Subprograms, and Arrays.

Project 10: Records

- **Using the concepts from Chapter 16, add constraint analysis and code generation for records.** Completion of this project results in the final version of your compiler.
- Correct the implementation for class `RecordType` plus any remaining errors.
- Test all correct and incorrect Records examples. Retest all correct and incorrect examples from CPRL/`0`, Subprograms, Arrays, and Strings.