Project: Compiler and Virtual Machine for A Programming Language.

Durations:

About 7 weeks

Teams

You will do this project in teams assigned by the CATME tool. Your P2 teams will be different than the P1 teams.

Submission

Create a bitbucket team and include your teammates as well as the grader. Name your bitbucket repo “SER502-P2”. In the repo you should have a file named “README.md” that contains a link to your teams’s video.

Description

Everyone has their favorite language; and also their little gripes about the language. This project is your chance to try and do better! You are to design, implement, and demonstrate your own language, compiler, and runtime for a language.

You are to implement a lexical analyzer, parser, intermediate code generator, and runtime environment (intermediate code executor) for a simple programming language. This is a team project, and is to be completed in teams of three or four students.

You will be expected to present your work in class and you will be expected to evaluate others and ask questions in class. In the interest of time all presentations will be pre-recorded as 5 minute videos to be shown in class, the group will stand in front of class and answer questions for an additional 5 minutes. Groups must complete presentations and share them with the class 48 hours before class on the day you present.

You will be expected to find a way to record and edit videos to meet the 5 minute deadline while including all the necessary information for your presentations.

Outcomes Addressed

This assignment addresses the following course outcomes:

- Communicate, apply and evaluate tools, activities and artifacts involved in programming language design, translation and execution. Working with program compilation using tools such as Flex and Bison or hand-constructed lexical analyzer and parser, to recognize and translate a language into an intermediate form.
- Understanding and design of language runtime environments, especially accessing user-defined program values (primarily data and methods), managing runtime storage, and fundamental operations needed to implement a simple programming language.
- Using regular and context-free language specifiers (regular expressions and context-free grammars,) and recognizers.

Teams & Collaboration

- You are expected to work in teams of 3-4.
- You are expected to create a calendar and set appropriate reminders so that your team stays on top of deadlines.
- You must create a **git** project on **bitbucket** (all students in the class must have access)
- You must use the ***ticketing*** system to create and assign tasks to individual team members. **Tickets must be evenly assigned to all team members**.
- Every member must make **at least two significant commits** to the project per week. If the project log shows a lack of commits or a tendency to commit at the last minute then you will lose points.
- All deliverables should be reflected in the project wiki. Presentations should be linked to from the project README as slideshows (PPT, Google, or PDF) as well as videos (MP4 or YouTube links). Again videos and slides must be accessible from all students and the instructor.
- Someone from your team must announce each presentation 48 hours before class.

Milestones

- **Pitch[~2wks]**. Your team should provide a 15 minute presentation that covers your language design, example programs, and corresponding intermediate code for the examples. The language should be described from the perspective of included features and their meaning/definition. It need not include a grammar for the language. The presentation should include 3 to 5 representative example programs in the language. Taken together, the examples should demonstrate all of the features included in your language. Your presentation should also include a specification of the intermediate form that you've designed. Show and explain the intermediate code for each of the example programs.
- **Design[~2wks]**. Your team should provide a 15 minute design review driving the implementation of the compiler and byte-code interpreter/runtime environment. Include a grammar for the language, together with a brief description of any changes that the team has made to the language since its original definition. Also include information

about the interpreter. Provide a class diagram for each program, as appropriate. Discuss the parsing technique you will employ. Also discuss any data structures used by the parser and interpreter. Describe the runtime environment for the program being executed by your interpreter.

- **Implementation[~3wks]**. Final delivery and presentation. Provide a presentation and demonstration of your final product. Include language and intermediate code definitions. Your final delivery should be packaged into a single project directory whose name is the name of your language (DKJ, Enigma, NeoC, SARS, or SLRR) that includes the following:
    - Source directory **src** that includes subdirectories **compiler** and **runtime**. Include the language source files in these sub-directories. If you use tools such as flex or bison, include their input files in these directories. Make sure that all of the source files have comment headers indicating the authors, purpose, and version or date.
    - **readme.md**. Include the system on which your compiler and runtime are built (GNUstep or MacOS), and directions for building and running your language tools.
    - Documentation directory, **doc**. Include the presentation powerpoints from the class presentations (updated if necessary to match the final version of the language and byte-code). Also include a reference manual which provides complete language description and byte-code definition. Be sure to include team member names as authors of all documents that are included in the doc directory.
    - Build script. Use an appropriate build script for your program. You may use cmake, qmake, ant, make, or whatever is most appropriate given the tools you used. You may not rely on the use of an IDE, a GUI, or a proprietary tool to build your project. Your projects readme must include the ONE LINE of bash script that builds the compiler and ONE LINE that runs the compiler.
    - Data directory **data**. This directory should contain sample source programs in your language and sample intermediate code generated by your compiler.
    - A list of project dependencies. You must receive approval for depending on tools other than those discussed in class. You may assume your program is built and run in the latest version of Ubuntu linux.

Your compiler should and interpreter should be invoked from the command-line with a single command line argument, which is the input file. For example:

```
./obj/enigmac Data/Factorial.enigma
```

or

```
java -cp classes cst502.EnigmaC Factorial.enigma
```

if its written in Java.
Your compiler should produce a text file containing code cells of the intermediate language that you design, for example in the file: **Data/Factorial.enigma.int**. The intermediate code file

(analogous to Java's bytecode), should be read in and executed by the runtime environment that you produce.

Language and Byte-Code Constraints

Here is a list of constraints that your language and byte-code designs must satisfy:

1. The language you design must have operators and primitive types for at least boolean values and at least one numeric type (int, float, double, for example).
2. The language you design must support parameterized procedures, functions, or methods. These must provide for some sort of local variables (see types above) and non-local variables.
3. The language must have a way to associate a value with an identifier, such as an assignment statement.
4. The language must support at least one construct to make decisions, such as an if-then-else construct.
5. The language must support at least one construct that provides for iterative execution, such as a while or for statement.
6. The language must be powerful enough to implement factorial computation, and a simple data structure definition and usage. For example, a stack.
7. Your language will also need to be powerful enough to define recursive methods.
8. Your byte-code language must be at the level of a byte-code language. Use a stack machine model, or select a model in which instructions have an opcode and one or two operands. That is, it must be a low-level language, and your compiler must have to translate from a high-level source language into a low-level language.

Base your language and your compiler on one (or both) of the following examples:

- Example using Bison and Flex to parse an expression grammar. parsing.jar.
- An example printing intermediate code for assignment statements. assigStmts.jar.

Sample Programs from the Double Programming Language

Here are some sample programs from prior years of offering the course.

- Example computing the factorial of an integer. Factorial.dpl
- Example computing the power function, x**n. Power.dpl
- Example computing summation 1 to n by algebra. Summation.dpl
- Example computing summation 1 to n by iteration. SummationIterate.dpl
- Example computing the factorial of an integer. TestMathExp.dpl
- Example testing nesting of while statements. TestNest.dpl
- Here is an example of the instructor's intermediate code for the Factorial program: Factorial.dpl.int

Grading Criteria

You will have a single grade on this assignment, that is based on the success of your software project (both programs), your presentations, and your documentation. In grading the project, the following criteria will be considered:

- **Lexical analyzer**. Rules defined in the input to flex or your lexical analyzer, and consistency with the parser.
- **Parsing**. Does the grammar exist and accurately define the language. Do the grammar and lexical analyzer work together appropriately.
- **Runtime**. Do the parser and runtime environment agree on the intermediate language? Does the runtime properly execute producing a correct result for each sample program?
- **Documentation**. Does the **doc** directory contain everything called for in the assignment?
- **Intermediate Code**. Does your parser generate an appropriate intermediate code, which is processed by the runtime environment? An intermediate code is appropriate when it can be reasonably produced by a parser (correspondence between the grammar rules and output of code), it represents the source program at a lower level, and it can be effectively executed by the runtime environment.
- The quality of your presentations, and the extent to which all team members have contributed to presentations and the final product.

The graders will verify that your code matches what you have presented, but the primary source graders use to verify these criteria will be your presentation videos.

Solutions

Be aware that there may have been subtle differences in the required formats etc in past semesters. These give you some ideas but use this document for precise details of what / how to submit.

- **Neo-C**. NeoC is a C-based language with nested blocks, procedures/functions, and procedure references. neoc_v4.jar
- **SLLR**. SLRR by Linda, Ryan, Sanjay, and Rishi is a programming language thats best characterized as a method-based language whose syntax reads like having a conversation with yourself. The compiler is written in **C** using **flex**, and **bison**. The runtime is written in **Java**. Download, extract and build with **Ant**: slrrAmended.jar
- **SKARS** SKARS_ver2.0.jar
- **Enigma** BNT_enigma.jar
- **DKJ** DKJassign1.zip

What To Hand-In

Your project will be graded by the state of your bitbucket repo as it appears at some undetermined time from between 48 hours before your final presentation and the time grades are reported (within 1 week of the presentation).

At least 48 hours before your presentations, please do the following:

- Check to assure your solution conforms to the directions on this page.
- Check grading criteria for this assignment to be sure you address all criteria and constraints defined on the assignment page.
- Clean your solution directory of any temporary files. Pull your solution from the remote (bitbucket) site into another directory, build and execute the solution to ensure everything has been pushed