Assignment 3 - Jack Compiler

Weighting and Due Dates

- Marks for this assignment contribute 10% of the overall course mark.
- Marks for functionality will be awarded automatically by the web submission system.
- **Due dates: Milestone** - 11:55pm Friday of week 11, **Final** - 11:55pm Friday of week 12.
- **Late penalties:** For each part, the maximum mark awarded will be reduced by 25% per day / part day late. If your mark is greater than the maximum, it will be reduced to the maximum.
- **Core Body of Knowledge (CBOK) Areas:** abstraction, design, hardware and software, data and information, and programming.

Project Description

In this assignment you will complete a variation of projects 10 and 11 in the nand2tetris course, reworked descriptions of **Nand2Tetris Projects 10 and 11** are shown below. In particular, you will write the following programs that are used to implement different components of an optimising Jack compiler that compiles a Jack class into Hack Virtual Machine (VM) code:

- **jparser** - this parses a Jack program and constructs an abstract syntax tree.
- **jcodegen** - this takes an abstract syntax tree and outputs equivalent VM code.
- **jpretty** - this takes an abstract syntax tree and produces a carefully formatted Jack program.
- **jopt-r** - this copies an abstract syntax tree and removes redundant code.
- **jopt-e** - this copies an abstract syntax tree and evaluates constant expressions.

SVN Repository

You must create a directory in your svn repository named: **<year>/<semester>/cs/assignment3**. This directory must only contain the following files and directories - the web submission system will check this:

- **Makefile** - this file is used by **make** to compile your submission - **do not modify this file.**
- **.cpp** C++ source files - naming as specified in the component program requirements.
- **.h** C++ include files - naming as specified in the component program requirements.
- **lib** - this directory contains precompiled programs and components - **do not modify this directory.**
- **includes** - this directory contains **.h** files for precompiled classes - **do not modify this directory.**

- **tests** - this directory contains a test script and test data, you can add your own tests too.

Note: if the file **lib/lib.a** does not get added to your svn repository you will need explicitly add it using:

```
% svn add lib/lib.a
```

Submission and Marking Scheme

This assignment has two assignments in the web submission system named: ***Assignment 3 - Milestone Submissions*** and ***Assignment 3 - Final Submissions***. The assessment is based on "Assessment of Programming Assignments".

Assignment 3 - Milestone Submissions: due 11:55pm Friday of week 11

The marks awarded by the web submission system for the milestone submission contribute up to 20% of your marks for assignment 3. Your milestone submission mark, after the application of late penalties, will be posted to the myuni gradebook when the assignment marking is complete.

Your programs must be written in C++ and will be tested using Jack language programs that that may or may not be syntactically correct and previously generated abstract syntax trees. Although a wide range of tests may be run, including some ***secret*** tests, marks will only be recorded for those tests that require a working **jack_parser** program. Your programs will be compiled using the **Makefile** and precompiled components in the **lib** directory**. Note**: you will get no feedback on the ***secret*** tests, even if you ask!

Assignment 3 - Final Submissions: due 11:55pm Friday of week 12

The marks awarded for the final submission contribute up to 80% of your marks for assignment 3.

Your final submission mark will be the geometric mean of the marks awarded by the web submission system, a mark for your logbook and a mark for your code. It will be limited to 20% more than the marks awarded by the web submission system. See "Assessment - Mark Calculations" for examples of how the marks are combined. Your final submission mark, after the application of late penalties, will be posted to the myuni gradebook when the assignment marking is complete.

Automatic Marking

The automatic marking will compile and test both of your tokenisers in exactly the same way as for the milestone submission. The difference is that marks will be recorded for **all** of the tests including the ***secret*** tests. **Note**: if your programs fail any of these ***secret*** tests you **will not** receive any feedback about these ***secret*** tests, even if you ask!

The marks from the automatic tests for each component program will be weighted as follows:

- **jparser** - 30%
- **jcodegen** - 40%
- **jpretty** - 10%
- **jopt-r** - 20%

The test scripts are able to test each component program independently but, we strongly suggest that you attempt the component programs in the order above so that you get the most return on your effort. You should gain the most significant learning benefit from completing both the **jparser** and **jcodegen** programs.

Logbook Marking

Important: the logbook must have entries for all work in this assignment, including your milestone submissions. See "Assessment - Logbook Review" for details of how your logbook will be assessed.

Code Review Marking

For each of your programming assignments you are expected to submit well written **code**. See "Assessment - Code Review" for details of how your code will be assessed.

Nand2Tetris Projects 10 & 11: Compiler I & II

Background

Modern compilers, like those of Java and C#, are multi-tiered: the compiler's front-end translates from the high-level language to an intermediate VM language; the compiler's back-end translates further from the VM language to the native code of the host platform. In an earlier workshop we started building the back-end tier of the Jack Compiler (we called it the VM Translator); we now turn to construct the compiler's front-end. This construction will span two parts: syntax analysis and code generation.

Objective

In this project we build a Syntax Analyser that parses Jack programs according to the Jack grammar, producing an abstract syntax tree that captures the program's structure. We then have a choice, we can morph the logic that generates the abstract syntax treeinto logic that generates VM code or we can write separate logic that can apply any number of transformations to our abstract syntax tree. The transformations may include pretty printing the original program, applying specific optimisations to the abstract syntax tree or generating VM code. This mirrors the alternative approaches used in the workshops.

Resources

The relevant reading for this project is Chapters 10 and 11. However, you should follow the program structure used in earlier workshops rather than the proposed structure in Chapters 10 and 11. **You must write your programs in C++.** You should use the Linux command **diff** to compare your program outputs to the example output files supplied by us. A set of precompiled classes similar to those used in the workshops and the previous assignment are in the zip file attached below. All the test files and test scripts necessary for this project are available in the zip file attached below.

Component Functions

We have a provided a description of the requirements for each component program on its own page. This includes instructions on how to compile, run and test each component program.

jparser

The **jparser** program uses the provided tokeniser to parse a Jack program and construct an equivalent abstract syntax tree. The specific requirements for this component program are described on the jack_parser() page.

jcodegen

The **jcodegen** program traverses an abstract syntax tree to generate virtual machine code. The specific requirements for this component program are described on the jack_codegen() page.

jpretty

The **jpretty** program traverses an abstract syntax tree produced and prints a Jack program formatted to a specific coding standard. The specific requirements for this component program are described on the jack_pretty() page.

jopt-r

The **jopt-r** program traverses an abstract syntax tree produced and generates a new abstract syntax tree with redundant program elements removed. The specific requirements for this component program are described on the jack_optimiser_r() page.

Testing

The test data including the convention used to name expected outputs for each test are described on the Testing page.