Second Year Compilers Exercise: The MAlice Compiler

Project Introduction

Summary

You are to write a compiler which, given a valid MAlice input file, will generate executable code for an architecture of your choice. You will be working in groups of up to 3.

The Input Language

In 1865 the author Lewis Carroll published a book called Alice's Adventures in Wonderland. The novel tells the story of Alice, a young girl who falls down a rabbit hole. It has remained popular, not least because of its many film adaptions.

Alice in Wonderland and many of Carroll's other works are categorised as "Nonsense literature"[1], a genre in which what seems like gibberish takes on semantic, syntactic and contextual meaning.

You will be writing a compiler for MAlice, a language inspired by the adventures of Alice in Wonderland.

While it may not be immediately apparent, the example programs given all have a well defined meaning and the language is describable by a LALR grammar. An understanding of compiling procedural languages is all that is required to begin working on the project.

Milestones

- Milestone 1: You will be asked to write a language specification for a subset of the MAlice programming language. [15%]
- Milestone 2: You will be asked to implement a compiler for the subset of MAlice seen in Milestone 1. [35%]

[1]Wim Tigges 'An Anatomy of Literary Nonsense' describes the genre well

- Milestone 3: You will be asked to implement a compiler for the full MAlice programming language. [35%]
- Extensions: You will be asked to implement an extension to either the MAlice language or your compiler. [15%]

Each milestone is specified in a separate exercise specification document.

Advice

Version Control

As with all large projects you will find it useful to use some version control software. Git[2], Mercurial[3] and SVN[4] are all installed on the lab machines. CSG provides guides on how to set up SVN at http://www.doc.ic.ac.uk/csg/guides/version-control/ but this is by no means an endorsement for SVN and you should use whichever software you feel suits you best.

Learning how to use version control effectively will lead to a huge increase in productivity and should be seen as an essential milestone on your path to becoming a professional programmer.

Makefiles

'`make`' is a powerful tool which is used to determine which elements of a program need to be recompiled.

A Makefile contains a series of rules which describe the relationships between target files and prerequisite files and the commands required to compile the target files.

The command '`make`' uses the rules in the Makefile and the timestamps of the prerequisite files to decide which commands to run to generate the target files.

A simple Makefile which builds the C file hello_world is shown below.

```
all: hello_world.o
	gcc hello_world.o -o hello_world
hello_world.o: hello_world.s
	as -o hello_world.o hello_world.s
hello_world.s: hello_world.c
	gcc hello_world.c -S -o hello_world.s
```

[2]http://git-scm.com
[3]http://mercurial.selenic.com/
[4]http://svnbook.red-bean.com/

The command `'make'` will recursively check each of the prerequisites of `all` to find which files need to be recompiled.

You will be asked in Milestones 2 and 3 to provide a Makefile which will compile all the component parts of your compiler.

More information on Makefiles can be found in The Gnu Make Manual[5].

Design

Until this point in your university career you have been asked to write programs without much complexity. The compiler project will not fall into this category.

From the very beginning of the project you should carefully consider your design decisions. A hasty decision early on may make future progress exponentially more difficult. Try to understand the entire problem and then to break it down into its logical component parts. Work out how these component parts are related and consider how they will interact with each other.

The structure of the milestones is such that if you have designed a very tightly coupled system you will find the final milestone significantly more challenging than if you had an easily expandable system to begin with.

In summary, don't use the Ball of Mud design pattern[6] you will only hurt yourself!

Resources

This exercise is left as open-ended as possible. You will be asked to produce a full compiler with very little additional guidance beyond the content of the lecture slides. You will probably find yourself stuck on what to do at some point during the project. When this happens you may find it useful to consult a textbook or the internet. Remember you **MUST** give proper credit for any code lifted directly from a website or textbook. remember also that code you find will not neccesarily be a good example of coding style, design or efficiency.

The following textbooks are recommended reading for the course. You may find them useful for this project (one is probably enough!).

- **Engineering a Compiler** - Keith Cooper and Linda Torczon, Morgan Kaufmann/Elesvier
- **Modern Compiler Implementation in Java** - Andrew Appel
- **Compilers: Principles, Techniques and Tools (second edition) [The 'Dragon Book']** - Alfred V. Aho, Ravi Sethi, Jeffery D. Ullman, Monika Lam
- **Advanced Compiler Design and Implementation** - Steven Muchnick

[5]http://www.gnu.org/software/make/manual/make.html#Introduction
[6]http://www.codinghorror.com/blog/2007/11/the-big-ball-of-mud-and-other-architectural-disasters.html

Getting the Most From the Exercise

- Many of the concepts introduced in the compilers course will seem very abstract in the context of lecture notes. If you take the time to try to implement them in your MAlice compiler you are likely to have a much firmer grasp of them when you sit the exam.
- A strong implementation will provide a good reference when you come to revise. If the design is clear and the implementation is readable you will find it much easier to revise from. It is therefore in your own interests to be as thorough, coherent and consistent with your code style as possible.
- Do not be afraid of throwing away your code if it doesn't work. If you have made decisions which are hindering your progress then you should feel comfortable making even very fundamental changes.
- Do not be afraid to try things. If you cannot understand why the textbook does something in a certain way then try your own ideas. You may notice that your method only covers certain cases and the generalisation gives the approach shown in the book.
- Be critical of your own work. Knowing the flaws in your workflow will allow you to improve it.
- Work iteratively. Continually consider how you could improve the design and the implementation. Test regularly.
- Don't leave the entire project to the final week. You will find that as you rush to finish you begin to neglect the design and readability of your code. If you are rushing to implement the fundamentals you won't have time to challenge yourself and you will miss out on a useful portion of the exercise.
- Be competitive! If another group has a more efficient compiler, generates better code, has a cleaner design or has more interesting features then challenge yourself to do better. Copying their code will teach you nothing, but having them explain the methods they are using will be useful for both groups.
- Be considerate to your group members. Try to help someone who is struggling to understand the concepts you are implementing. Give the dedication to your group that they deserve.

The best things about this exercise come from James Greenhaigh and Kyrylo Tkakov. Advice and a little polishing came from Paul Kelly, Naranker Dulay and Tony Field. Responsibility for the shortcomings falls entirely on the lecturers.