The MAlice Compiler: Second Year Compilers Exercise

Second Year Computing Laboratory
Department of Computing
Imperial College London

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 adaptations.

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 Look-Ahead Left to Right (LALR) grammar. An understanding of compiling procedural languages is all that is required to begin working on the project.

Project Management

You are not assessed on how your group is organised, but a well organised group will be able to do more in the time available.

Version Control

As with all large projects, you will find it useful to use some version control software. 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.

For this exercise we have set up a Git repository for your group to use. Note that as with the previous lab exercises, this repository will be taken to be your final submission. Be aware that we will be looking at your commit logs to determine the level of input of each group member. It is therefore important that you divide the work between you and use meaningful commit messages that indicate which group members were invloved in that commit.

Further details on how to access your repository will be included the Milestone 1 specification document.

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

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

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

hello_world: hello_world.o
    gcc hello_world.o -o hello_world

hello_world.o: hello_world.c
    gcc -c hello_world.c

clean:
    rm -rf *.o hello_world
```

The command `'make'` will recursively check each of the prerequisites of `all` to find which files need to be recompiled. Running `'make clean'` will remove all of the `.o` files and the `hello_world` binary from the current directory. In Milestones 2 and 3 you will be asked 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[2].

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 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[3], 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 necessarily be a good example of coding style, design or efficiency.

The following textbooks are recommended reading for the Compilers course. You may find them useful for this project.

- **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

[2] `http://www.gnu.org/software/make/manual/make.html#Introduction`
[3] `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 work-flow 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.

Timetable and Assessment

The timetable for the MALice lab, and the weighting of marks, are given below:

Date	**Task**	**Mark %**
17th October 2012	Organise your Group	–
5th November 2012	Milestone 1: Specification	15%
26th November 2012	Milestone 2: Compiler Front-End	35%
14th December 2012	Milestone 3: Compiler Back-End	30%
14th December 2012	Report and Extensions	20%

Organise your group

Your first task is to set up a group of *up to three* people for the MAlice lab. You should then submit this information to CATe by uploading a text document with the logins of your group members and signing off on this, as is standard for group submissions.

Note that the text document is only being used as a back-up. The groups created on CATe will be used to assign shared group directories for the lab. By the start of week 4 (22nd October) you will receive an e-mail from CSG telling you the location of your shared group directory and how to access it.

Milestone 1: Specification

You will be asked to write a language specification for a subset of the MAlice programming language. This specification will provide the basis for the front-end of your compiler. A formal specification is not required. You should, however, aim to be accurate enough that were your specification to be given to a competent programmer unfamiliar with the MAlcie language they would be able to write a valid program. More details on this milestone can be found under the respective exercise on CATe.

Milestone 2: Compiler Front-End

You will be asked to implement the front-end of a compiler for the full MAlice programming language. This front-end will consist of a lexer, a parser and a semantic analyser. Your front-end must be able to successfully parse any valid MAlice program and generate appropriate error messages for invalid MAlice programs. Note that at this stage you do not need to generate any code.

This exercise should help you get to grips with the tool-chain you will use to implement the full MAlice compiler in Milestone 3. More details on this milestone can be found under the respective exercise on CATe.

Milestone 3: Compiler Back-End, Report and Extensions

You will be asked to implement a full compiler for the MAlice programming language using the lessons you have learned in the previous milestones. In particular, you must add a code generator to the back-end of the compiler, including code generation optimisations (such as constant propagation). You are welcome to write the compiler for an architecture of your choosing, but should consider the costs and benefits of each when making your choice. You must also ensure that your generated code can be run on the lab machines.

You will also be asked to implement an extension to either the MAlice language or your compiler. You are free to add any feature you want to any aspect of the compiler, from the language specification to the code generation. This is your chance to implement some of the more exciting compiler features that you were exposed to during the lectures. You will then have to write a short report summarising outcomes your project. More details on this milestone can be found under the respective exercise on CATe.