The MAlice Compiler: Milestone 3

Second Year Computing Laboratory
Department of Computing
Imperial College London

Summary

So far you have only built the front-end of your MAlice compiler. You will now use the lessons learned in the previous Milestones to implement a full compiler for the MAlice language. That is, you will add a code generator back-end to your MAlice compiler.

You will also implement some extensions to your compiler and produce a report on your project.

Details

Recall the four stages of the compilation process:

1. **Perform Lexical Analysis:** splitting the input file into tokens.
2. **Perform Syntactic Analysis:** parsing the tokens and creating a representation of the structure of the input file.
3. **Perform Semantic Analysis:** working out and ensuring the integrity of the meaning of the input file.
4. **Generate Machine Code:** synthesizing output in the target language, maintaining the semantic meaning of the input file.

You should already have built the front-end of your compiler in Milestone 2, which should handle the fist 3 stages. This means that you should be able to perform lexical analysis, syntactic analysis and semantic analysis on MAlice programs to determine if they are valid. You should also be able to detect invalid MAlice programs and report appropriate error messages.

For this milestone you need to work on the back-end of your compiler, the code generation stage. Your compiler must be able to produce assembly code, for an architecture of your choosing, when given a valid MAlice program. You should consider the costs and benefits of each architecture, along with your knowledge of the architecture, when making your choice.

Note that the three-address code generated by the reference compiler is not considered an appropriate target language, as it is aimed at a bespoke virtual machine. You should instead generate assembly code for a *real* machine, e.g. x86, or an *actual* virtual machine, e.g. LLVM bitcode. You are free to generate three-address code during your compilation process (this might make it easier to perform certain optimisations), but the final output should be some executable code.

It is likely that you may need to modify your parser so that it produces suitable output for the code generator. You should also consider code generating optimisations, such as constant propagation.

As with the previous milestone, you will probably find it useful to refer to the reference implementation of the MAlice compiler. The reference compiler generates three-address code for each input file that it successfully parses. Looking at these output files should help you design the output for your own MAlice compiler.

Submit by: 14th December 2012

What To Do:

1. Write a back-end for your compiler so that you have a full compiler for the MAlice programming language. When designing and developing your compiler, you may find it useful to go back to your lecture notes and try to understand if any of the discussed optimisations can be applied.

2. For the final part of this lab exercise you are expected to extend either the compiler or the language with a new feature. 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. Some ideas for extensions are:

   - Structures.
   - Object-oriented features.
   - Control flow optimisation.
   - More efficient register allocation.
   - Further semantic and runtime checks.
   - Dynamic memory allocation and garbage collection
   - Additional data types
   - Improved string handling (e.g. string concatenation)
   - Interfacing with C functions and libraries

   You are also welcome to propose your own extensions, but you should have this approved by one of the lab organisers before beginning implementation to ensure that it is appropriate.

   If you have been careful in designing your compiler, extensions should not be difficult to incorporate. Marks will be awarded based on the complexity of the extension, the functionality of its implementation and the clarity of its explanation in the report.

3. Write a short report (*four pages maximum*) providing a summary of your project. The main goal of this exercise is to promote reflection on your experiences during the MAlice lab. Assessment will focus on the professionalism and analysis you are able to demonstrate. The report should be divided into 3 sections:

   - **The Product**: An analysis and critical evaluation of the quality of the compiler you built You should consider both whether it meets the functional specification and whether you judge that it forms a sound basis for future development. You may wish to address performance issues.
   - **The Design Choices**: An analysis of the design choices that you made, including an evaluation of the tools you chose to use or a reflection on your decision not to use tools. You should also discuss what went well and what you would do differently if you were to do the lab again.
   - **Beyond the Specification**: An evaluation of what you think are the most interesting directions for extending the project. You should focus on the language extensions, optimisations, or other aspects that you have added to your compiler. You may wish to cite evidence from your preliminary extensions.

Your compiler will be tested by a script which will run the following commands:

```
> make                         to build your compiler
> ./compile FILENAME.alice     to compile FILENAME
> ./FILENAME                   to run the compiled executable
```

You should therefore provide a Makefile which builds your compiler and a front-end command `compile` which takes `FILENAME.alice` as an argument and produces runnable output in `FILENAME`. To give a concrete example, the test program may run:

```
> make
> ./compile ex01.alice
> ./ex01
```

Additional Help Getting Started

You will need to ensure that your parser generates a tree describing the structure of the expression which has been parsed. You will probably find it helpful to define a specialised abstract syntax tree class for this purpose.

You can then create a walker which will visit each node of the tree and generate assembly code. If your test cases are suitably simple you should not initially need to consider efficient register allocation. If you are using Linux and the result can fit in an 8 bit integer, you will be able to use the Linux system call 1 (exit) with the result of the expression as the exit code. This will save you from having to implement print code at this stage.

Some other general points to consider:

- If you discover that some parts of the specification from Milestone 1 or parts of the code from Milestone 2 do not fit with the requirements of this milestone, do not hesitate to apply significant modifications to your previous work.
- As before you are advised to implement iteratively, do not try to implement every feature in one session.
- As with all labs in the second year, time management is key. Do not leave everything until the final week, you will not be able to complete the work in time.

Submission

As you work, you should *add*, *commit* and *push* your changes to your Git repository. Your `labranch` repository should contain all of the source code for your program. In particular you should ensure that this includes:

- The files required to build your compiler,
- A Makefile which builds the compiler
- A command `compile` which runs your compiler.

The auto-tester will be cloning your repository for testing. To be sure that we test the correct version of your code you will need to submit a text file to CATe containing the *revision id* that you consider to be your final solution.

You should check the state of your `labranch` repository using the repository webpages at `https://www.doc.ic.ac.uk/~tora/firstyear/lab/`. If you click through to the `MAlice` exercise you'll see a list of the different versions of your work that you have pushed. Next to each version you'll see a link to download a CATe submission token. You need to download this for the version that you want to submit as your "final" version.

You should submit your chosen CATe submission token (`cate_token.txt`) and your report (`report.pdf`) as a group to CATe by 14th December 2012.

Assessment

In total there are 60 marks avalible in this milestone and 20 marks avaliable for the extensions and report. These are allocated as follows:

Code Generation: print/read	3
Code Generation: expressions	3
Code Generation: functions	3
Code Generation: procedures	3
Code Generation: loops (`while`)	3
Code Generation: conditionals (`if`)	3
Code Generation: nested functions	5
Code Generation: recursion	5
Code Generation: global declarations	2
Code Generation: pass-by-reference	5
Code Generation: nested blocks	5
Optimisations	10
Design, Style and Readability	10
Extensions	20
Report	20

This milestone will constitute 30% of the marks for the MAlice exercise and your extensions will constitute 20% of the marks for the MAlice exercise. Feedback on your work will be returned by 18th January 2013.