MAlice Compiler Report

Sarah Tattersall Peter Hamilton

December 16, 2011

1 Introduction

MAlice is an imperative language, inspired by Lewis Carroll's book 'Alice's Adventures in Wonderland'.

The syntax of MAlice is based upon events and dialogue which occur throughout the book and loosely follows English prose. The following document outlines our implementation of MAlice and why we made some of our design decisions.

2 The Product

2.1 Quality

We believe our code to be of high quality. We have focused on user readability and thus have added lots of comments so that a programmer unfamiliar with the code could understand what we have done.

2.2 Meeting the Specification

To help make sure our code met the specification we repeatedly tested it in a TDD style on the examples given for both milestone 2 and milestone 3. We also wrote our own test cases to test examples and situations not covered in the provided files. Unfortunately we were not able to meet the specification in it's entirety as implementing newlines in sentences proved incredibly difficult to code in assembly. This was because scanf doesn't automatically encode a '
n' in a string. Therefore, while the output is produced correctly, newlines are not supported. However, we successfully implemented all the other features require for MAlice and made several optimisations which improved them further.

2.3 Sound Basis For Further Development

Our code could be extended further but could be improved by some re-structuring to deal with whether a value is in a register or a memory location. (Discussed below). That aside, our code has well designed data structures for abstract syntax tree nodes and intermediate Nodes and as such it would be very easy to develop these further. The lexer and parser configurations are also well laid out and would also be easy to understand and extend.

2.4 Optimisations & Enhancements

2.4.1 Arithmetic Parentheses

We implemented the use of parenthesis around expressions which allows for more complicated mathematical expressions to be built. It allows expressions which do not obey the given operator precedences and give the programmer more flexibility within the language.

2.4.2 Removing Unnecessary Includes

To reduce the size of the assembly file where possible, we used some logical statements and flags to decide whether to include certain sections of code. These include the extern statements ‘printf’, ‘scanf’, ‘fflush’, ‘malloc’ and ‘free’. We also conditionally included the statements for formatting integers and characters and the ’.bss’ section for variable definitions.

2.4.3 Temporary Registers and Graph Colouring

To assist with register allocation we used temporary registers to work out a worst case scenario for how many individual registers we would need. We then used graph colouring to go through the temporary registers and calculate which ones could be replaced by the same real registers whilst preventing interference between registers (or memory locations in the case of overflow). This greatly reduced the number of registers needed in the generated assembly code.

To perform the calculations we worked out the live ranges for the variables used in the source code and from there worked out when and where they could share registers.

2.4.4 Smart ‘mov’ Instructions

When the intermediate code is generated, some instructions result in mov instructions going from a register to itself, to get around this we added a logical statement to our mov node which will return no instruction if it would have otherwise resulted in a pointless mov instruction.

2.4.5 Explanatory Error Messages

Although error messages do not give a performance improvement in code runtime, they dramatically improve the programmer’s exprerience with MAlice. They make debugging easier and consequently make development easier and faster meaning the programmer is likely to use the language again. We have also included run time errors for division by zero and arithmetic overflow. These all produce a helpful message.

2.4.6 Assembly Readability

Due to the number and variety of print statements normally used in a MAlice program we decided to implement print subroutines for sentences, letters and numbers. This greatly reduces the size of our assembly files and consequently improves low level code readability and debugging.

3 Design Decisions

3.1 Languages

3.1.1 Python

The first design choice we had to make was which language in which to write our compiler. In the end we chose to write it in Python. Python is quick to develop and has many useful libraries which have proved to be an invaluable asset whilst writing our compiler. We did not have to waste time writing many functions since they come pre-defined with Python and this allowed us to focus more on the design of our compiler. A downside of Python is that it can become very untidy very quickly and so we both had to be aware of this and try to comment and format the code as we went along.

3.1.2 IA-64

The next design choice was which language to use for our assembly language. We decided to write the output in IA-64. Our reasoning was that most computers nowdays are 64-bit which superseeds the 32 bit architecture. Another viable option was Java Bytecode but the reason we decided not to use this was because it's performance is not as good as IA-64 and we felt it would be a good learning experience for us. Java Bytecode relies entirely on the stack whereas IA-64 can make use of registers which speeds up the run-time. We next needed to find a lexer and a parser for our compiler. We decided to use a module called PLY because it does both lexing and parsing, making the implementation much easier. Whilst PLY was a very useful tool, it was lacking some features specific to MAlice so we added the ability to keep track of clause numbers to the module. The only downside to using PLY is that compilers are not written in Python very often and so there was little documentation for it.

3.2 Semantic Analysis

In our code we had to make several design decisions along the way. When parsing the code we decided to build an abstract syntax tree made of nodes which we created (ASTNodes.py). For milestone2 we performed semantic analysis by parsing the tree in a static function. However this lead to a very long and unreadable function with one very large 'IF' statement. For milestone 3 we decided that each node would contain it's own check and translate methods which lead to the code being far more readable and maintainable.

3.3 Register Overflow

This was a particularly difficult issue owing to the fact that pushing and popping from the stack at the right time to ensure there are no register conflicts and that all variables are available at the right time is not a trivial task.

Therefore the way we dealt with register overflow was that any registers which would overflow were put into memory. We then added these memory addresses to the register map so they could be used in place of normal registers, whilst keeping our code for translating intermediate nodes the same.

Whilst performing operations, we had to be slightly careful with memory addresses and registers.For example, in the case of mov you cannot move an immediate value into a memory address, you have to move it into a register first. As a result, generating the code for the intermediate nodes also takes this into account.

3.4 What we'd do differently

If we were to start the project again, there are several things we would do differently to improve the maintainability and performance of our code.

Although we discussed how to implement our code for milestone 3 we probably didn't think enough about all potential future requirements. The biggest problem we encountered during milestone 3 was that our ASTNodes assumed everything would be in a register, as was the case for milestone 2 with it's primative types. When doing it again we would use our knowledge of other programming languages to brainstorm all possible features which might be implemented in the future. Had we done this more extensively when writing milestone 2 our code might have been a lot easier to extend in milestone 3.

4 Extensions Beyond The Specification

4.1 Loop invariants

Unfortunately we ran out of time to implement moving loop invariants outside of the loops. If we had more time we would definitely have done this since it would make our compiled code run a lot faster.

4.2 Classes

A very interesting extension would be to add classes to MAlice. A possible way to implement them could be in a similar fashion to functions. For example 'Alice held a tea-party for ClassName with (TYPE ID, ...)' could be used to represent a class constructor ClassName(Type ID, ...).