MAlice Evaluation, Reflection, Extensions

Suhaib Sarmad, Alex Appetiti, Jamal Khan

December 18, 2010

The Product

Our compiler met all the specifications as it implemented all that was required by milestone 3. In addition we also made sure that there were no invalid type casts and invalid operator usage such as adding a string and number. Our function parameter and return type checking are also strict in this respect for both normal and lambda functions. We also managed to output the line numbers in which there were syntactic and semantic errors. We have also catered for array out of bound exceptions to prevent programs from reading parts of memory that it should not be reading.

Our compiler is flexible and has a solid grounding to be used for further development. For example if we had a change of syntax, we could adjust the BNF Grammar, and have GOLD generate another engine file. This would only require a small rewrite of the semantic analyser, so that our control flow graph can be fed to the code generator without any problems.

In milestone 2, our compiler created a proper control flow graph which was then analyzed and optimised (by use of a graph colouring algorithm) the code before generating Intel x86 (NASM) Assembly Code. Unfortunately in milestone 3 due to the extensive changes and some unfortunate problems we ran into, we did not have the time to re-implement this system and had to stick with generating ANSI C code and using the GCC compiler to then create an executable binary file.

The advantage of this, is that we can now adjust the code to compile for any system just by changing the GCC command making the generated code somewhat more portable than it's previous counterpart. However some efficiency may be lost in the process of translating it into C first rather than outputting assembly to start off with; an example of this can be seen with the lambda functions; C doesn't support them natively and therefore our current implementation is less efficient than had we directly outputted assembly code.

The Design Choices

Our MAlice compiler was written in C, this was because there were many lexing and parsing tools available and we thought it would be an interesting language to implement it in. We ended up using a tool called GOLD, which uses the LALR algorithm to analyse the syntax and generate an Abstract Syntax Tree that is represented as a C struct.

We believe that it was important to use tools for the lexing and parsing for this exercise as it allowed us to concentrate on the more interesting aspects of this exercise, code generation and semantic analysis, as opposed to rewriting our own parser, which can be a very lengthy and cumbersome process. We quickly found that there are many very good lexers and parsers available.

As mentioned above, we used the GOLD parser. We chose this over the other parsers available (to name a few: ANTLR, Grammatica, Spirit, YACC...) because it was incredibly well documented, it had Integrated Grammar Testing (which allowed us to test and single out errors in our grammar on the fly) and many other features which made it less tedious to use than its counterparts.

If we were to redo our compiler, instead of outputting to C/NASM code, we would output intermediate LLVM assembly so that we could have LLVM generate the binary for us. This would make it so that we could easily have our compiler target a different architecture, also, LLVM would do all optimisations related to the assembly code (e.g. register allocation) for us, and so the outputted code would be heavily optimised without us having to worry about it, hence letting us concentrate more on the other parts of the compiler instead.

Another design choice we would make is that we would have chosen a higher level language (possibly functional or object-oriented) to write the compiler; C is incredibly fast and is compatible with many of the parser-generators out there, but due to the many low-level concepts we had to deal with (i.e. dynamic memory allocation), it became very hard to manage after a while; it was frustrating having to debug memory-related errors rather than being able to purely concentrate on the logic of the compiler itself.

Beyond the Specification

At the moment MAlice is an imperative programming language. A major improvement would be to make it more object oriented by adding features such as classes, inheritance, interfaces and polymorphism. An advantage of this approach is that we can begin to create data structures making the language more powerful.

The GOLD engine also supports Unicode within the grammar which we didn't allow for within our program, perhaps enabling this feature within the language would make it so that programmers who do not use the latin alphabet could easily implement programming solutions in their own languages.

Another interesting development would be adding libraries of functions (an example is the Math library in Java) to MAlice. This would save lots of time as programmers would not have to rewrite common functions repeatedly. Seeing as the language is (arguably) closer to spoken language than most mainstream languages seen today, I can see the language used as a means to teach programming to beginner programmers of a younger age. To this end, it would be interesting to see libraries which make it easy to output images, text, movie clips and audio easily (or take input from a touchscreen) so that simple games could be created.