Second Year Compilers Exercise: The MAlice Compiler

Milestone 2

Summary

In the previous exercice you were asked to devise a specification for the MAlice language from code samples. These code samples were deliberately insufficient to precisely define the language syntax and semantics in order for you to devise and justify sound assumptions for the language.

In this exercise you will be implementing a full compiler for the subset of the MAlice language you have seen so far. To that end, you have been given additional code samples and access to the reference implementation of the MAlice language. Note that in this exercice, you are asked to implement the semantics of the reference implementation.

This exercise aims to give you experience with the tool-chain you will use to implement the full MAlice compiler in Milestone 3.

You are free to choose the tools you will use, or to handcraft your own if you feel none are appropriate. You will need to provide a lexer, parser and semantic analyser for the front-end of the compiler and a code generator for the back-end.

You are welcome to write the compiler for an architecture of your choosing. You should consider the costs and benefits of each architecture along with your knowledge of the architecture when making your choice.

Details

As you will no doubt remember from the lectures, a compiler takes a source input file, does something magic to it, and produces an output file in the target language. In this exercise you will be performing the magic.

Breaking the process into stages the compiler must;

- Perform lexical analysis, splitting the input file into tokens.

- Perform Syntactic Analysis, parsing the tokens and creating a representation of the structure of the input file.
- Perform Semantic Analysis, working out and ensuring the integrity of the meaning of the input file.
- Synthesize output in the target language, maintaining the semantic meaning of the input file.

You are welcome to use tools for each stage. You are encouraged to look online for ideas of what tools are available, and to discuss the merits of each tool before finalising your choice.

Important Changes

The `Alice found ...` return statement from milestone 1 has been removed from the language specification. It has been replaced by the "spoke" statement for outputing numbers. Code samples have been updated to reflect the change.

Compiler Reference Implementation

You now have access to a reference implementation of the MAlice language. Your implementation should mimic the behavior of that implementation. You can run valid MAlice program as follows:

```
MAlice program.alice
```

Note, that the file `program.alice` must be readable by *all* (`chmod a+r program.alice`)

Submit by: November 21st

What To Do

- Write a compiler for the MAlice programming language subset provided.

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 addition.alice
./addition
```

Additional Help Getting Started

Start by choosing an implementation language. Unless you are especially emotionally opposed to Java, it is recommended that you use it. If you choose another implementation language, you must ensure that your compiler can be built and run on the lab machines.

Having chosen the implementation language decide which tools, if any, you plan on using. Remember that some lexer generation tools will have better interfaces with parser generation tools than others and that this can greatly affect the difficulty of your task.

Get used to your lexer generation tool. Try generating a lexer that can match arithmetic expressions made up of only numbers and the symbols '+' and '-'. If you have your lexer print the items it is matching you can feel confident it is working as you expect. Create some simple test files.

Now interface with your parser generation tool. Depending on the tools you have chosen this can be done in a multitude of ways. Continue to work with the simple test cases you created. Try to evaluate the results of the expressions using the features of your parser generator. This will not necessarily be the best way to evaluate expressions in your final compiler, but will give you experience working with the parser.

Change your parser generator to instead return a tree describing the structure of the expression which has been parsed. 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 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.

You are welcome to use assemblers and linkers as tools if neccessary.

Assessment

This milestone will constitute 30% of the marks for the exercise.

You should provide a `tar` archive containing:

- The files required to build your compiler,
- A Makefile which builds the compiler

- A command `'compile'` which runs your compiler.