The MAlice Compiler: Milestone 2

Second Year Computing Laboratory
Department of Computing
Imperial College London

Summary

In the previous exercise you were asked to devise a specification for a subset of the MAlice language from a small set of code samples. These samples were deliberately insufficient to precisely define the language syntax and semantics in order to force you to devise and justify sounds assumptions for the language.

In this exercise you will be implementing the front-end of a compiler for the full MAlice language. That is, you will be providing a lexer, a parser and a semantics analyser for your MAlice compiler. To this end you will be provided with additional code samples and access to a reference implementation of the MAlice compiler. Note that in this exercise you are asked to implement the semantics of the reference implementation, and not those you gave in Milestone 1 (although you are free to target an implementation of your choosing, which may affect certain aspects such as overflow and underflow).

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.

Details

As you will recall 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:

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 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.

For this milestone you need to work on the front-end of your compiler (the first 3 stages given above). Your compiler front-end must be able to successfully parse any valid MAlice program, generating some internal representation of that program. It must also be able to detect an invalid MAlice program and generate appropriate error messages. In particular, when your compiler fails to parse an input file it should report the nature of the error.

You will once again have to discern what constitutes a valid program from the samples given. You may have to reconsider the assumptions you made as to correct behaviour during the previous milestone. Think about the semantics of each new construct that you encounter in the provided examples.

Compiler Reference Implementation

You now have access to a reference implementation of the MAlice compiler via the command `MAlice`. Your implementation should mimic the behaviour of this implementation (note that this will not necessarily be the same as your specification from Milestone 1) although you are not required to work to the same precision, this will be determined by your target language. To see a list of the compiler's options run:

```
MAlice -h
```

The reference compiler parses each input file and, if this was successful, generates a form of three-address code for each input file.[1]. It outputs this into a file with the suffix `.3code`. For example, running:

```
MAlice ex01.alice
```

produces an output file `ex01.3code`. The compiler can be called on a list of files, for example:

```
MAlice ex01.alice ex02.alice ex03.alice
```

You can also get the compiler to systematically walk through a directory and compile each MAlice program file (`*.alice`) it encounters, by using the `-D` flag. For example, try running:

```
MAlice -D malice_examples
```

You can get the reference compiler to simulate the execution of the generated three-address code using the `--exe` option. For example, running:

```
MAlice -D malice_examples --exe
```

will simulate the generated code for each program that gets though the compiler's front-end. You may find it useful to pipe this output to a file. For example, running:

```
MAlice -D malice_examples --exe > output
```

on the `malice_examples` directory provided for Milestone 1 will generate the file `output` found in that directory.

One other useful feature of the reference compiler is that it can generate graphs (in uDrawGraph format) of the internal structures produced when parsing an input file. To get the reference compiler to generate these graphs, call it with the `-u` flag. For example, running:

```
MAlice -u ex01.alice
```

will generate the graphs for the abstract syntax tree (`ex01.ast.udraw`), the symbol table (`ex01.stable.udraw`), the call graph (`ex01.callg.udraw`), and control flow graphs for each subprogram (e.g. `ex01.hatta.cfg.udraw`) generated during the compilation of MAlice program `ex01.alice`. You may find these visualisations helpful when developing your MAlice compiler.

These graphs can be viewed with the `uDrawGraph` command. For example, running the command:

```
uDrawGraph ex01.ast.udraw
```

will display the abstract syntax tree generated for `ex01.alice` above.

Finally, you can get the reference compiler to clean up the files it has generated by using the `-C` flag. That is, running:

```
MAlice -c malice_examples
```

will remove all of the generated three-address code (`*.3code`) and graphs (`*.udraw`) from the `malice_examples` directory.

[1]Note that the three-address code generated by the reference compiler is not considered an appropriate target language for your MAlice compiler. In the next milestone you should be generating assembly code for a *real* machine, e.g. x86, or an *actual* virtual machine, e.g. LLVM bitcode.

Submit by: 26th November 2012

What To Do:

1. Obtain the extra example MAlice programs from the `malice_examples` repository. You can do this either be re-cloning the repository into you local area, or by using Git's `fetch` command.
2. Write the front-end of a compiler for the full MAlice programming language. Your compiler front end will need to perform lexical analysis, syntactic analysis and semantics analysis on an input file. You may find it useful to extend the specification of the language you gave in Milestone 1, or to define a brand new specification.

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

`> make`	to build your compiler
`> ./compile FILENAME.alice`	to parse FILENAME.alice

You should therefore provide a Makefile which builds your compiler and a front-end command `compile` which takes `FILENAME.alice` as an argument and runs it through your compiler's front-end processes either successfully parsing the file or generating error messages as appropriate. To give a concrete example, the test program may run:

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

and expect to successfully parse the input file. It may also run:

```
> make
> ./compile ex02.alice
```

and expect to see a type error for the binary operator `+` (you can't add a number to a letter).

Additional Help Getting Started

By looking at the examples provided you should recognise some very well-known constructs which you have encountered in previous courses and in other programming languages. Whenever you are giving semantics to the MAlice constructs, think about the semantics of similar constructs, how you use them in your programs, and how you expect them to behave. In particular:

- Consider the process you went through during Milestone 1.
- Think carefully about design - poorly thought out design will slow your development and make debugging harder.
- Implement iteratively, do not try to implement every feature in one session.
- Manage your time carefully. Do not leave everything until the final week.

You should start by choosing an implementation language. You are free to use Java, C, C++, Haskell or Python. Unless you are especially emotionally opposed to Java, it is the recommended language. If you would like to use another language, you must check with one of the lab organisers first. It is up to you to ensure that your compiler can be built and run on the lab machines.

Having chosen the implementation language you should 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. You are welcome to use assemblers and linkers as tools if necessary.

Next, you should 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. Be sure to create some simple test files at this point.

Now, you should be able to interface your lexer with your parser generation tool. Depending on the tools you have chosen this can be done in a multitude of ways. At first you should continue to work with simple arithmetic expression test cases. Once you can fully process your simple tests you can then extend your compiler to handle the full MAlice input language.

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.

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`) as a group to CATe by 26th November 2012.

Assessment

In total there are 50 marks available in this milestone. These are allocated as follows:

Lexical Analysis	5
Syntactic Analysis (Parsing)	15
Semantic Analysis	20
Design, Style and Readability	10

This milestone will constitute 35% of the marks for the MAlice exercise. Feedback on your work will be returned by 10th December 2012.