Inf2A 2015–16: Assignment 1

The Language Processing Pipeline for Micro-Haskell

Issued 20 October 2015

The deadline for this assignment is **4pm, Tuesday 3 November 2015**.

Overview

The objective of this practical is to illustrate the language processing pipeline in the case of a particular formal language: a rudimentary programming language, which we call *Micro-Haskell* (or MH for short) since it is a small (but interesting) fragment of the Haskell programming language. Although the practical is self contained, you will find it helpful to refer to the introduction to Micro-Haskell given in Lecture 12. Prior knowledge of Haskell itself is *not* required for this assignment. However, students without previous experience with Haskell may need to invest more time in understanding the practical.

The practical illustrates all stages of the language processing pipeline for programming languages, taking us from a source program, written in MH, to execution of the program. In our case, the pipeline has four stages: lexing, parsing, type-checking and evaluation (executing the program). Your task is to provide language-specific material to assist with the first three cases: *lexing*, covered in Part A of the practical; *parsing*, covered in Part B; and *type-checking*, covered in Part C.

The code implementing the lexer, parser, type-checker and evaluator will all be implemented in Java. (So you are warned at the outset that you will need to think in two languages at once!) Several general-purpose components are provided for you. Your task is to supply the parts specific to MH, using your understanding of the course material. Once you have finished, you can link up your software to a simple evaluator (which is provided for you) to obtain a complete working implementation of MH.

Of course, many libraries of lexing and parsing tools in Java are available on the Internet, but the point of this practical is to build things up from first principles in order to understand how they work. You should therefore not attempt to use any lexer or parser utilities you may find online, nor any tools such as `StringTokenizer` or `StreamTokenizer` that might otherwise appear tempting.

Instructions

To begin, download the code file `Inf2A_Prac1_Files.zip` from the Informatics 2A Assignments webpage. On a DICE machine, this can be unpacked using

```
unzip Inf2A_Prac1_Files.zip
```

This will build a subdirectory `Inf2A_Prac1_Files` inside your working directory (folder), within which you will find all source files needed for the assignment. Look first at the file `MH_example.txt`, which provides a sample of Micro-Haskell.

The assignment comprises four parts: A–D. In each of Parts A–C, you have to complete a template Java file that is provided for you. In doing this, fill in only the parts of code you are instructed to write. Except when explicitly told to do so, **do not make any alterations to the code that has been written for you!** You are required to submit your solutions **from a DICE machine**, using the `submit` command, as specified below.

Part	Submit command	Marks
A	`submit inf2a 1 MH_Lexer.java`	35
B	`submit inf2a 1 MH_Parser.java`	30
C	`submit inf2a 1 MH_Typechecker.java`	25
D	no further submission required	10

It is also possible to submit your solutions to parts A–C simultaneously:

```
submit inf2a 1 MH_Lexer.java MH_Parser.java MH_Typechecker.java
```

Part D combines Parts A–C into a full working implementation of MH, and does not require any submission of its own.

Part A: Lexing in Micro-Haskell [35 marks]

In this part, we construct a lexical analyser for MH. A general-purpose longest-match lexer is already provided. Your task is to supply deterministic finite-state machines that serve as recognizers for the various classes of lexical tokens in the language.

Look at the provided file `GenLexer.java`. It begins with some Java functions that define certain useful classes of characters: *letter*, *small*, *large*, *digit*, *symbolic*, *whitespace*, *newline*. Next comes a Java interface `DFA` which defines the functionality that any finite state machine has to provide. Some of this is provided in the class `GenAcceptor` which

follows, but notice that this class contains stubs for five 'abstract' methods whose implementation will be specific to the particular DFA in question. There then follow three examples of how to construct implementations of particular DFAs: `EvenLetterAcceptor`, `AndAcceptor` and `SpaceAcceptor`.

Notice that states are represented by integers, with 0 as the initial state. Besides the transition operation and the set of accepting states, our DFAs here must also be equipped with a set of one or more *dead* states: that is, non-accepting states from which no sequence of transitions can lead to an accepting state. Note also that our DFAs implement the method `String lexClass()`, which provides the name of the lexical class they are associated with. This is done because we wish our lexer to output a stream of tokens each tagged with their lexical class.

Your objective in Part A is to implement DFAs in the same style corresponding to the lexical classes of Micro-Haskell. This is to be done in the file `MH_Lexer.java`, which currently provides a template containing some gaps for you to fill in. For the first six of these gaps, follow the pattern of the examples in `GenLexer.java` to construct DFAs for the following lexical classes defined by regular expressions (these correspond closely to lexical classes of actual Haskell).

- A class `VAR` of *variables*, defined by

$$small\ (small + large + digit + \text{'})^*$$

- A class `NUM` of *numeric literals*, defined by

$$digit\ digit^*$$

- A class `BOOLEAN` of *boolean literals*, defined by

$$\texttt{True} + \texttt{False}$$

- A class `SYM` of *symbolic tokens*, defined by

$$symbolic\ symbolic^*$$

- A class of *whitespace elements*, defined by

$$whitespace\ whitespace^*$$

- A class of *comments*, defined by

$$\texttt{---}^* \; (\mathit{nonSymbolNewline}\ \mathit{nonNewline}^* \ + \ \varepsilon)$$

  where *nonSymbolNewline* is the set of all characters except those of *symbol* or *newline*, and *nonNewline* is the set of all characters except those of *newline*. Note that `---`* effectively means 'two or more dashes'.

The names of the last two classes, implemented by the `lexClass()` method, should both be the empty string. This will notify the lexer that tokens of these classes should be discarded.

In addition to these classes, keywords such as `if` and special symbols such as `;` will require 'singleton' (i.e. one-element) lexical classes all to themselves. For this purpose, we provide an operation `tokAcceptor` which yields a DFA that accepts a specified string (called `tok`) and nothing else. Fill in the gap in the code for this operation to provide such a DFA. Here the name of the lexical class should be identical to the string itself — this will serve to make the specific token we are dealing with visible to the parser.

The lexical classes we require for MH are now the six lexical classes listed above, together with singleton classes for the five keywords `Integer`, `Bool`, `if`, `then`, `else` and for the three special symbols `(`, `)`, `;`.

Following the example of class `DemoLexer` in the file `GenLexer.java`, add a few lines of code to construct acceptors for these fourteen classes, and put these together in an array called `MH_acceptors`. The acceptors should be listed in order of priority, with highest priority first, which should be sensibly chosen so that keywords like `if` are assigned an appropriate lexical class. To obtain full marks, your chosen priorities should emulate the lexing behaviour of the Glasgow Haskell Compiler (ghc).

The `MH_acceptors` array is fed to a general-purpose routine that performs longest-match lexing (also known as *maximal munch*) using the method described in Lecture 6. Take a brief look at the code for this in `GenLexer.java`, and check that you broadly understand what it is doing.

You should now be able to compile `GenLexer.java` and your file `MH_Lexer.java` to create a lexer for MH. To test your lexer, you might wish to adapt the `LexerDemo` code in `GenLexer.java`; this will allow you to create a simple command-line driven lexical analyser for MH. You are not required to submit this test code, however.

Before we leave the subject of lexing, take a quick glance at the code provided in `CheckedSymbolLexer.java`. This performs some mild post-processing on the stream of lexical tokens: symbolic tokens are checked to ensure that they are among the tokens that feature in MH:

```
::   ->   =   ==   <=   +   -
```

If they are, then the lexical classname `SYM` is replaced with the token itself, just as for keywords and `(`, `)`, `;`.

Submission: Submit your answer to part A, **from a DICE machine**, using the command: `submit inf2a 1 MH_Lexer.java`

Part B: An LL(1) parser for Micro-Haskell [30 marks]

Take a look at the provided file `GenParser.java`. This begins with an interface `TREE` and a class `STree` for representing syntax trees (for any context-free grammar). The class `GenParser` then provides an implementation of the general LL(1) parsing algorithm as described in lectures (again, check that you broadly understand it).

In order to complete this and obtain a working parser, some grammar-specific ingredients must be provided: a parse table and a choice of start symbol. The class `EvenAndParser` gives a simple example of how to do this, for an artificial language that uses the lexical classes defined in `GenLexer.java`. Note in particular the convention that the names of nonterminals are identified by adding the symbol `#` (we can get away with this because `#` doesn't itself feature in any lexical tokens of MH). You can try out this parser on the sample input file `EvenAnd_example.txt`, by compiling `GenParser.java`[1] and then typing

```
java ParserDemo EvenAnd_example.txt
```

Your task is to implement a similar working parser for the language MH, in the file `MH_Parser.java` (which is discussed below), following the pattern of `EvenAndParser`.

Now we consider the grammar of MH itself. The terminal symbols are the names of lexical classes in tokens output by `CheckedSymbolLexer`. The complete list of these is as follows:

```
VAR   NUM   BOOLEAN   Integer   Bool   if   then   else
(   )   ;   ::   ->   =   ==   <=   +   -
```

The start symbol of the grammar is *Prog*, and the productions are as follows. (For legibility, we omit the `#` symbol, instead distinguishing nonterminals by choice of font.)

The full grammar is presented in Figure 1. If this looks daunting at first, the following observations may be helpful:

[1] Don't worry if you get a warning about unchecked operations when you do this — it will work anyway.

$$
\begin{array}{rcl}
\mathit{Prog} & \rightarrow & \epsilon \mid \mathit{Decl}\ \mathit{Prog} \\
\mathit{Decl} & \rightarrow & \mathit{TypeDecl}\ \mathit{TermDecl} \\
\mathit{TypeDecl} & \rightarrow & \texttt{VAR}\ \texttt{::}\ \mathit{Type}\ \texttt{;} \\
\mathit{Type} & \rightarrow & \mathit{Type1}\ \mathit{TypeOps} \\
\mathit{TypeOps} & \rightarrow & \epsilon \mid \texttt{->}\ \mathit{Type} \\
\mathit{Type1} & \rightarrow & \texttt{Integer} \mid \texttt{Bool} \mid \texttt{(}\ \mathit{Type}\ \texttt{)} \\
\mathit{TermDecl} & \rightarrow & \texttt{VAR}\ \mathit{Args}\ \texttt{=}\ \mathit{Exp}\ \texttt{;} \\
\mathit{Args} & \rightarrow & \epsilon \mid \texttt{VAR}\ \mathit{Args} \\
\mathit{Exp} & \rightarrow & \mathit{Exp1} \mid \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
\mathit{Exp1} & \rightarrow & \mathit{Exp2}\ \mathit{Op1} \\
\mathit{Op1} & \rightarrow & \epsilon \mid \texttt{==}\ \mathit{Exp2} \mid \texttt{<=}\ \mathit{Exp2} \\
\mathit{Exp2} & \rightarrow & \mathit{Exp3}\ \mathit{Ops2} \\
\mathit{Ops2} & \rightarrow & \epsilon \mid \texttt{+}\ \mathit{Exp3}\ \mathit{Ops2} \mid \texttt{-}\ \mathit{Exp3}\ \mathit{Ops2} \\
\mathit{Exp3} & \rightarrow & \mathit{Exp4}\ \mathit{Ops3} \\
\mathit{Ops3} & \rightarrow & \epsilon \mid \mathit{Exp4}\ \mathit{Ops3} \\
\mathit{Exp4} & \rightarrow & \texttt{VAR} \mid \texttt{NUM} \mid \texttt{BOOLEAN} \mid \texttt{(}\ \mathit{Exp}\ \texttt{)}
\end{array}
$$

Figure 1: Grammar for Micro-Haskell

- The grammar for types (i.e. the rules for *Type*, *TypeOps*, *Type1*) is a self-contained sub-grammar that can be understood in isolation; see Lecture 13.
- The grammar for expressions (the rules for all nonterminals from *Exp* onwards) is another self-contained sub-grammar, and is broadly similar in its workings to the LL(1) grammar for arithmetic expressions from Lecture 12. Note that the productions for *Exp3* and *Ops3* are intended to cater for multiple function applications, such as `f x y`.
- It may be helpful to look at the example in `MH_example.txt` in conjunction with the rules above.

Once you feel you have assimilated the grammar, find yourself a large sheet of paper and work out the complete LL(1) parse table. (Most of the entries will be blank, so don't panic!) You may find that some calculations of First and Follow sets help you to do this; however, you will not be required to submit these calculations or the written-out parse table you construct.

Now open the file `MH_Parser.java`. You will see that the right hand sides of all the grammar rules have already been declared for your convenience, so all you have to do is to supply an implementation of the parse table itself in the style of `EvenAndParser`. You may make use of auxiliary definitions and other reasonable devices to reduce the amount of code you need to write, provided that your code remains clearly readable and its correspondence to the parse table you have drawn up remains transparent.

After completing and compiling this, you will now be able to try out your parser on the sample source file provided:

```
java MH_ParserDemo MH_example.txt
```

If this reports successful parsing, it's certainly an encouraging sign that your parser is largely correct and will obtain a reasonable mark. However, to ensure your parser is *completely* correct, you will have to do some further testing, since (a) there are possible parsing scenarios not represented by this small example, and (b) you also need to ensure that your parser rejects *incorrect* programs and that the error report it produces is plausible.

Submission: Submit your answer to part B, **from a DICE machine**, using the command: `submit inf2a 1 MH_Parser.java`

Part C: Typechecking for Micro-Haskell [25 marks]

In this section, you will implement critical parts of a typechecker for MH.

The LL(1) grammar we have been using serves to disambiguate inputs and make them readily parseable; but once these issues have been got out of the way, it is much more convenient to work with simpler trees known as *abstract syntax trees (ASTs)* in which extraneous detail has been stripped away. For example, as in Lecture 13, types in MH are conceptually just trees for the grammar:

$$\textit{Type} \;\rightarrow\; \texttt{Integer} \mid \texttt{Bool} \mid \textit{Type}\ \texttt{->}\ \textit{Type}$$

Look at the file `Types.java`, which defines a Java representation of MH types in this stripped-down form. The interface `MH_TYPE` declares various operations one can perform on such types (check that you understand what they are intended to do), while further down, the class `MH_Type_Impl` provides predefined constants for the MH types `Integer` and `Bool`, as well as a constructor for building an arrow type from two previously existing MH types. In the typechecking code you will be writing, these may be utilized as follows:

```
MH_Type_Impl.IntegerType ;  // AST for Integer
MH_Type_Impl.BoolType;      // AST for Bool
new MH_Type_Impl (t1,t2);   // AST for (t1->t2)
```

Clearly, we will need a way to convert syntax trees as produced by the parser into ASTs of this kind. This is done by the provided methods `convertType` and `convertType1` in `Types.java`. A good warm-up to your own task would be to try and understand the workings of `convertType` and `convertType1` with the help of the comments provided.

A similar notion of abstract syntax trees is also required for *expressions* (trees with topmost label `#Exp`). In effect, ASTs for expressions are just trees for the simplified grammar:

$$\textit{Exp} \;\rightarrow\; \texttt{VAR} \mid \texttt{NUM} \mid \texttt{BOOLEAN} \mid \textit{Exp}\ \textit{Exp} \mid \textit{Exp}\ \textit{infix}\ \textit{Exp} \mid \texttt{if}\ \textit{Exp}\ \texttt{then}\ \textit{Exp}\ \texttt{else}\ \textit{Exp}$$

where *infix* ranges over `==`, `<=`, `+`, `-`. Look in the file `Expressions.java` at the interface `MH_EXP`, which declares various operations that can be performed on such trees. The intended meanings of these operations are all you need to understand from this file (and you can ignore `isLAMBDA`). Their workings are further explained by commented examples in the file `Expressions.java` immediately below the `MH_EXP` interface. You don't need

to get to grips with the class `MH_Exp_Impl`, which contains (among other things) some code for converting trees returned by the parser into ASTs for expressions.

Assuming the conversions to ASTs have already been done, your task is to write a typechecker for ASTs, by completing the body of the method `computeType` in the file `MH_Typechecker.java`. More precisely, your code should compute the MH type of an expression given as an AST `exp` of Java type `MH_EXP`, returning the result as an AST of Java type `MH_TYPE`. If the expression is not correctly typed, your code should flag up a type error, which you can do by means of the command:

```
throw new TypeError ("blah blah") ;
```

Each time such a command appears, the string should provide a brief description of the nature of the type error in question. Such error messages should be designed to be helpful to MH programmers who need to identify and correct type errors in their programs.

There is one other important ingredient to be explained. The type of an expression such as `if x then y else z`, or even whether it is well-typed at all, will depend on the types ascribed to the variables `x`, `y`, `z`. In general, then, an expression `exp` will have to be typechecked relative to a *type environment* which maps certain variables to certain types associated with them. This is the purpose of `env`, the second argument to `computeType`. You may access the type associated with the variable `x`, for instance, by calling `env.typeOf("x")`.

The definition of the type of an expression (if it has one) is given *compositionally*: that is, it is computed from the types of its subexpressions. This will be reflected in the recursive nature of your implementation of `computeType`: it should compute the type of any subexpressions in order to obtain the type of the whole expression. Here are some hints on how this should work:

1. The types of `NUM`s and `BOOLEAN`s are what you think they are.

2. You should assume that each of the infix operations accepts only integer arguments; however, the type of the resulting expression will depend on the infix operation in question.

3. In an application expression $e_1\ e_2$, the type of e_2 should match the argument type expected by e_1, and you should think about what the type of the whole expression will be.

4. An expression `if` e_1 `then` e_2 `else` e_3 may in principle have any type; however, you should consider what the types of e_1, e_2, e_3 respectively will need to be in order for the whole expression to have type t.

A final hint on the Java side. To test whether two given type ASTs are equal, you should use the `equals` method from the interface `MH_TYPE`, *not* the `==` operator.

As a rough guideline, the model implementation of `computeType` consists of about 40 lines of Java code.

When you have finished, compile the files `Types.java`, `Expressions.java` and `MH_Typechecker.java` (in that order), and try executing

```
java MH_Typechecker MH_example.txt
```

Once your typechecker works, this will report that the parse, type conversion and typecheck have all been successful. To see what it is doing, look again at `MH_example.txt`. The system is using your code to check that the right hand side of each function definition has the expected type relative to an environment that can be inferred from the types specified in the MH code. (All this is managed by the remaining code in the file `MH_Typechecker.java`.) You should also try out your typechecker on other MH programs — including some that contain type errors to check that your code catches them correctly and supplies appropriate error messages.

Submission: Submit your answer to part C, **from a DICE machine**, using the command: `submit inf2a 1 MH_Typechecker.java`

Part D: Execution of Micro-Haskell [10 marks]

This part of the practical puts all the pieces together to obtain a full working implementation of Micro-Haskell.

The file `Evaluator.java` contains an evaluator for Micro-Haskell. It uses the other parts of the practical to lex, parse and type-check a program, after which, the resulting abstract syntax tree for the program is executed using a small-step operational semantics (as will be covered in Lecture 27). This results in an implementation that is *very* slow compared with a real-world implementation of Haskell. But its purpose is to illustrate how the basic principles of language processing feed into the construction of a compiler/interpreter/evaluator. Indeed, the source code for `Evaluator.java` is very concise, because the bulk of the work in processing a source program has already been done at earlier stages of the language processing pipeline. [2]

[2] You are encouraged to take a look at the evaluator source code to get a rough idea of how it works. If you are interested in how to produce a more efficient implementation, then take the UG3 Informatics course on Compiling Techniques next year!

To use the evaluator, once you have completed the rest of the practical, compile `Evaluator.java` and then run it on the source file of your choice, e.g.:

```
java Evaluator MH_example.txt
```

This will load and typecheck the MH program, and display a prompt `MH>`. Type in an expression you would like to evaluate, and hit return. The expression may involve the functions declared in your MH program. Do this as many times as you like; e.g.:

```
MH>  3+5
...
MH>  fib 6
...
MH>  gcd 104 (fib 12)
...
```

To quit, hit `CTRL-c`.

This last part of the assignment does not require you to do any further coding. Your solutions to Parts A–C will be combined with the evaluator, and the resulting complete implementation of MH will be marked for the correctness of its behaviour on a test suite of five MH programs (different from those in `MH_example.txt`. However, in order to gain assurance that your solutions to A–C will indeed combine to yield a correct implementation of Micro-Haskell, you should spent a little time testing your complete system on some small MH programs of your own devising.