Inf2A: Assignment 2

A Lexer, Parser and Typechecker for Micro-Haskell

John Longley

Issued 14 November 2011

The deadline for this assignment is **12 noon, Monday 28 November**. Please read through the whole of this handout before you start.

In this assignment you will construct a lexer, parser and typechecker for a small fragment of Haskell, which we will call *Micro-Haskell* or MH for short.[1] The lexer, parser and typechecker themselves will 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 the language in question using your understanding of the course material. Once you have finished, it should be possible to link up your software to a simple evaluator (which will be provided) 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.

To begin, download the code file `Inf2A_Prac2_Files.tar` from the Informatics 2A Assignments webpage, and unpack it using

```
tar -xf Inf2A_Prac2_Files.tar
```

Look first at the file `MH_example.txt`, which provides a sample of the fragment of Haskell we will be addressing. Conceptually, the Haskell *types* we shall be dealing with are just those given by the grammar

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

although as we shall see below, our official grammar for MH will employ a disambiguated 'LL(1)' version of this.

[1] In theory, no prior knowledge of Haskell is required for this assignment. In practice, some acquaintance with the basics may be helpful, particularly for Part C.

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 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*, *nonzeroDigit*, *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`. Your job is to produce DFAs in the same style corresponding to the lexical classes of micro-Haskell.

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 provide the name of the lexical class they are associated with: this is because we wish our lexer to output a stream of tokens each tagged with their lexical class.

Now open the file `MH_Lexer.java`. This 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

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

- A class `NUM` of *numeric literals*, defined by[2]

$$\texttt{0} + nonzeroDigit\ digit^*$$

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

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

[2]Actually, Haskell allows numeric literals with leading zeros, but our more restrictive definition here is slightly more interesting as an exercise. This means that some programs will have different meanings in Haskell and in Micro-Haskell (e.g. `01` is treated as a single token in Haskell and as two tokens in MH), but never mind.

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

  $$\textit{symbolic}\ \textit{symbolic}^*$$

- A class of *whitespace elements*, defined by

  $$\textit{whitespace}\ \textit{whitespace}^*$$

- A class of *comments*, defined by

  $$\texttt{---}^*\ \textit{nonSymbolNewline}\ \textit{nonNewline}^*\ \textit{newline}$$

  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 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 a 'singleton' lexical class 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, which should be sensibly chosen so that keywords like `if` are assigned an appropriate lexical class.

This array is then fed to a general-purpose routine that performs longest-match lexing (also known as *maximal munch*) using the method described in the lectures. We also equip the lexer with a buffer of size 1, allowing us to look ahead to the next lexical token without removing it from the output stream. Take a brief look at the code for all 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 `(`, `)`, `;`. (To ponder: why did we do it this way, rather than simply building `tokAcceptor`s for the eight symbolic tokens above?)

Part B: An LL(1) parser for Micro-Haskell [35 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`[3] and then typing

```
java ParserDemo EvenAnd_example.txt
```

Your task will be to create a similar working parser for the language MH, following the pattern of `EvenAndParser`.

Now for 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:

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

```
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 here omit the # symbol, distinguishing nonterminals instead by the choice of font.)

$$
\begin{array}{rcl}
\mathit{Prog} & \to & \epsilon \mid \mathit{Decl}\ \mathit{Prog} \\
\mathit{Decl} & \to & \mathit{TypeDecl}\ \mathit{TermDecl} \\
\mathit{TypeDecl} & \to & \texttt{VAR}\ \texttt{::}\ \mathit{Type}\ \texttt{;} \\
\mathit{Type} & \to & \mathit{Type1}\ \mathit{TypeOps} \\
\mathit{Type1} & \to & \texttt{Integer} \mid \texttt{Bool} \mid \texttt{(}\ \mathit{Type}\ \texttt{)} \\
\mathit{TypeOps} & \to & \epsilon \mid \texttt{->}\ \mathit{Type} \\
\mathit{TermDecl} & \to & \texttt{VAR}\ \mathit{Args}\ \texttt{=}\ \mathit{Exp}\ \texttt{;} \\
\mathit{Args} & \to & \epsilon \mid \texttt{VAR}\ \mathit{Args} \\
\mathit{Exp} & \to & \mathit{Exp1} \mid \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \\
\mathit{Exp1} & \to & \mathit{Exp2}\ \mathit{Op1} \\
\mathit{Op1} & \to & \epsilon \mid \texttt{==}\ \mathit{Exp2} \mid \texttt{<}\ \mathit{Exp2} \\
\mathit{Exp2} & \to & \mathit{Exp3}\ \mathit{Ops2} \\
\mathit{Ops2} & \to & \epsilon \mid \texttt{+}\ \mathit{Exp3}\ \mathit{Ops2} \mid \texttt{-}\ \mathit{Exp3}\ \mathit{Ops2} \\
\mathit{Exp3} & \to & \mathit{Exp4}\ \mathit{Ops3} \\
\mathit{Ops3} & \to & \epsilon \mid \texttt{*}\ \mathit{Exp4}\ \mathit{Ops3} \\
\mathit{Exp4} & \to & \mathit{Exp5}\ \mathit{Ops4} \\
\mathit{Ops4} & \to & \epsilon \mid \mathit{Exp5}\ \mathit{Ops4} \\
\mathit{Exp5} & \to & \texttt{VAR} \mid \texttt{NUM} \mid \texttt{BOOLEAN} \mid \texttt{(}\ \mathit{Exp}\ \texttt{)}
\end{array}
$$

If this looks daunting at first, the following observations may be helpful:

- The grammar for types (i.e. the rules for *Type*, *Type1*, *TypeOps*) is a self-contained sub-grammar that can be understood in isolation, and defines the same language as the 'natural' grammar for types given on Page 1.
- The grammar for expressions (the rules for all nonterminals from *Exp* onwards) is another such sub-grammar, and is broadly similar in its workings to the one provided in the sample solutions to Tutorial Sheet 4. You are recommended to study the latter grammar and the parse table that accompanies it in the sample

solutions. Note that the above rules for *Exp4* and *Ops4* are intended to cater for multiple function applications, as in `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.

Part C: Typechecking for Micro-Haskell [30 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, we noted earlier that types in MH are conceptually just trees for the grammar:

Type → `Integer` | `Bool` | *Type* `->` *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:

Exp → `VAR` | `NUM` | `BOOLEAN` | *Exp Exp* | *Exp infix Exp* | `if` *Exp* `then` *Exp* `else` *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`). 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.

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. In principle, it would also be sensible to check here that the *value* of such expressions is a string of the appropriate kind (e.g. `"42"` for a `NUM`), but you are not required to do this.

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 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, my own 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.

Final remarks

To submit your work from the DICE machines, use the command

```
submit inf2a 2 MH_Lexer.java MH_Parser.java MH_Typechecker.java
```

Whilst it is not necessary for completing or testing your assignment submission, it would be satisfying to be able to complete the language processing pipeline by combining your lexer, parser and typechecker with an *evaluator* for ASTs to yield a complete working implementation of MH. Such an evaluator can be constructed quite simply, and I hope to provide one within the next few days for your general interest. Needless to say, whilst this simple evaluator illustrates the basic principle of language processing, it doesn't result in *efficient* execution of programs — for that, see the third year course on Compiling Techniques.