G53CMP Compilers: Coursework Part I

Autumn, Academic Year 2014/15

Henrik Nilsson
School of Computer Science
University of Nottingham

October 4, 2014

1 Introduction

The assessed coursework for the module G53CMP Compilers is centred around a compiler for a small language, *MiniTriangle*. You will be asked to extend MiniTriangle with various new constructs and to then extend the compiler accordingly. The compiler is called HMTC (for Haskell MiniTriangle Compiler).

There are two parts to the coursework, Part I and Part II. This document details Part I. The weights of the two parts are as follows:

- Part I: 10 %
- Part II: 15 %

Overall, the coursework is thus worth 25 % of the G53CMP mark.

The G53CMP coursework is to be carried out *individually*. You are welcome to discuss the coursework with friends, in the G53CMP Moodle forum, with the module team, etc., but, in the end, you must solve the problems on your own and demonstrate that you have done so by being able to explain your solutions as well as their wider context. The assessment of the coursework is in part written and in part *oral*. The oral examination will take place during a slot assigned to you during one of the lab sessions in the two weeks following the coursework deadline. Note that both the written and the oral parts are integral to the assessment: no marks will be awarded unless both aspects are undertaken and judged sufficiently good.

As its name suggests, the compiler is implemented in Haskell. You are referred to the main G53CMP web page for references to a number of Haskell resources that will be helpful to you should you need to brush up your Haskell knowledge. The page include references to a number of introductory texts and tutorials, including to Graham Hutton's book *Programming in Haskell*. The first 70 pages or so of this book should be enough to get you up to speed.

Additionally, there are references to documents more closely related to how Haskell is used in the context of this coursework. In particular, there is a set of "getting started" exercises, which among other things give hands-on instructions regarding GHC and the School's Haskell installations, and there is a set of slides that explain some Haskell facilities for "programming in the large" that are used in the HMTC source code but which you might not have come across before. Even if you have encountered Haskell or other functional programming languages before, it is recommended that you quickly browse through these latter documents as they cover aspects of Haskell used in the supplied code with which you may not be familiar.

Finally, the lecture notes, available via the main module web page as well, are obviously also a very important source of information.

2 Submission

For practical information about deadlines, timetables for oral vivas, and so forth, see the module web page and the coursework support page (linked from the main page). There you can also find links to other practical information; such as on electronic submission.

For Part I of the coursework, the following has to be submitted by the deadline:

- A brief written report as specified below.
- The complete source tree for the extended compiler.

The submission is part physical, part electronic:

- *Physical:* hard copy of the report to the School office
- *Electronic:*
  - Electronic copy of the report (PDF). The file should be called `xyz99u-report-partI.pdf`, where `xyz99u` should be replaced by your School of Computer Science user ID.
  - Archive of the source code hierarchy (gzipped TAR, or zip). The archive should be called `xyz99u-src-partI.tgz` or `xyz99u-src-partI.zip`,

where `xyz99u` again should be replaced by your School of Computer Science user ID, and it should contain a single top-level directory containing all the other files.

The written report should be structured by task. For each task:

- *Brief* comments about the *key* idea of the solution and any subtle aspects; a *few* sentences should suffice.
- Answers to any theoretical questions, such as additions of productions to the context free grammar or modifications of existing productions.
- All *added* or *modified* code, with enough context to make an incomplete definition easy to understand. (Thus, you should *not* include all code!)
- Anything extra that the task specifically asks for.

To exemplify the point about added and modified code, if you:

- have added a new function, then include the complete function definition, including the type signature;
- have extended a lengthy function with a few cases, then include the new cases along with immediately surrounding cases to the extent needed to make it clear where the extension was made;
- have added a constructor to a datatype, include the definition and state the name of the extended type explicitly.

3 Assessment and Feedback

Both Part I and Part II are structured by tasks, each carrying a weight: a maximal mark between 0 and 100 such that the weights of all tasks add up to 100. Each *individual* task is assessed on two aspects:

- Correctness:
  - 2 (Good): Solution entirely correct according to the specification, except possibly in some very minor way.
  - 1 (Pass): Solution mostly correct, but fails to entirely meet the specification; AND/OR minor omissions.
  - 0 (Fail): Solution mostly incorrect; AND/OR major omissions.
- Style:

  – 2 (Good): Solution elegant and simple, and thus easy to understand; code is well-written, well-formatted, tidy, good names.
  – 1 (Pass): Solution unnecessarily convoluted; AND/OR coding style has major flaws.
  – 0 (Fail): Solution is incomprehensible; AND/OR coding style is unacceptably poor.

Part II is additionally subject to an oral examination where each task is assessed as follows:

- 2 (Good): Examination demonstrated complete and thorough understanding of the problem and submitted solution.
- 1 (Pass): Examination revealed significant lack of understanding of central aspects of the problem and/or submitted solution.
- 0 (Fail): Examination revealed severe lack of understanding; OR no oral examination took place.

The mark for each Part I task is computed as follows:

$$\mathit{mark} = \mathit{weight} \times \frac{\mathit{correctness} + \mathit{style}}{4}$$

while the mark for each Part II task is computed as follows:

$$\mathit{mark} = \mathit{weight} \times \frac{\mathit{correctness} + \mathit{style}}{4} \times \mathit{score}(\mathit{oral})$$

where

$$\mathit{score}(\mathit{oral}) = \begin{cases} 0.00, & \text{if } \mathit{oral} = 0 \\ 0.65, & \text{if } \mathit{oral} = 1 \\ 1.00, & \text{if } \mathit{oral} = 2 \end{cases}$$

After marking, you will, as feedback, get your written report back with each task assessed according to the scheme above. For Part II, you will get your reports back at the end of your oral examination.

4 Getting Started

In the following, it is assumed that you are going to use the Haskell system GHC on the School's Linux/Unix servers. However, GHC is also available on the School's Windows machines, and for the most part, with the exception of using `gmake`, things work the same. However, if you do this, please read section 4.2 first, as there are some caveats that tend to cause a lot of unnecessary confusion. It may be possible to use other Haskell implementations, such as Hugs or NHC, and you could certainly use a different platform, such as Mac OS X if you prefer. But then you need to fetch and install those systems yourself, and you cannot expect the module TAs to provide much if any technical assistance if you run into trouble with your installation. The site `www.haskell.org` is your starting point for most things you might want to know about Haskell, and for downloading Haskell implementations, related tools, and documentation.

4.1 Notes for Working on the Linux Servers

The following assumes that you use one of the School's Windows machines, e.g. in the main lab A32, effectively as a terminal. Log on to your Linux server using e.g. the SSH Secure Shell Client (easiest) or PuTTY (there should be shortcuts to both on your desktop). At time of writing, the servers are *avon* for 1st year students, *bann* for 2nd year students and *clyde* for 3rd and 4th year students.

Start the interactive GHC environment by issuing the command `ghci` at the command line prompt:

```
bann$ ghci
```

Some information about GHCi gets printed, and you'll then get a new prompt:

```
Prelude>
```

From here, you can enter and evaluate Haskell expressions, load Haskell code from files, etc.

You can also edit code on the servers using text editors like Emacs (command `emacs`) or Vi (command `vi`). Using a terminal multiplexer like Screen (command `screen`; do `man screen` for info) you can start a number of interactive sessions (e.g. GHCi, Emacs, shell) and quickly and easily switch between them, all within one window. Alternatively, you can start a number of SSH sessions in separate windows by invoking the SSH client multiple times.

4.2 Notes for Working on the Windows Machines

The Haskell Platform, which includes GHCi, has been installed on the Windows machines in the lab. Just select GHCi from the start menu (you will find it under All Programs, Haskell Platform).

Note that you can navigate around the directory structure using the `:cd` command. For example, to get to the H drive:

```
:cd H:
```

Also, you can set GHCi (if it isn't already) as the default program associated with `.hs` files, so you can load them into GHCi just by clicking on them in a file browser window.

Alternatively, you can use WinGHCi. It allows you do do simple things like loading, editing, and running code through GUI shortcuts. However, the associated editor is Notepad, and as Notepad does not understand Unix line-ending conventions, you may need to work around that one way or another in certain cases; see section 4.2.1.

You can edit Haskell files on the Windows machines using editors like XEmacs or Notepad++ (there should be shortcuts to both on your desktop). Both of these adapt automatically to different line-ending conventions, but Notepad++ may need some configuration regarding the width of tab stops; see section 4.2.2.

4.2.1 Unix and Windows Line-Ending Conventions

As you may be aware, Unix (and hence also Linux, Solaris, Mac OS, etc.) and Windows use different line-ending conventions for text files. Consequently, you could encounter problems if you switch between systems. In particular, the source code for the G53CMP coursework was created under Linux, and thus uses its line-ending convention. To get around this problem, you can either use a text editor that adapts to the convention used, or you can use programs such as `unix2dos` and `dos2unix` to convert text files from Unix to Windows and vice-versa. You can run these programs (under Linux) by supplying them with the names of one or more files to convert (old files will be overwritten); for example:

```
unix2dos MyFile1.hs MyFile2.hs MyFile3.hs
```

Alternatively, you can specify input-output file pairs; for example:

```
unix2dos -n MyFile-Unix.hs MyFile-Windows.hs
```

In more detail, the Unix convention is to use a single character LF (for "Line Feed", ASCII/UNICODE character 10). The Windows convention is to use a character CR (for "Carriage Return", ASCII/UNICODE character 13) followed by LF. An additional complication is that some languages (e.g. C and Haskell) have some provisions for hiding such platform-dependent differences.

For example, the character escape sequence `\n` stands for an abstract newline character that signifies a line ending. *Internally*, this may be (and typically will be, but is not guaranteed to be) mapped to the LF character. However, for input/output purposes (in text mode), this character is mapped to and from the appropriate *external, platform specific* line-ending convention, such as LF on Unix-like platforms and CR+LF on Windows platforms. Other languages (e.g Java) takes a different approach and simply define `\n` to be LF and `\r` to be CR. For more information on these issues, see Wikipedia:

`http://en.wikipedia.org/wiki/Newline`.

Note that the HMTC scanner has been written to work with text files using both Unix and Windows line-ending conventions (to the extent visible: see above regarding platform-specific mapping between internal and external representations). Thus, when you are testing the (extended versions of) the HMTC compiler on some MiniTriangle source code, it does not matter whether this code was written using Unix or Windows line-ending conventions.

4.2.2 Haskell Layout and the Width of Tab Stops

Another issue concerns assumptions about the width of tab stops, although this is more of a tool issue (in particular, text editors, like Emacs or Notepad++) than an operating system issue. If you are using a Windows-specific editor like Notepad++, it is important that you read the following.

Parsing of Haskell programs take layout (indentation) into account (unless the structure is made explicit using curly braces and semicolons). If tab characters are used in a Haskell file, it thus become a critical question just how wide (in spaces) a tab *stop* is supposed to be, as the presence of a tab character means that the horizontal position of the next character should be aligned with the next tab stop. The Haskell language standard has a precise definition (to ensure that Haskell programs always are interpreted the same way): a tab stop is 8 spaces wide. This is also the default in many text editors, like Emacs.

However, for example Notepad++, which is a popular text editor among Windows users, has a different default: it opts for tab stops being 4 spaces wide. To avoid unnecessary grief caused by this (such as seemingly inexplicable parse errors), it is recommended that you, when editing Haskell source using Notepad++, go to Settings, Preferences, Tab Settings and change the width of tab stops to 8, and that you also tick the box "treat tabs like spaces".

If using Notepad, at least from within WinGHCi, the width of a tab stop seems to default to 8, which is appropriate for Haskell, but as noted above, it seems Notepad cannot handle Unix line-ending conventions, so you might need to convert files from Unix to Windows conventions manually.

4.3 Downloading and Compiling HMTC

First download the archive that contains the HMTC source code along with some MiniTriangle test programs. There are links from the G53CMP Compilers module web pages. To unpack the archive:

```
clyde$ tar zxvf G53CMP-CWPartI.tgz
clyde$ cd G53CMP-CWPartI
```

In the top-level directory, you will find a copy of this document and a subdirectory containing the source code for HMTC relevant to Part I. That in turn contains a further subdirectory containing some MiniTriangle test programs.

The HMTC source directory contains a makefile (called `Makefile`) that specifies how to build HMTC and its documentation. To compile HMTC, change to the source code directory and invoke GNU Make. Building the compiler is the default goal in the makefile, so no further arguments to GNU Make are needed:

```
clyde$ gmake
```

However, note that the makefile is written specifically for GNU Make, so other versions of Make will likely not work. However, on a typical Linux system, `make` is just another name for `gmake`.

The HMTC source code is well-documented, and the source code comments have furthermore been formatted to allow the generation of separate, typeset, documentation by means of Haddock, the Haskell Documentation system. To create this documentation, invoke GNU Make with the goal `doc`:

```
clyde$ gmake doc
```

At present, hyper-linked, indexed, HTML documentation is built. All documentation files are created in the subdirectory `Doc`. To view the documentation, point your browser to the file `Doc/index.html`. Browsing through this documentation is an excellent way to get familiar with HMTC. Make sure you understand how the hyper-linking works and to check out the various indices.

When building the documentation you will likely see warnings from Haddock that it cannot find link destinations for standard Haskell types like `GHC.Types.Int`. This just means there will be no hyper-links to those types in the generated documentation. This is not a problem and those warning can thus be ignored.

4.4 Using HMTC

Once HMTC has been compiled, it can be invoked to "compile" a MiniTriangle program as follows:

```
clyde$ ./hmtc filename
```

where *filename* gives the path of the program to compile. For example

```
./hmtc MTTests/test2.mt
```

Unless there are errors, you will not see very much since no code is being generated yet. However, you can ask the compiler to print the intermediate representation at various stages to get some more information. For example:

```
./hmtc --print-after-parsing MTTests/test2.mt
```

To get help with the HMTC command-line syntax, call HMTC with the `--help` option:

```
./hmtc --help
```

Note that it is possible to ask HMTC to pretty-print the intermediate representation after one or more different compilation phases, and to instruct HMTC to stop after a specific phase. For example, if the scanner and parser have been extended with new functionality, but the corresponding changes still have to be undertaken in the contextual checker, it is useful to be able to show the result both after scanning and parsing, and then stop before checking in order to test the new features in isolation:

```
./hmtc --print-after-scanning --stop-after-parsing ...
```

(Note further that an option `--stop-after-XXX` always implies the corresponding print option `--print-after-XXX`.)

HMTC can also compile a program provided through the standard input. Just invoke HMTC without any file name argument. This allows you to enter the program to compile directly, without first having to write it into a named file. Terminate the input by pressing CTRL-D immediately after a newline. For example:

```
clyde$ ./hmtc
let const x : Integer = 1 in putint(x)
CTRL-D
```

You can also test individual functions from within GHCi. In particular, there are specific test functions in the modules `Scanner` and `Parser` called `testScanner` and `testParser`. For example:

```
clyde$ ghci
Prelude> :load Scanner
Scanner> :type testScanner
testScanner :: String -> IO ()
Scanner> testScanner "1 42 xyzzy let"
Diagnostics:

Tokens:
(LitInt {liVal = 1}, line 1, column 1)
(LitInt {liVal = 42}, line 1, column 3)
(Id {idName = "xyzzy"}, line 1, column 6)
(Let, line 1, column 12)
(EOF, line 1, column 15)
```

Of course, you may want to write your own test utilities along similar lines.

Familiarise yourself with MiniTriangle and HMTC. For instance, try out HMTC on various MiniTriangle examples. Go through the documentation in order to become familiar with the various modules.

4.5 It just is not working; what is wrong?

OK, so your code just does not compile. Or it does, but HMTC still refuses to parse even a simple example. Well, don't despair; it can be made to work, and chances are you're very close. First of all, make sure you have not made any of the following simple, but very common, mistakes[1]:

- Are you *sure* your tab stops are set to be 8 characters wide? *Really* sure??? A setting of the tab stop width to some other value than 8 in Wordpad++ is responsible for a large part of all seemingly inexplicable parse errors.

- Are you loading or compiling the versions of the files you think you are using? If you're editing files on one machine, and then manually copying them over to one of the School servers for compilation there, it is easy enough to make mistakes. But it could also be that the files you are editing are being saved to a different place than you thought. Or you might have forgotten to save! Double-check that you are using the right versions. For example, look at the problematic file using a command-line tool like `less` in the same shell (window) you are using

[1]Apologies if they seem too obvious; but they really are mistakes that we come across a lot.

to compile the compiler. You might discover the file is different from the one you are editing. Or change the problematic file in various ways just to see what happens. For example, comment out bits you have changed. Or insert some deliberate errors. If nothing changes, chances are that you are compiling a different file from what you are editing.

- Have you compiled the compiler? Be aware that loading files into GHCi will neither re-build the executable `hmtc`, nor run the parser generator Happy on the file `Parser.y` to generate a new version of the parser. So, for example, if you have changed `Parser.y`, and then just load `Parser` into GHCi, you will be using an old version of the parser. Whenever you change `Parser.y`, you have to run Happy one way or another; e.g., by running Make or by running Happy manually. Similarly, if you change things in the compiler and then verify that things work through GHCi, but you then decide you want to continue testing by invoking `hmtc`, you need to re-build `hmtc` first; e.g., by running Make.

- Are you sure your MiniTriangle test program actually are valid MiniTriangle programs? Study the grammar in Appendix A carefully. In particular, note that a MiniTriangle program is a *command* (not an expression). So attempting to parse a "program" like `1 + 2` is going to fail, and rightly so. But it is easy enough to turn this into a command and thus a program; e.g., `putint(1 + 2)` or `x := 1 + 2`. (As far as the *parser* is concerned, the last program is fine; contextual checking happens later.) Another common mistake is to use semicolon as a terminator and not a separator. In particular, there must not be any semicolon after the last declaration in a list of declarations or after the last command in a `begin`-`end`-block.

5 Tasks

Part I of the coursework is concerned with scanning (or lexical analysis) and parsing. You will extend the front end of the given HMTC compiler to support a number of new language construct, which means that both the scanner and parser as such will have to be extended, as well as the internal representation of tokens (lexical symbols) and the abstract syntax.

Task I.1 (Weight 15 %) Extend MiniTriangle with a repeat-loop. Informally, the loop construct has the following syntax:

```
repeat
```

    cmd
`until`
    boolExp

The semantics is that the command *cmd* is repeated (at least once) until *boolExp* evaluates to true.

For example, the following should be a valid MiniTriangle program fragment:

```
repeat
    x := x + 1
until x > 42
```

The MiniTriangle lexical, context-free, and abstract syntax can be found in Appendix A. First extend these grammars (add or extend productions as necessary) to formally define the syntax of the new language construct. *Remember that the grammar extensions should be included in the written report!* Then extend the provided HMTC code accordingly. You will have to modify:

- `Token.hs`
- `Scanner.hs`
- `AST.hs`
- `Parser.y` (*not* `Parser.hs`: this latter file is generated from `Parser.y` by the parser generator Happy)
- `PPAST.hs`

Important! Do start by extending the grammar for the concrete syntax on paper before attempting the programming! If you are not able to correctly express the extended concrete syntax by extending the grammar, this (and the following tasks) are likely going to be very confusing to you. Conversely, once you *have* extended the grammar properly, you will find that much of the work actually is done: you can take your new grammar rules and add them to the Happy parser specification with only minor syntactic adaptations, and then add in the semantic actions to actually build the abstract syntax tree. Should you have forgotten what a context-free grammar is and how it works, please do review these notions first, e.g. by referring to the G52MAL lecture notes, before continuing.

Task I.2 (Weight 20 %) Extend MiniTriangle with C/Java-style conditional expressions. Informally, the conditional expression should have the following syntax:

$$\mathit{boolExp}\ \texttt{?}\ \mathit{exp}_1\ \texttt{:}\ \mathit{exp}_2$$

The dynamic semantics is as follows. If *boolExp* evaluates to true, then exp_1 is evaluated, and the value of the entire conditional expression is the value of that expression. If *boolExp* evaluates to false, then exp_2 is evaluated instead, and the value of the entire conditional expression is its value.

Note that MiniTriangle's `if-then-else` is a *command* (a statement), *not* an expression. The following is thus syntactically invalid in MiniTriangle:

```
(if b then x else y) + z
```

But, using the new conditional expression, the above can be expressed as follows:

```
(b ? x : y) + z
```

First extend the MiniTriangle grammars with the relevant productions. *Remember that the grammar extensions should be included in the written report!* Then extend the provided HMTC code accordingly. You will have to modify the same files as for Task I.1.

You don't have to worry (too much) about ambiguity of the context-free grammar here: keep the productions simple, and use Happy's support for operator precedence and associativity to disambiguate; that is, just follow the present design.

The `?` should be a new, distinct, token (do not treat it as an operator), and conditional expressions should be a new, distinct, kind of expressions in the abstract syntax (do not attempt to use *ExpApp*). The pair `?` and `:` *together* is an example of a *ternary* operator. However, `?` on its own is not an operator. Moreover, the typing rules for the conditional expression and its dynamic semantics (only one of exp_1 and exp_2, evaluated, not both) is such that it cannot easily be treated just as a function, unlike the binary operators. Thus, in the context of MiniTriangle, it will be much simpler to treat the conditional expression as a distinct language construct.

The conditional operator should have the lowest precedence of all operators, and it should be right associative. That is,

$$e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3\ \texttt{?}\ e_4\ \texttt{:}\ e_5$$

should be parsed as

e_1 `?` e_2 `:` `(` e_3 `?` e_4 `:` e_5 `)`

not as

`(` e_1 `?` e_2 `:` e_3 `)` `?` e_4 `:` e_5

and

e_1 `?` e_2 `?` e_3 `:` e_4 `:` e_5

should be parsed as

e_1 `?` `(` e_2 `?` e_3 `:` e_4 `)` `:` e_5

Task I.3 (Weight 35 %) Extend the syntax of MiniTriangle `if`-command so that:

- the `else`-branch is optional
- zero or more Ada-style "`elsif` ... `then` ..." are allowed after the `then`-branch but before the (now optional) `else`-branch.

For example:

```
if x > 1 then
    y := 1
```

and

```
if x < 10 then
    y := 1
elsif x == 10 then
    y := 2
elsif x > 10 then
    y := 3
else
    y := 0
```

should both be valid MiniTriangle fragments.

First extend the MiniTriangle grammars with the relevant productions. *Remember that the grammar extensions should be included in the written report!* Then extend the provided HMTC code accordingly. You will have to modify the same files as for Task I.1.

Important! Do start by extending the grammar rules on paper. If you don't understand how to express the desired syntax using grammar rules, you will likely find it very challenging to modify the Happy parser specification in a correct manner. Getting your head around how to extend the grammar to express that the `else`-branch may be left out and that there may be one or more `elsif`s with associated `then`-branches is the very key to approaching this task successfully.

Note that your extended grammar likely will be *ambiguous* (the "dangling-else" problem). This manifests itself by the parser generator Happy reporting shift/reduce conflicts. Happy's default disambiguation strategy for shift/reduce conflicts is to prefer shifting to reduction. This will result in, for example, an `else`-branch becoming associated with the nearest `if`, which is what usually is desired. Thus we choose to accept the ambiguity and the default disambiguation strategy in this case.

The definition of the constructor `CmdIf` in AST needs to be changed to accommodate the richer syntax. The pretty-printing utility functions `ppOpt` and `ppSeq` will likely be useful to you when you extend the pretty printer. For example, the second MiniTriangle fragment above should be printed along the lines:

```
CmdIf <line 1, column 1>
  ExpApp <line 1, column 4>
    ExpVar "<"
    ExpVar "x"
    ExpLitInt 10
  CmdAssign <line 2, column 5>
    ExpVar "y"
    ExpLitInt 1
  ExpApp <line 3, column 7>
    ExpVar "=="
    ExpVar "x"
    ExpLitInt 10
  CmdAssign <line 4, column 5>
    ExpVar "y"
    ExpLitInt 2
  ExpApp <line 5, column 7>
    ExpVar ">"
    ExpVar "x"
    ExpLitInt 10
  CmdAssign <line 6, column 5>
    ExpVar "y"
    ExpLitInt 3
  CmdAssign <line 8, column 5>
```

```
        ExpVar "y"
        ExpLitInt 0
```

Hint: Do the optional `else`-branch first. Make sure this works properly. Then attempt `elsif`. Introduce extra non-terminals and productions as needed.

Task I.4 (Weight 30 %) Extend MiniTriangle with character literals as described by the following productions:

Character-Literal	→	' (Graphic | Character-Escape) '
Graphic	→	*any non-control character except* ' *and* \
Character-Escape	→	\ (n | r | t | \ | ')

For example, `'1'`, `'A'`, `'z'`, `'?'`, `'\n'`, `'\\'`, `'\''` are all valid character literals. But `'\'` and `'''` are not. Thus, after your extension, you should be able to parse programs like:

```
let
    var c : Character := 'a'
in
    begin
        c := '?';
        c := '\n'
    end
```

First extend the MiniTriangle context-free and abstract grammars to accommodate character literals as a new kind of expression. Then extend the provided HMTC code accordingly.

"Non-control" characters refer to any character with a printable representation. For simplicity, you can take that to mean the ASCII range 32 (space) to 126 (tilde). The meaning of the escape sequences (i.e. the corresponding character to be carried by the token) is given by the following table:

Escape	Meaning
`\n`	New Line
`\r`	Carriage Return
`\t`	Tab
`\\`	The character \
`\'`	The character '

As it happens, the above escape sequences agree with the escape sequences that are used in Haskell to encode special characters.

You will again have to modify the same files as for Task I.1. Beside extending the scanner itself, you also need to introduce character literals as a new kind of expression into the context-free syntax of MiniTriangle, and you need to extend the representation of tokens and of the AST.

Dealing with characters and character encodings at a number of different levels at the same time can be a bit confusing. The levels have to be carefully distinguished:

- *The input character sequence*: That which is passed to the MiniTriangle scanner and which may be a textual representation of a sequence of MiniTriangle tokens and ultimately a MiniTriangle program, a program in the *object* language.
- *MiniTriangle character literals*: Specific subsequences of input characters, defined by the grammar above. Each such sequence is a constant MiniTriangle expressions whose value is a single character. Just like a sequence of digit characters (like `1`, `2`, `3`) in most programming languages, including MiniTriangle, is a constant expression whose value is an integer (in this case the integer 123).
- *Haskell character encodings*: The way strings and character literals are written in Haskell, our implementation or *meta* language. These have to be used when we implement the scanner and look for specific characters in the input sequence, or sometimes when we construct tokens that represent MiniTriangle character literals. They are also used when working in GHCi, both to enter string and character values and when string or character-valued results are printed.

In fact, making you appreciate these distinctions is part of the point of this task.

To exemplify, if the scanner encounters the *three* characters `'`, `A`, and `'` in the input sequence, it should turn those into a *single* literal character token carrying the *single* character `A`. But be aware that if you later print that token from within GHCi, the carried character will be rendered `'A'` again, as that is how Haskell happens to print character values.

To give another example, if the scanner encounters the *four* characters `'`, `\`, `n`, and `'` in the input sequence, it should turn those into a *single* literal character token carrying a *single* New Line character (character number 10 in the ASCII/ISO code). But be aware that if you later print that token, the carried character will be rendered `'\n'` again, as that is how Haskell happens to print the New Line character. Also, if you need to mention the New Line character in your Haskell code (such as when constructing a token carrying a

New Line character), you would again write `'\n'` as that is how the character literal standing for the New Line character is written in Haskell.

You may find the following hints helpful:

- *Hint 1:* At the GHCi prompt, type the expression

  ```
  length "\\\""
  ```

  (all 13 characters, exactly as above). Make sure you fully understand why the result is 2!

- *Hint 2:* Use the function `length` to check that the length of input strings to the scanner is what you think they should be.

- *Hint 3:* Avoid the whole issue of how to encode strings containing backslashes at the GHCi prompt by putting the input in a text file and reading it from there.

A MiniTriangle Grammars

This appendix contains the grammars that define the concrete and abstract syntax of the version of MiniTriangle used in this module. The concrete syntax is divided into two parts: lexical syntax and context-free syntax. The grammars are slight variations on what can be found in the book by Watt & Brown.

A.1 MiniTriangle Lexical Syntax

Non-terminals are typeset in italics, like *this*. Terminals are typeset in typewriter font, like `this`. Terminals whose spelling (the concrete character sequence) is different from what is shown in the grammar, such as names of special characters, are typeset in italics and underlined, like <u>*this*</u>. For simplicity, we resort to a slightly informal way of stating that the keywords are not valid identifiers.

Program	→	(*Token* | *Separator*)*
Token	→	*Keyword* | *Identifier* | *IntegerLiteral* | *Operator* | `,` | `;` | `:` | `:=` | `=` | `(` | `)` | <u>*eot*</u>
Keyword	→	`begin` | `const` | `do` | `else` | `end` | `if` | `in` | `let` | `then` | `var` | `while`
Identifier	→	*Letter* | *Identifier Letter* | *Identifier Digit* except *Keyword*
IntegerLiteral	→	*Digit* | *IntegerLiteral Digit*
Operator	→	`^` | `*` | `/` | `+` | `-` | `<` | `<=` | `==` | `!=` | `>=` | `>` | `&&` | `||` | `!`
Letter	→	`A` | `B` | ... | `Z` | `a` | `b` | ... | `z`
Digit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
Separator	→	*Comment* | <u>*space*</u> | <u>*eol*</u>
Comment	→	`//` (any character except <u>*eol*</u>)* <u>*eol*</u>

A.2 MiniTriangle Context-Free Syntax

Non-terminals are typeset in italics, like *this*. Terminals are typeset in typewriter font, like `this`. Terminals whose spelling (the concrete character sequence) is different from what is shown in the grammar are typeset in italics and underlined, like $\underline{\textit{Identifier}}$ and $\underline{\textit{IntegerLiteral}}$. Their spelling is defined by the lexical grammar (where they are non-terminals!).

$$
\begin{array}{lcl}
\textit{Program} & \rightarrow & \textit{Command} \\
\\
\textit{Commands} & \rightarrow & \textit{Command} \\
 & \mid & \textit{Command}\ \texttt{;}\ \textit{Commands} \\
\\
\textit{Command} & \rightarrow & \textit{VarExpression}\ \texttt{:=}\ \textit{Expression} \\
 & \mid & \textit{VarExpression}\ \texttt{(}\ \textit{Expressions}\ \texttt{)} \\
 & \mid & \texttt{if}\ \textit{Expression}\ \texttt{then}\ \textit{Command} \\
 & & \texttt{else}\ \textit{Command} \\
 & \mid & \texttt{while}\ \textit{Expression}\ \texttt{do}\ \textit{Command} \\
 & \mid & \texttt{let}\ \textit{Declarations}\ \texttt{in}\ \textit{Command} \\
 & \mid & \texttt{begin}\ \textit{Commands}\ \texttt{end} \\
\\
\textit{Expressions} & \rightarrow & \textit{Expression} \\
 & \mid & \textit{Expression}\ \texttt{,}\ \textit{Expressions} \\
\\
\textit{Expression} & \rightarrow & \textit{PrimaryExpression} \\
 & \mid & \textit{Expression}\ \textit{BinaryOperator}\ \textit{Expression} \\
\\
\textit{PrimaryExpression} & \rightarrow & \underline{\textit{IntegerLiteral}} \\
 & \mid & \textit{VarExpression} \\
 & \mid & \textit{UnaryOperator}\ \textit{PrimaryExpression} \\
 & \mid & \texttt{(}\ \textit{Expression}\ \texttt{)} \\
\\
\textit{VarExpression} & \rightarrow & \underline{\textit{Identifier}} \\
\\
\textit{BinaryOperator} & \rightarrow & \texttt{\^{}} \mid \texttt{*} \mid \texttt{/} \mid \texttt{+} \mid \texttt{-} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{\&\&} \mid \texttt{|\,|} \\
\\
\textit{UnaryOperator} & \rightarrow & \texttt{-} \mid \texttt{!}
\end{array}
$$

Declarations	$\rightarrow$	*Declaration*
	|	*Declaration* ; *Declarations*
Declaration	$\rightarrow$	`const` $\underline{\mathit{Identifier}}$: *TypeDenoter* = *Expression*
	|	`var` $\underline{\mathit{Identifier}}$: *TypeDenoter*
	|	`var` $\underline{\mathit{Identifier}}$: *TypeDenoter* := *Expression*
TypeDenoter	$\rightarrow$	$\underline{\mathit{Identifier}}$

Note that the productions for *Expression* makes the grammar as stated above ambiguous. Operator precedence and associativity for the *binary* operators as defined in the following table is used to disambiguate:

`Operator`	Precedence	Associativity
`^`	1	right
`* /`	2	left
`+ -`	3	left
`< <= == != >= >`	4	non
`&&`	5	left
`||`	6	left

A precedence level of 1 means the highest precedence, 2 means second highest, and so on.

Also note that the syntactic category *Declaration* includes *definition* of constants, as not only the type is given, but also the value of the constant (through an expression), and that a variable declaration includes an optional initialization.

A.3 MiniTriangle Abstract Syntax

This is the MiniTriangle abstract syntax. It captures the tree structure of MiniTriangle programs as concisely as possible. It is used as the basis for designing the datatypes for representing MiniTriangle programs. For example, note that there is only one non-terminal for expressions as opposed to three in the grammar for the concrete syntax. The extra non-terminals (along with a specification of binary operator associativity and precedence) are needed to make the concrete syntax *unambiguous*, which is necessary for parsing. But once a program has been successfully parsed, its structure has been determined, and ambiguity is no longer an issue. Another difference is that general function application has been introduced as one expression form. This replaces both concrete unary operator application and concrete (infix) binary operator application, as such operators *are* functions of one and two arguments, respectively. (The concrete syntax of MiniTriangle currently does not include general function application, but that could easily be added.) As a consequence, a single “variable” terminal $\underline{\textit{Name}}$ also replaces both $\underline{\textit{Identifier}}$ and $\underline{\textit{Operator}}$; i.e., $\underline{\textit{Identifier}} \subseteq \underline{\textit{Name}}$ and $\underline{\textit{Operator}} \subseteq \underline{\textit{Name}}$.

The rightmost column gives the node labels for drawing abstract syntax trees. (They correspond to the names of the data constructors of the of the abstract syntax datatypes in the compiler.) Note that some elements of concrete syntax, such as keywords, do occur in the productions. They are there to make the connection between the concrete and abstract syntax clear, and to provide an *alternative* textual representation for the abstract syntax (e.g. for use in typing rules). However, these fragments of concrete syntax are *omitted* when drawing abstract syntax trees, as they are implied by the node labels and thus superfluous. Also note that some of the productions make use of the EBNF *-notation for sequences. When drawing an abstract syntax tree, that means that the corresponding nodes will have a varying number of children.

Program	→	*Command*	Program
Command	→	*Expression* `:=` *Expression*	CmdAssign
	$\mid$	*Expression* `(` *Expression** `)`	CmdCall
	$\mid$	`begin` *Command** `end`	CmdSeq
	$\mid$	`if` *Expression* `then` *Command*	
		`else` *Command*	CmdIf
	$\mid$	`while` *Expression* `do` *Command*	CmdWhile
	$\mid$	`let` *Declaration** `in` *Command*	CmdLet
Expression	→	$\underline{\textit{IntegerLiteral}}$	ExpLitInt
	$\mid$	$\underline{\textit{Name}}$	ExpVar
	$\mid$	*Expression* `(` *Expression** `)`	ExpApp
Declaration	→	`const` $\underline{\textit{Name}}$ `:` *TypeDenoter*	
		`=` *Expression*	DeclConst
	$\mid$	`var` $\underline{\textit{Name}}$ `:` *TypeDenoter*	
		(`:=` *Expression* $\mid$ ϵ)	DeclVar
TypeDenoter	→	$\underline{\textit{Name}}$	TDBaseType