# Assessed Exercise 1 (Mandatory): Parsing, Code Generation and State Manipulation in Haskell: a Real-world Application

**Functional Programming 4**
School of Computing Science, University of Glasgow

Hand-out/hand-in: see Moodle

## The problem

The objective of this coursework is to assess your understanding of functional programming and in particular of the concepts of higher-order functions, list manipulation, state and input/output, as well as the use of libraries, code organisation and testing.

The aim of this coursework is to write a program in Haskell which will analyse a text file for specific patterns and produce a new text file according to a number of templates. To be specific, your program will analyse Fortran-90 code which has been annotated with OpenACC pragmas, and generate code for GPU acceleration using the OpenCL API. This is a real-world application of parsing, code generation and state manipulation.

However, you do not need to know what OpenACC or OpenCL are to do this coursework. You will be provided with a code skeleton and and example input file. Your parser should be able to process not just the example input file but similar files as well, as your final code will be tested against a similar but different file.

## What to hand in

Your solution should be an archive (.tgz, .tbz2 or .zip) containing a set of Haskell source code files based on the provided code skeleton.

Include full identification information at the beginning of the file in a block comment(`{- -}`), one entry per line: your name and matriculation number (one one line), the course (Functional Programming 4), the exercise title, and the date. These should all be inside a comment. See the Main.hs skeleton file, it already contains these block comments.

After the identification, include a short status report, also inside a comment. This isn't documentation of the program; it simply *reports the status of your submission.* Which parts did you do? Does your solution work? What bugs are you aware of? Which extensions, if any, did you implement?

After the status report comes the program.

Although it is essential that you perform unit tests on your code, you don't have to submit the unit test code.

If your code can't be built with the command `ghc --make Main` then you must provide a Cabal build file.

Submit your code archive on Moodle, before the hand-in deadline.

## Assessment

The entire exercise is worth 20% of the course assessment, or 15% if you also do the optional second coursework, and it will marked in bands.

The assessment is *entirely* based on unit tests. For each function provided in the code skeleton, one or more unit tests will be run to evaluate the correcness of the implementation. The mark you get is based solely on the number of unit tests passed.

## Code skeleton and example source files

The code is contained in a number of Haskell source files, in the archive `Skeleton.zip` on Moodle. Each of the files contains a number of functions and/or datatypes. For each function the type declarations and a stub are provided.

The archive `Fortran95-sources.zip` contains the Fortran 95 source code template `module_LES_ocl_TEMPL.f95` and an example of the generated code, `module_LES_ocl.f95`. You should study both source files to understand what information you need to extract from the template and what code you need to generate.

Your code *must* use the code skeleton, i.e. you must implement all functions provided in the skeleton without changing the type signature or the type declarations.

## Input and output Files

The folder `Fortran95-sources` contains the input file `module_LES_ocl_TEMPL.f95`. You *must* put this file in the *same* folder as your `Main.hs` and read it from there. Your program *must* generate an output file named `module_LES_ocl.f95` in the *same* folder.

## Some more information on Fortran-90, OpenACC and OpenCL

I will use the simplified regular expressions below to describe the format of the code. I use {regex} to enclose the regular expressions. So, for example

- {long|short} means a choice between 'long' and 'short';
- {[1-8]D} means a choices of a digit from 1 to 8, followed by 'D';
- {unsigned?} means one or zero occurences of 'unsigned';

- {[0-9]+} means one or more digits between 0 and 9, i.e. an unsigned integer value
- ${var} is used to indicate the name of a variable to be substituted

## Fortran-90 syntax

- Comments start with '!', there are no block comments
- Whitespace is not significant.
- Fortran is NOT case sensitive.
- Statements must fit on a single line, but lines can be broken up using '&'.
- If you want to put multiple statements on a single line, they must be separated by semicolons; otherwise no semicolons.
- Apart from the actual type, the other attributes of a type declaration are optional; their order is arbitrary. However, you can assume the order as used in the example code.
- A full BNF syntax for Fortran-90 can be found at `http://docs.cray.com/books/007-3694-003/html-007-3694-003/faxalchri.html`
- *NOTE:* Fortran-95 and Fortran-90 are syntactically identical.

## Fortran-90 variable and parameter declarations

Some examples of valid variable declarations:

```
real(kind=4), dimension(-1:ip+1,-1:jp+1,0:kp+1)  :: sm1, sm2, sm3
  real(kind=4), dimension(kp+2)  :: z2
  real(kind=4) :: dt
  integer :: im
  real(kind=8), intent(inout) :: sor_err
```

- the comma-separarted fields before the '::' are called *attributes*
- The *type* attribute (e.g. `real(kind=4)` or `integer(8)` has an optional *kind* field, which in its turn has the optional keyword `kind`. If this field is not specified, its value defaults to *4* (it represents the word size in bytes). So the following types are equivalent:

  ```
  real(kind=4)
  real(4)
  real
  ```

- The *intent* attribute can have values `in`, `out` or `inout` (this indicates if a subroutine argument is used for input, output or both). This attribute is optional, the default *intent* is *inout*.

- Fortran arrays start at *1* by default, so `dimension(kp+2)` is the same as `dimension(1:kp+2)`. Note that the bounds of the array can be negative values.

If a variable has the attribute `parameter`, it is a Fortran *parameter*, this means it's a constant for the run of the program. You'll need to parse the parameter declarations in order to compute the array sizes.

## OpenACC pragmas

The Fortran template code contains OpenACC pragmas. These are a special type of comment starting with `!$ACC` and occur in two forms:

1. Pragmas for code analysis: *Arguments*, *ConstArguments* and *ArgMode*

   ```
   !$ACC {Const?}Arguments
       <variable declaration>
       ....
   !$ACC End {Const?}Arguments
   ```

   These define regions of code (in the case of the coursework, variable declarations) that are required by the code generator.

   Each variable declaration can *optionally* be followed by an OpenACC pragma

   ```
   !$ACC ArgMode {Read|Write|ReadWrite}
   ```

   For example

   ```
   integer(kind=8), dimension(ip), intent(in)  :: t1 !$ACC ArgMode Read
   ```

   The *ArgMode* defaults to *ReadWrite*.

2. Pragmas for code generation

   ```
   !$ACC {BufDecls|SizeDecls|MakeSizes|MakeBuffers|SetArgs|WriteBuffers}
   ```

   These mark the place where the generated code should be inserted and identify what type of code should be generated.

OpenACC pragmas are not case sensitive and whitespace is not significant.

*NOTE:* Comments other than OpenACC pragmas have no special meaning for the code generator, and can be ignored.

## Code generation and OclWrapper OpenCL API

For each of the keywords in the pragmas for code generation above, specific code has to be generated. We use a simple naming convention: for a variable name *${var}*, the corresponding buffer will be name *${var}_buf* and the corresponding size *${var}_sz*.

- BufDecls

  Buffer declarations are of the form

  ```
  integer(8) :: ${var}_buf
  ```

- SizeDecls

  Size declarations are of the form

  ```
  integer, dimension({[1-4]}):: ${var}_sz
  ```

- MakeSizes

  There are two ways to create the sizes: generate-time computed and run-time computed.

  – For run-time computed, the form is

  ```
  ${var}_sz = shape(${var})
  ```

  For generate-time computed, the form is a Fortran array constant:

  ```
  ${var}_sz = (/ {([0-9]+, )*[0-9]+} /)
  ```

  where the size has been pre-computed by the code generator, for example

  ```
  p_sz = (/ 150, 150, 90 /)
  ```

  This means the code generator has to interpret the expressions for the dimensions of the arrays.

For generating the OpenCL code we use OclWrapper[1], an API to facilitate integration of OpenCL in large Fortran-90 codebases.

For the purpose of the coursework, the following API calls are required:

- MakeBuffers

  ```
  call oclMake{[1-4]D}{$type}Array{$mode}Buffer(${var}_buf, ${var}_sz, ${var})
  ```

where

```
$type = Int|Float
$mode = Read|Write|ReadWrite
```

- SetArgs

  For array arguments:

  ```
  call oclSet{Int|Float}ArrayArg({[0-9]+}, ${var}_buf )
  ```

  For constant arguments:

  ```
  call oclSet{Int|Float}ConstArg({[0-9]+}, ${var} )
  ```

[1] https://github.com/wimvanderbauwhede/OpenCLIntegration

Here, the integer constant reflects the order of the variables in the source code, i.e. the first encountered variable has argument position 0, the second one has position 1 and so on.

- WriteBuffers

```
call oclWrite{[1-4]D}{Int|Float}ArrayBuffer(${var}_buf, ${var}_sz, ${var})
```