# COMS22201: Language Engineering

Coursework 2: weighting 25%

16/03/2017, csxor@bristol.ac.uk

This coursework is released in week 20 so you can begin working on it in the lab of week 21. It consists of three parts involving one lab exam and two Haskell implementation exercises. Collectively these are worth 25% of your unit mark.

The lab exam (part 1) will be based on the questions you have already worked on in the labs. You will need to attend a designated closed-book session in week 23 (the first week back after Easter) for a written exam-style test.

The Haskell exercises (parts 2 and 3) concern the syntax and semantics of a richer language **Proc** that extends **While** with simple blocks and procedures. You will need to write a parser for **Proc** together with natural operational semantics that evaluates programs under various static and dynamic scoping rules. You will need to submit one file `cw2.hs` to the `CW2p23` unit component in SAFE by 4pm on Wednesday the 3rd of May in week 24 (the last week of term).

Note, due to popular demand, the deadline for Haskell submission has already been extended by one week (which means I am now in the process of re-scheduling the lab exam so you can continue working on your Haskell in the week 23 lab).

## Part 1:

The first part of the coursework is is worth 5% of the unit mark and involves a 50 minute lab exam that you will sit in week 23. Detailed instructions including time and room information will be emailed to you and posted on the course webpage over Easter. But, in brief, each student will get a script with 3 questions that are randomly selected from the previous lab sheets (where the examinable questions are explicitly marked as such on the lab solutions I've started releasing). You will need to write your answers on the paper provided. You will be given a mark in SAFE and a selection of the best answers will be put online to serve as feedback on good practice and to assist with your exam revision.

## Part 2

The second part of the coursework is worth 8% of the unit mark and involves writing a compiler for the **Proc** language, defined as follows (on p.52 of the book):

$$
\begin{array}{lcl}
S & ::= & x \texttt{ := } a \mid \texttt{skip} \mid S_1 \texttt{ ; } S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \\
 & \mid & \texttt{while } b \texttt{ do } S \mid \texttt{begin } D_V\ D_P\ S \texttt{ end} \mid \texttt{call } p \\
D_V & ::= & \texttt{var } x \texttt{ := } a \texttt{ ; } D_V \mid \epsilon \\
D_P & ::= & \texttt{proc } p \texttt{ is } S \texttt{ ; } D_P \mid \epsilon
\end{array}
$$

The idea is that statements can be made into blocks by enclosing them within the `begin` and `end` tokens and specifying a set of variable and procedure declarations $D_V$ and $D_P$; and procedures can be run using the `call` keyword.

Although the definitions of arithmetics and Booleans are unchanged from **While**, you are asked to use `!`, `&` and `<=` instead of (or as well as) $\neg$, $\wedge$ and $\leq$ to represent negation, conjunction and less than. This is because the former are easier to type and less likely to be corrupted during file transfers. Your submission will only be tested on programs using `!`, `&` and `<=` (but if you decide to also support $\neg$, $\wedge$ and $\leq$ then make sure your submission runs on a lab machine with no UTF errors).

In addition to these changes, we will allow **Proc** programs to contain parentheses `( ... )` so programmers can specify the intended parse; and, for convenience, we will also allow C-style comments of the form `//...` and `/*...*/` .

The aim is to write a parser that compiles **Proc** code, as defined above, into an abstract syntax tree built from the following Haskell type declarations (which are unchanged from the lab sheets apart from the block and procedure extensions):

```
type Num   = Integer
type Var   = String
type Pname = String
type DecV  = [(Var,Aexp)]
type DecP  = [(Pname,Stm)]

data Aexp  = N Num | V Var | Mult Aexp Aexp
           | Add Aexp Aexp | Sub Aexp Aexp
data Bexp  = TRUE | FALSE | Neg Bexp | And Bexp Bexp
           | Le Aexp Aexp | Eq Aexp Aexp
data Stm   = Skip | Ass Var Aexp | Comp Stm Stm
           | If Bexp Stm Stm | While Bexp Stm
           | Block DecV DecP Stm | Call Pname
```

In this way, we can now write source code like the following examples of iterative and recursive implementations of factorial (using loops or procedure calls):

```
/*fac_loop (p.23)*/
y:=1;
while !(x=1) do (
 y:=y*x;
 x:=x-1
)
```

```
//fac_call (p.55)
begin
  proc fac is
   begin
    var z:=x;
    if x=1 then
     skip
    else (
     x:=x-1;
     call fac;
     y:=z*y )
   end;
  (y:=1;
  call fac)
end
```

Your task is to write a function `parse :: String -> Stm` that takes a string representation of some *well-formed* **Proc** code such as (for the left program above)

```
"/*fac_loop (p.23)*/\ny:=1;\nwhile !(x=1) do (\n y:=y*x;\n x:=x-1\n)"
```

and returns a Haskell term such as (for the left program above):

```
(Comp (Ass "y" (N 1)) (While (Neg (Eq (V "x") (N 1))) (Comp (Ass "y"
      (Mult (V "y") (V "x"))) (Ass "x" (Sub (V "x") (N 1))))))
```

You are advised to use parser combinators from TB1 (and you may use Parsec libraries on lab machines). White space and comments should be discarded. Use standard precedence and associativity for arithmetics and Booleans. Although your parser will only be tested on code that includes sufficient parentheses to remove any semantic ambiguity, you may wish to make the concrete grammar unambiguous by enforcing the following conventions (working from left to right while respecting blocks and brackets): first ensure 'else' and 'do' statements are as large as possible; then ensure procedure bodies are as small as possible; and finally make composition right associative. Note that under these conventions, both the above programs could have been written without brackets (though any test programs will in fact include brackets). You should aim to spend no more than 10 hours on this part.

## Part 3

The third part of the coursework is worth 12% of the unit mark and involves writing a natural operational semantics for **Proc**. You are expected to work on your own by closely following the detailed guidance and implementation hints given in Section 2.5 and Appendix C.3 of the book.

A key aspect of providing a semantics for block structured languages like **Proc** (which allow local declarations to effectively hide global declarations of the same name) involves deciding the scoping rules for variables and procedures. In each case there is a choice of **static scope** or **dynamic scope** to determine which declaration a reference should be resolved to.

Static scope (early or compile-time binding) means that a reference is resolved lexically (in terms of the source code) to the declaration made in the innermost enclosing block. Dynamic scope (late or run-time binding) means that a reference is resolved temporally (in terms of the execution context) to the declaration made in the most recently executed block

In this coursework you will explore three combinations that we will call **static** (static variables and procedures), **mixed** (dynamic variables but static procedures), and **dynamic** (dynamic variables and procedures) These concepts are illustrated by the following program which you should be able to see terminates with `y`=5 for static scope, `y`=10 for mixed scope `y`=6 for dynamic scope:

```
//scope_test (p.53)
begin
  var x:=0;
  proc p is x:=x*2;
  proc q is call p;
  begin
   var x:=5;
   proc p is x:=x+1;
   (call q;
   y:=x)
  end
end
```

By following the exposition in the book you are required to implement a natural semantics for all three combinations. In order to do this you will need to implement various aspects such as stores, locations and variable and procedure environments.

You may write any semantic functions and auxiliary definitions you like, but you should include the following types (in addition to those given in the previous part) and you should implement three semantic functions `s_static, s_mixed, s_dynamic :: Stm -> State -> State` that evaluate a given program from a given state under the specified scoping convention:

```
type T     = Bool
type Z     = Integer
type State = Var -> Z
```

Note that type definitions of the declarations `DecV` and `DecP` use standard Haskell lists. This is simply to avoid introducing two pairs of new constructors `ConsV`, `NilV` and `ConsP`, `NilP` that would be isomorphic to the standard list constructors `:` and `[]` but would not benefit from the syntactic sugar that we can exploit here to make variable and procedure declarations easier to read with no loss of generality.

The marking scheme for this part is not linear as you will need to work harder to obtain higher marks. An approximate breakdown of marks is shown in the following table which maps the scoping rules (left to right) against the language features (top to bottom). The arrows show the direction of increasing difficulty.

| Marking scheme (approx) | Dynamic | Mixed | Static | Weight |
|---|---|---|---|---|
| Basic features (While) | easiest | $\rightarrow$ | | 20% |
| Variable scope (Block) | $\downarrow$ | $\searrow$ | | 25% |
| Procedure scope (Proc) | | | | 25% |
| Self recursion | | | | 20% |
| Mutual recursion | | | hardest | 10% |
| Weight | 40% | 30 % | 30% | 100% |

You need to decide which order to tackle the problem and when to stop. You should not necessarily aim to get 100% or tackle every feature. The book considers the three binding types one at a time, moving in turn from, left to right through the above table. You should aim to spend about 15 hours on this part (although you may be able to pinch time from the previous parts).

**Submission:** You should submit one file `cw2.hs` to the `CW2p23` unit component in SAFE by 4pm on the 3rd of May. Your file should include your parser and semantic function(s). You should ensure it loads into GHCI on a lab machine with no errors. You may loose marks if your submission it is late, incorrectly named, generates load errors, or if you modify any of the type definitions given above.