CW2 Part I: Compiler Front End for FUNC [10 MARKS]

Your overall task is to develop a compiler for the programming language given below, called FUNC. You are expected to use the Thursday labs slots, as well as private questions to Arash & the Lab Helpers throughout the course to work on it. This overall task is composed of two parts:

- **CW2 Part I: Part I** is concerned with the implementation of **the compiler's front end** (this document). This is worth **10 marks**.
- **CW2 Part II: Part II** is concerned with the implementation of **the compiler's back end**. This is also worth **10 marks** and released later in due course. The compiler should eventually produce MIPS code that can be emulated using the MARS MIPS emulator.

SUBMISSION DETAILS below on page 3.

The Source Language: FUNC

The FUNC language has the following syntax:

```
<program> ::= <methods>
<methods> ::= <method>;[<methods>]
<method> ::= method <id>([<args>]) [vars <args>]
     begin <statements> [return <id>;] endmethod
<args> ::= <id>[,<args>]
<statements> ::= <statement>;[<statements>]
<statement> ::= <assign> | <if> | <while> | <rw>
<rw> ::= read <id> | write <exp>
<assign> ::= <id> := <exp>
<if> ::= if  <cond> then <statements> [else <statements>] endif
<while> ::= while <cond> begin <statements> endwhile
<cond> ::= <bop> ( <exps> )
<bop> ::= less | lessEq | eq | nEq
<exps> ::= <exp> [,<exps>]
<exp> ::= <id>[( <exps> )] | <int>
<int>  is a natural number
<id>  is any string starting with character followed by characters or numbers (that is disjoint from the keywords)
```

- Each program should have a function called **main** with no arguments and no return value.
- All other functions should an optional return value.
- You should support the following built-in functions - assume they have been defined; they accept **two** integers and returns an integer:
    - **plus,** which adds its arguments;
    - **times,** which multiplies its arguments;
    - **minus,** which subtracts its arguments;
    - **divide,** which divides its arguments.
- All the boolean operators (**less, lessEq, eq, nEq)** are also binary, i.e. take two arguments.
- The **read** command assumes that the given variable is an int variable.

The following example illustrates a valid FUNC program (more examples later in the document):

```
method pow(x, y) vars i, res
begin
   res := x;
   i := 1;
   while less(i,y)
   begin
      res := times(res,x);
      i := plus(i,1);
   endwhile;
   write res;
   return res;
endmethod;

method main() vars a, b, x
begin
   a := 5; b := 2;
   x := pow(b,a);
   if  eq(x,32) then write 1; else write 0; endif;
endmethod;
```

PART 1: Front End: 10 Marks

Your task for Part I is to implement the front end of a compiler for FUNC. To complete this part you need to understand lexical analysis, syntax analysis and abstract syntax trees. Specifically, your tasks are:

Task 1: *To produce A FLEX file* with a suitable token representation for the FUNC language (3 marks) and to generate a lexical analyser for FUNC

Task 2: *To implement a recursive descent parser* for the grammar given above (4 marks), using (1)

Task 3: *Produce an AST parser* with a suitable representation of the nodes generated by the parser (3 marks).

You can do Tasks 1, 2 and 3 either in C, or in Java – but we strongly recommend that you do it in C.

What/How to submit

Please **submit a .zip file with your code on Vision.** For the sake of uniformity, and if you are hopefully submitting C code, your code **MUST** use/extend the starter code/templates provided:

1. Download **the starter code** from Vision -> F29LP -> Coursework -> CW2 Part I (Compiler Front-end): `CW2.1-Starter-code.zip`
2. There is a **`README.txt`** file which you should read. It contains a description of each of the files in the package & explains the steps to *compile & test your code.*
3. To generate the lexical analyser (1 above), you'll need to complete the very partially specified `func.flex` file and use this to generate the lexical analyser.
4. For the recursive decent parser without AST construction (2 above), complete the `parser.c` file – there is already some helper functions defined for you.
5. For (3) complete the `ast-parser.c` file – there is already some helper functions defined for you
    a. Note that **the code includes an implementation of a multiple branching tree data structure in C.** This is in the `tree.h` header file which is implemented in the `tree.c` file. You can use the tree manipulation functions included (they each have a small comment explaining what they do, and are otherwise self-explanatory). You are also welcome to use your own tree implementation.

Note 1: Even though the AST parser (3 above) is essentially an extension of the recursive decent parser (Task 2), please submit your solutions separately in the `parser.c` and `ast-parser.c` files respectively for Tasks 2 & 3.
Note 2: No need to submit binaries (.o files) or any executables – the marker will compile your code locally.

Happy coding!

Appendix: Some more examples of valid FUNC code – you can use these to test your parser(s) – these are included in the starter code package as `ex1.func` and `ex2.func`

```
method main() vars inp, res
begin
   read inp;
   res:=0;
   while less(0,inp)
   begin
      res := plus(res,inp);
      inp := minus(inp,1);
   endwhile;
   write res;
endmethod;
```

(More on next page)

```
method sum(inp) vars res
begin
   res:=0;
   while less(0,inp)
   begin
      res := plus(res,inp);
      inp := minus(inp,1);
   endwhile;
   return res;
endmethod;

method main() vars inp,res
begin
   read inp;
   res := sum(inp);
   write res;
endmethod;
```

```
method sum(inp) vars tmp
begin
   if eq(inp,0) then
      res := inp;
   else
      tmp := sum(minus(inp,1));
      res := plus(tmp,inp);
   endif;
endmethod;

method main() vars inp,res
begin
   read inp;
   res := sum(inp);
   write res;
endmethod;
```