ADVANCED PROGRAMMING

The Final Programming Project should be submitted by e-mail by **Jan 7 2015 (midnight)** to **giangi@di.unipi.it** and **cisterni@di.unipi.it** with the subject prefix **[AP-Final]**.

You can solve exercises using a programming language of choice among: C/C++, Java, C#, F#, Javascript. The submission must include all required files for compiling and executing the proposed solution.

It is allowed to discuss the problem among the class but the proposed solution should be individual. Violation of this rule imply to be reported to the President of Degree.

1. The Funw@p Programming Language

Modern programming languages support multiple programming paradigms. They mix imperative, object oriented and functional programming styles into a single programming language. Notable examples include Python, Javascript, Java 8, Scala and C#4.0 to cite a few. For the AP final programming project we ask you to create a modern programming language that has an imperative structure, with parallel execution support and higher order functions.

Language Description

Funw@p (read fun with AP) is a domain specific programming language designed to support general mathematical and logical (boolean) operations. The language has native support for higher order functions and parallel programming. Furthermore, in order to implement higher order functions, we need to take advantage from closures. Furthermore it is possible to simply define blocks of code that run on separate threads. The Funw@p language allows expressing parallel threads in an easy way.

Define the parser and the compiler for the Funw@p language.
The implementation should not interpret the language but generate an equivalent program in the language of choice.
It is up to the candidate to fill the required gaps in the specification and provide motivations for the choices made.

Programming Examples

The syntax of Funw@p is standard. Some programming examples follow.

```
fun main() {
// single-line comments
// a is 0, so is b

 var a int = 0;
 var b int = 0;

// a_eq_b is TRUE
 var a_eq_b bool = a == b;

 for i := 0; i < 32; i++ {
  a += i }

 if a_eq_b && a == 0 {
    println(”a is 0, so is b”);
  } else {
    println(”at least one of a b is not zero”)
  }
}
```

The **fun** keyword indicates that the block of code that follows is a routine. They are first class citizens in Funw@p so they can be assigned to variables and passed/returned as arguments. Since functions can be passed around, closures become necessary to bind free variables of a function to a specific variable.

```
// fun ID (PARAMS) RETURN_TYPE
  fun outside_adder() fun(int) int {
    // Closure
    var sum int = 10
    // Anonymous functions supported, without the ID
    return fun(x int) int {
      sum += x
      return sum
    }
```

```
fun main() {
   var adder fun = outside_adder(); //Using closure properly
   var another_adder fun ;
   var yet_another_adder fun;
   println(adder(5)) // 15

 //First class variable
   another_adder = adder ;
  println(another_adder(60)); // 75
  yet_another_adder = outside_adder();

// A differen closure
   println(yet_another_adder(60)); // 70
   println(adder(60)); // 135
}
```

Code that can be executed asynchronously is prefixed by the **async** keyword. We can wrap a generic piece of code inside an **async** block. Race conditions are avoided by performing a closure of the code inside the **async** block. In other words, the **async** block has no side effects. Only a single assignment to a variable in the outside scope is allowed (using the keyword return).

```
// Asynchronous Fibonacci Function
fun fib (n int) int {
  var a, b int;
  //We do not wait for fib(n-1)
  a = async{return bib(n-1)}
  // ... because we perform fib(n-2) in  parallel
  b = async{return bib(n-2)}
 // but here we wait ....
  return a+ b;
}
```

2. Extending the Funw@p Programming Language

The Funw@p extension helps you distribute your computationally extensive or network related problems across multiple machines in a cluster. Those machines can execute their portion of the problem and then send results back to the master program.

The true magic of DsitributedFunw@p is in one, special primitive called ”**dasync**”. This primitive takes a network address (a URL) and a function and provides back the result of the call.

```
// Distributed Fibonacci Function
fun fib (n int) int {
  var a, b int;
 var u: url;
 u = readln();
  //We do not wait for fib(n-1)
  a = async{return bib(n-1)}
  // ... because we perform fib(n-2) in  a remote site
  b = dasync{u, return bib(n-2)}
 // but here we wait ....
return a+ b;
}
```