Lab Work – Just-in-Time Compilation[1]

Name: _____ ID: _____

The goal of this assignment is to augment the simple language that we have compiled via LLVM with syntax to handle exponentiation. A description of our language, as well as of its LLVM compiler, is available on-line[2]. To remind you of what it can do, our language solves simple lambda expressions, involving only multiplication and addition with integer constants. The implementation of our compiler is also available on-line[3].

We will add a new syntax to our programming language, so that a construction like (^ e c) denotes the expression e^c, where e is any valid expression, and c is a constant. Notice that we only admit constants as exponents. A few examples of acceptable outputs, with the optimizer turned on, are shown below.

```
$> ./driver 3

^ 4 5

Module before optimizations:
; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  ret i32 1024
}
Module after optimizations:
; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  ret i32 1024
}
Result: 1024
```

```
$> ./driver 3

^ x 4

Module before optimizations:
; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %expAux = mul i32 %x, %x
  %expAux1 = mul i32 %expAux, %expAux
  %prodAux = mul i32 1, %expAux1
  ret i32 %prodAux
}
Module after optimizations:
; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %expAux = mul i32 %x, %x
  %prodAux = mul i32 %expAux, %expAux
  ret i32 %prodAux
}
Result: 81
```

```
$> ./driver 3

^ * x x + 2 1

Module before optimizations:
; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %multmp = mul i32 %x, %x
  %expAux = mul i32 %multmp, %multmp
  %expAux1 = mul i32 %expAux, %expAux
  %prodAux = mul i32 1, %multmp
  %prodAux2 = mul i32 %prodAux, %expAux
  ret i32 %prodAux2
}
Module after optimizations:
; ModuleID = 'Example'

define i32 @fun(i32 %x) {
entry:
  %multmp = mul i32 %x, %x
  %0 = mul i32 %multmp, %x
  %1 = mul i32 %0, %0
  ret i32 %1
}
Result: 729
```

We are implementing exponentiation by multiplying powers of two of the base e, according to the binary representation of the exponent. Figure 1 clarifies this method with an example, e.g., 3^{10}. Use this algorithm to guide your code generator when producing the instructions necessary to implement the exponentiation. Do not worry about producing more instructions than necessary: the optimizer will remove them away. The steps to add the new syntax, and its companion semantics, to our programming language are described in what follows.

1. Download and compile the implementation of our compiler, which is available in the course's webpage.
2. Add a new class `ExpExpr`, subclass of `Expr`, to our implementation. The declaration of this class, in the file `Expr.h`, is given below. We have provided just a stub to the method `eval()`, as it will not be necessary in the rest of this exercise.

[1]The material necessary for this assignment is available at `http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/lab/exercises/JIT.tgz`

[2]`http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/LLVM/LLVM_005.pdf`

[3]A version for LLVM 3.4 is available at `http://homepages.dcc.ufmg.br~fernando/classes/dcc888/ementa/slides/LLVM/LLVM_005.zip`. The same website also contains a version for LLVM 3.5.

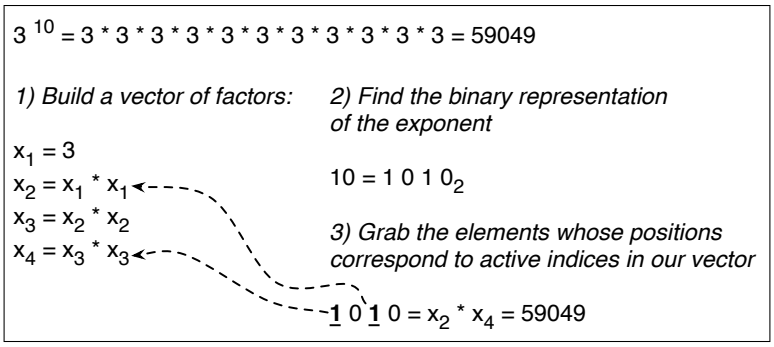

Figure 1: Algorithm by example: exponentiation with constant factor.

```
class ExpExpr : public Expr {
  public:
    ExpExpr(Expr* baseArg, int expArg) : base(baseArg), exp(expArg) {
    }
    int eval() const { return 0; }
    llvm::Value *gen(llvm::IRBuilder<> *builder, llvm::LLVMContext& con) const;
  private:
    const Expr* base;
    const int exp;
};
```

3. Change the method `Parser::parseExpr()` in file `Parser.cpp` so that your compiler can recognize the exponentiation symbol. Part of the new implementation is given below. You will have to implement the missing bit of it.

```
Expr* Parser::parseExpr() {
  std::string tk = lexer->getToken();
  if (tk == "") {
    return NULL;
  } else if (isdigit(tk[0])) {
    return new NumExpr(atoi(tk.c_str()));
  } else if (tk[0] == 'x') {
    return new VarExpr();
  } else if (tk[0] == '+') {
    Expr *op1 = parseExpr();
    Expr *op2 = parseExpr();
    return new AddExpr(op1, op2);
  } else if (tk[0] == '*') {
    Expr *op1 = parseExpr();
    Expr *op2 = parseExpr();
    return new MulExpr(op1, op2);
  } else if (tk[0] == '^') {
    // Insert your code here.
  } else {
    return NULL;
  }
}
```

4. Implement the method `ExpExpr::gen(...)` in file `Expr.cpp`. This is the more complicated part of this exercise. You will have to implement the algorithm used to carry on the example in Figure 1.