Notes on Code generation

J.R.B. Cockett
Department of Computer Science, University of Calgary,
Calgary, T2N 1N4, Alberta, Canada

September 14, 2007

CPSC510 – Implementing register allocation for trees (see Aho-Ullman chapter 9 especially section 9.10 and also Appel 11.5)

1 Ershov numbers

It is possible to work out very simply the minimal number of registers one requires to compute a tree with operations of arbitrary arity. This is called the Ershov[1] number of the tree. The basic idea is that you have to compute the argument which needs the most registers first. Once you have computed that argument you have to hold the result, as an intermediate value, in a register. Thus, the overall register requirement is the maximum of the register requirement for the argument you computed first and one more register than the remaining register requirement.

Thus, suppose you are computing

```
op5(x1,x2,x3,x4,x5)
```

and you discover recursively that

```
need(x1) = 3
need(x2) = 3
need(x3) = 5
need(x4) = 6
need(x5) = 3
```

then you should compute these in order of greatest register need

```
[x4,x3,x2,x5,x1].
```

When you have computed `x4` you will need a register to hold its result, and when you have computed `x3` you will need a register to hold its result, and .... so on. By doing the computations in this order, the high register need computations are done when the number of intermediate values which need to be stored is lowest. This reduces the overall register requirement.

[1] Academician Andrey Petrovych Ershov (19 April 1931 - 8 December 1988) was a Soviet computer scientist, notable as a pioneer in systems programming and programming language research. He was responsible for the languages ALPHA and Rapira, AIST-0 the first Soviet time-sharing system, electronic publishing system RUBIN, and MRAMOR, a multiprocessing workstation.

To calculate the number of registers required one must add a register cost across this list to indicate what one must store at that stage. This is given by the register "ramp":

```
[0,1,2,3,4].
```

The register need of the whole computations is then the maximum

```
MAX [need(x4) + 0,need(x3) + 1,need(x2) + 2,need(x5) + 3,need(x1) + 4]
 = MAX [6 + 0,5 + 1,3 + 2,3 + 3,3 + 4]
```

which is seven registers.

It is important to note that, when we come to consider spilling, essentially this same need function can be used to calculate the stack space requirement by the expression. Indeed just stare at this calculation and you will realize that for a stack machine we have just calculated the (minimal) stack space required for calculating the expression.

Also note this technique can be simplified greatly in the case that you have only two arguments. In this case you always do the most needy argument first and this is the need unless both require the same number N of registers in which case the need is $N + 1$.

Here are some questions:

1. Find the smallest binary expression trees which have a register need 5, 6, 7, 8, 9, 10.
2. Is it true that the depth/hieght of the expression is always less than the register need? Or is it always greater than the register need?
3. Which sorts of expression trees have the highest Eshov number?

1.1 Implementation

To implement this we need a datatype which allows operations of arbitrary arity (which means this belongs to the "Rose tree" family of datatypes), a sorting routine (here we use a quick sort), the ability to calculate the maximum of a list of integers, and the abilty to add a "ramp" `[0,1,2,3,..]` of register needs as the arguments increase.

Here is the Haskell code with commentary – which you can run if you copy this into Haskell!! First we have a number of standard utility programs (many of which are in the standard prelude):

```
concat::[[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ (concat xs)

zipWith::(a -> b ->c) -> [a] -> [b] -> c
zipWith  [] _ = []
zipWith  _ [] = []
zipWith (x:xs) (y:ys) = (f x y):(zipWith f xs ys)

len [] = 0
len (_:xs) = 1+(len xs)
```

```
max:: Integer -> Integer -> Integer
max x y |x >= y = x
        |otherwise = y

maximum:: [Integer] -> Integer
maximum [] = 0
maximum (n::ns) = max n (maxs ns)

qsort:: Ord a => [a] -> [a]
qsort = sort (\x y -> x <= y)

sort:: (a -> a -> Bool) -> [a] -> [a]
sort _ [] = []
sort p (x:xs) = (sort p [y | y <- xs, not (p x y)])
                ++ [x] ++
                (sort p [y | y <- xs, p x y])

monus:: Int -> Int -> Int
monus x y| x-y > 0 = x-y
         | otherwise = 0

-- Various projections
pr13(x,_,_) = x
pr23(_,x,_) = x
pr33(_,_,x) = x
pr14(x,_,_,_) = x
pr24(_,x,_,_) = x
pr34(_,_,x,_) = x
pr34(_,_,_,x) = x
```

Then we have the code for calculating need. This has two components:

1. We must handle the calculation of the need at operations: this involves zipping a register ramp to the sorted list of arguments and taking the maximum.
2. We must then do this recursively across the expression tree (a fold!).

To calculate the register need at an operation we assume we have a list of register needs for each argument we then zip these needs with the register ramp and take their maximum. You always need one register for the output. This gives the following:

```
reg_width:: String -> [integer] -> Integer
reg_width _ ts  =  maximum (1::(zipWith (+) (qsort ts) [0..])))
```

Next we define the datatype of expressions and develop its show function and the fold function

```
data  Exp = OPN String [Exp]
          | VAR String
```

```
instance Show Exp where
    show (VAR str) = str
    show (OPN name args) = name++"("++(display args)++")" where
          display [] = ""
          display [t] = show t
          display (t:ts) = (show t)++","++(display ts)

-- fold over expressions
fold_exp:: (String -> [b] -> b) -> (String -> b) -> Exp -> b
fold_exp g v (OPN s ts)  = g s (map (fold_exp g v) ts)
fold_exp g v (VAR s)  = v s
```

This leaves the calculation of the register need itself which is done by folding over the tree:

```
register_need:: Exp -> Integer
register_need = fold_exp reg_width (\_ -> 1)
```

1.2 Generating machine code

Now this is a little unconvincing as we have not generated any code: so let us add this aspect to the program above so that we actually generate some simple instructions.

The first problem is to have a simple notation for the instructions into which we will translate the expressions. For this we require a bank of registers. We shall have three sorts:

General registers: These are numbered variables staring with `r`:

```
r0, r1, r2, r3, r4, ...
```

Frame pointer: The frame pointer holds the satrt address of the current activation record. to access things in the record one uses an offset which is denoted by the backslash operator: thus `fp\3` means take offset `3` from the frame pointer. In general the offset can be given by a register. One can store things at and load things from such locations.

General storage locations: A general storage location is a pointer to location in memory from which an offset can be taken. As for the frame pointer the offset is indicated by the backslash `x\3`.

The registers are implemented by the following datatype:

```
data Register  = REG Int | LOC String | FP

instance Show Register where
  show (REG n) = "r"++(show n)
  show FP = "fp"
  show (LOC str) = str
```

Next we have to describe some basic operations. We shall use three basic "three address" style instructions:

ASSIGN: An assigmment of an operation on registers to a register:

```
r1 = ADD(r2,r1)
```

We shall allow n-ary operations (for function calls). Usually machine codes will allow only binary operations as the instructions are of fixed width.

LOAD: This instruction loads a value from a main memory location, specified by a location and an offset, and puts it into a register:

```
r1 <- x\r2
```

STORE: This instruction stores a value in a register at a location in main memory location:

```
r1 -> x\r2
```

These instructions with their show functions are implemented by:

```
data Instr = APP   String Register [Register]
           | STORE Register Int Register
           | LOAD  Register Register Int

instance Show Instr where
  show (APP str r rs) = "\t"++(show r)++" = "++str++(args rs)++"\n" where
    args rs = "("++(display rs)++")" where
        display [] = ""
        display [r] = show r
        display (r:rs) = (show r)++","++(args' rs)
  show (STORE r n r') = "\t"++(show r')++" -> "++(show r)++"\\"++(show n)++"\n"
  show (LOAD r str) = "\t"++(show r)++" <- "++(show r')++"\\"++(show n)++"\n"
```

THe code we shall produce depends on the starting register number, which depends on the number of intermediate values we need to remember – as these must be held in registers with indices lower than that starting register number. We shall use an implementation trick when we generate the code: we shall leave it dependent on the starting register number. This means that when we reorder arguments to obtain the best computational arrangement we can obtain the actual register allocation for the computation at an argument by just applying its dependent code to the correct starting register number. Hurrah for λ-abstraction and functional programming!! Of course, this could also be achieved by generating code starting at register zero and traversing the code and adjusting all the register numbers to allow for the actual starting point of the allocation.

The fact that we are no longer computing the arguments of an operation in order does creates a little problem as which register holds which argument has to be tracked. To do this we tuple each argument not only with its need and dependent code but also with its argument number:

```
(NEED,dependent_code,ARGUMENT_NUMBER)
```

The inverse of the permutation which is produced on the arguments, by computing them in greatest need order, must be performed to determine which register holds the value of each argument. This is the purpose of the `inv` function. The most important function in the code is `dep_code_gen` this produces both the register need *and* the code dependent on a register starting address.

```
gencode:: Exp -> [Instr]
gencode t = snd (dep_code_gen t) 1 where
  dep_code_gen:: Exp -> (Int,Int -> [Instr])
  dep_code_gen (VAR str) = (1,(\n -> [LOAD (REG n) (LOC str) 0]))
  dep_code_gen (OPN str rL) =  (new_register_need,new_dcode) where
   sub_dcodes = sort (\(n,_,_) (m,_,_) -> n >= m)
                 (zipWith (\ (x,y) n -> (x,y,n)) (map dep_code_gen rL) [0..])
   inv xs = map fst (sort (\(_,n) (_,m) -> m >= n) (zip [0..] xs))
   new_dcode m = (concat ( zipWith  (\(n,c) -> c (n+m))
                                    [0..]
                                    (map pr23 sub_dcodes))
                  ++ [APP str (REG m)
                        (map (\n -> REG(n+m))
                             (inv (map pr33 sub_dcodes)))] )
   new_register_need =
       maximum (1:(zipWith (+) [0..] (map pr13 sub_dcodes)))
```

1.3 Examples without spilling

1. When we generate code on:

   ```
   OPN "ADD" [OPN "ADD" [VAR "x1",VAR "x2"]
             ,VAR "x1"]
   ```

   which we may write as

   $$(x_1 + x_2) + x_1$$

   we generate the code:

   ```
   r1 <- x1\0
   r2 <- x2\0
   r1 = r1 + r2
   r2 <- x1\0
   r1 = r1 + r2
   ```

2. When we generate code on:

   ```
   OPN "ADD" [VAR "x1"
             ,OPN "ADD" [VAR "x2",VAR "x3"]]
   ```

   which we may be written more understandably as:

   $$x_1 + (x_2 + x_3)$$

   we no longer do a left to right evaluation instead we have:

```
r1 <- x2\0
r2 <- x3\0
r1 = r1 +r2
r2 <- x1\0
r1 = r2+r1
```

3. When we generate code on:

$$\text{fun}_3(x_1, (x_1 + x_2) * (x_3 + x_4), (x_5/x_6) + (x_7/x_8))$$

we get:

```
r1 <- x1\0
r2 <- x2\0
r1 = r1+r2
r2 <- x3\0
r3 <- x4\0
r2 = r2+r3
r1 = r1*r2
r2 <- x5\0
r3 <- x6\0
r2 = r2/r3
r3 <- x7\0
r4 <- x8\0
r3 = r3/r4
r2 = r2+r3
r3 <- x1\0
r1 = fun3(r3,r1,r2)
```

where we note that the order in which the arguments are evaluated has been altered so that the more expensive arguments are evaluated first. (The complete reversal is a consequence of our – ok MY – sorting algorithm! It swaps "equal need" computations on even boundaries being a merge sort which splits lists into even and odd elements ..)

It is worth noting that this code is not optimal as we are reloading variables which are repeated at the leaves of the tree. If these variables all separate things are good! However, if not it would clearly be better to put these repeated variable values in registers so that they are to be used again without accessing main memory. However, adding such a possibility to the problem makes the register allocation problem much more complex – in fact intractable. So it is important to accept this limitation (at the moment)!

Here are some questions:

1. Develop your own code and test it on the above examples and invent some of your own examples!

2. Try to do an optimal register allocation in which you avoid reloading variables. Develop some examples and illustrate the difficulty.

1.4 The Sethi-Ullman algorithm with spilling

The last added sophistication to this suite of algorithms is to add spilling into the picture. This is the Sethi-Ullman algorithm. The idea is that we run into trouble in the above algorithm when more than one subtree requires the full K registers because then we no longer have the required registers to store the intermediate result. So we have to decide what to do in that case!

Again the situation is that we have a number of subtrees whose register requirements we know. So suppose we are computing (as above)

```
op5(x1,x2,x3,x4,x5)
```

and $K = 5$. Suppose also that we have discovered recursively that

```
need(x1) = 4
need(x2) = 2
need(x3) = 5
need(x4) = 5
need(x5) = 4
```

Notice that no subtree requires more than five registers as this would have already precipitated spilling. As before we will compute the arguments in greatest register need order:

```
[x4,x3,x1,x5,x2]
```

where the needs are:

```
[ 5, 5, 4, 4, 2]
```

Unfortunately when you have computed x_4 you will need a register to hold its result in, but then the computation of x_3 requires all the registers. This means you have to spill the result of x_4. But when you have computed x_3 you will need a register to hold its result, which appears to be OK as x_1 only requires four registers ... but now you run into trouble again as you need to store two values but x_5 requires four registers!!!

So the situation is not so simple apparently! What is, of course, happening is that when you spill you do not need any registers to hold the intermediate result ... thus if did not spill the first argument you would need:

```
MAX [ 5+0, 5+1, 4+2, 4+3, 2+4] = 7
```

registers. When you spill the first argument (but none of the others) you need:

```
MAX [ 5+0, 5+0, 4+1, 4+2, 2+3] = 6
```

registers. Where notice that the maximum occurs later in the list at the fourth position ... So to determine how many arguments you have to spill you just keep shifting the point where the register "ramp" starts,

```
MAX [ 5+0, 5+0, 4+0, 4+1, 2+2] = 5
```

registers, until you get under the register width. Now you simply spill the early computations eventually reloading them into the registers which are free before performing the operation. (Clearly you MUST have the register width of the operation itself!)

Well of course another way to look at things is to realize that, if you NEED two more registers than you have, you must spill two arguments. So the calculation of the number of spills necessary can be obtained directly from the need.

Now there is another worthwhile thing to do when you do this allocation and that is to calculate the stack requirement. This is conceptually VERY simple as the number you want is simply $\mathsf{need}(t) - k$! However, you should really combine the calculation of the stack space required with the calculation that generates the code this is what done below....

1.5 An implementation of the Sethi-Ullman algorithm

Below is an implementation of the Sethi-Ullman algorithm for register allocation and spilling for a tree of operations of arbitrary arity.

We first need to be able to calculate the register width given spilling. Basically if we spill we know it must be K and if we do not spill then it is the usual calculation. However, in `spilled_width` we give the calculation from first principles.

Next we have the tricky problem of sorting out how the argument order is permutated by the calculation. This is especially tricky as spilled arguments do not fill registers until the end of the calculation. This means they are picked up in reverse after the non-spilled calculations are done. The `spill_inv` function calculates the inverse permutation given the number of spilled arguments and the order in which the arguments are calculated.

First we calculate, given the register width available and the needs, the number of spills required:

```
spilled_width:: Int -> [Int] -> int
spilled_width k ks =  monus (reg_width "" ks) k

spill_inv::Int -> Int -> [Int] -> [Int]
spill_inv base spills ps = map snd
               (sort (\(n,_) (m,_) -> m >= n) (zip ps rns)) where
            nps = length ps
            f n | n > spills = n-spills+base-1
                | otherwise = nps-n+base
            rns = map f [1..nps]
```

Note we cannot implement a function with more arguments than the register width: this we test as first we enter `genscode`.

Once again we calculate the code for each term dependent on an initial register index. The dependent code is sorted by need, as before `sub_dcodes` produces a list of (dependent code) in need order as a list of tuples:

```
(REGISTER_NEED,STACK_NEED, dependent_code,ARGUMENT_NUMBER)
```

Where `STACK_NEED` is the number of slots needed on the stack to manage spilling.

This time things are more complex as we have to be able to either generate spilled code (`spill_code`) with the spilling instructions correctly instered or ordinary code: this accounts for

the complexity of the the new dependent code, the `new_dcode` function. Here for completeness sake is the code. You should look at the examples it generates ... the code itself, I am sure, could do with more cleaning ...

```
genscode (VAR str) k nStack | k > 1
      = (1,nStack,(\n -> [LOAD (REG n) (LOC str) 0]))
genscode (OPN f0 rL) k nStack
      = (new_reg_need,new_stack_need,snd (new_dcode sub_dcodes nStack)) where
  nrL = length rL
  nargs | nrL > k = error  "WIDTH IMPOSSIBLE"
        | otherwise = nrL
  sub_dcodes = sort (\(n,_,_,_) (m,_,_,_) -> (n >= m))
                 (zipWith (\(x,y,z) n -> (x,y,z,n))
                       (map ( \x -> genscode x k nStack) rL)
                       [0..nargs])
  reg_wid = reg_width "" (map pr14 sub_dcodes)
  new_reg_need | reg_wid > k  = k
               | otherwise = max 1 reg_wid
  nspill | reg_wid > k = reg_wid - k
         | otherwise = 0
  new_stack_need = spilled_width nspill (map pr24 sub_dcodes)
  new_dcode aRGS nstack = (nsp,(\n -> (fc n) ++
              [APP f0 (REG n)
                      (map (\x -> REG x)
                      (spill_inv n nsp
                             (map pr44 aRGS)))])) where
     (nsp,fc) = begin_dcode aRGS nstack
  begin_dcode aRGS nstack | regwid <= k
   =  (0, (\nN -> concat (zipWith (\c n -> c (n+nN))
                                  (map pr34 aRGS) [0..nargs])))
                          | otherwise = (nsp, (\ _ -> code)) where
           (nsp,code) = spill_code aRGS nstack
           regwid = reg_width "" (map pr14 aRGS)
  spill_code (a:as) nstack = (spills+1,code_B) where
          code_A = (pr34 a 1)++[STORE FP nstack (REG 1)]
          decode = begin_dcode as (nstack+1)
          code_args = (snd decode) 1
          spills = fst decode
          ret_reg = (length as)+1
          code_B = code_A ++ code_args ++ [LOAD (REG ret_reg) FP nstack]
```

1.6 Examples with spilling

Here is a commentary on the example:

```
        OPN("F3", [OPN("F3",[OPN("x1",[])
```

```
                ,OPN("x2",[])
                ,OPN("x3",[])])
      ,OPN("ADD",[OPN("ADD",[OPN("y1",[]),OPN("y2",[])])
                  ,OPN("ADD",[OPN("y3",[]),OPN("y4",[])])
                  ])
      ,OPN("MUL",[OPN("F3",[OPN("z1",[])
                            ,OPN("z2",[])
                            ,OPN("z3",[])])
                  ,OPN("z5",[])])
      ])
```

which may be written more simply as

$$F_3(F_3(x_1, x_2, x_3), (y_1 + y_2) + (y_3 + y_4), F_3(z_1, z_2, z_3) * z_5)$$

The register need for this term is 5. We shall look at the effect of having various numbers of registers, K, available:

(1) $K = 5$ produces the following code with no spills:

```
r1 <- x1\0
r2 <- x3\0
r3 <- x2\0
r1 = F3(r1,r3,r2)
r2 <- z1\0
r3 <- z3\0
r4 <- z2\0
r2 = F3(r2,r4,r3)
r3 <- z5\0
r2 = r2*r3
r3 <- y1\0
r4 <- y2\0
r3 = r3+r4
r4 <- y3\0
r5 <- y4\0
r4 = r4+r5
r3 = r3+r4
r1 = F3(r1,r3,r2)
```

(2) $K = 4$ produces the following code:

```
r1 <- x1\0
r2 <- x3\0
r3 <- x2\0
r1 = F3(r1,r3,r2)
r1 -> fp\0
r1 <- z1\0
```

```
r2 <- z3\0
r3 <- z2\0
r1 = F3(r1,r3,r2)
r2 <- z5\0
r1 = r1*r2
r2 <- y1\0
r3 <- y2\0
r2 = r2+r3
r3 <- y3\0
r4 <- y4\0
r3 = r3+r4
r2 = r2+r3
r3 <- fp\0
r1 = F3(r3,r2,r1)
```

(3) $K = 3$ produces the following code:

```
r1 <- x1\0
r2 <- x3\0
r3 <- x2\0
r1 = F3(r1,r3,r2)
r1 -> fp\0
r1 <- z1\0
r2 <- z3\0
r3 <- z2\0
r1 = F3(r1,r3,r2)
r2 <- z5\0
r1 = r1*r2
r1 -> fp\1
r1 <- y1\0
r2 <- y2\0
r1 = r1+r2
r2 <- y3\0
r3 <- y4\0
r2 = r2+r3
r1 = r1+r2
r2 <- fp\1
r3 <- fp\0
r1 = F3(r3,r1,r2)
```

here, notice, there are two spills.

2 Register allocation for trees using attribute grammars

Another way of looking at the programming problem underlying register allocation for trees (or expressions) is to use an attributed grammar. The aim of this section is to show how these algorithms

become very simple to implement once one provides a description of the basic register allocation program as an attribute system (provided one knows how to implement attribute systems!).

We shall consider the simplest translation scheme for trees which involves doing a left-to-right traversal of the tree creating new registers as required. This can then be modified into an optimal register allocation and whence into one which can have the possibility of spilling.

2.1 An attribute system for creating instructions for expressions

If we have a tree or an expression it has a single value which is output: this needs to be put into a register. Furthermore, during the evaluation of the expression we may need to use some registers. However we may have already had to store certain values for later use so that it is necessary to reserve some specified registers to hold these values. Thus, in calculating the register allocation for an expression we may wish to put the answer in a certain register `A_reg` and avoid certain reserved registers `Rs_Regs`..

We may think of this as an attribute system as follows:

A. Expressions, `EXP`, have the following inherited attributes:

```
EXP.Rs_Reg, EXP.A_Reg
```

and the following synthesized attributes:

```
EXP.code, EXP.width
```

where the first of these is the code which is generated and the second is the register width required.

An operator, `OPN`, which is assumed to be a machine operation of some sort, has the synthesized attribute `OP.name`.

B. The code output depends on the reserved registers and the chosen output register.

C. The attribute system equations for the development of the basic code for an expression is as follows (where we use a meta-notation for the list of arguments to show how these are handled).

```
EXP = APP OP [EXP_1,...,EXP_n]
                    { EXP_1.RsReg = EXP.RsReg
                      EXP_1.AReg = next_free(EXP.RsReg)
                      width_1 = EXP_1.width + 0
                      EXP_2.RsReg = add(EXP_1.AReg,EXP_1.RsReg)
                      EXP_2.AReg = next_free(EXP_1.RsReg)
                      width_2 = EXP_2.width + 1
                           .....
                      EXP_n.RsReg = add(EXP_{n-1}.AReg,EXP_{n-1}.RsReg)
                      EXP_n.AReg = next_free(EXP_{n-1}.RsReg)
                      width_n = EXP_n.width + n-1
                      EXP.width = max{width_1,...,width_n}
```

```
    EXP.code =
         "EXP_1.code ;
            ....
          EXP_n.code ;
          OP.name EXP.AReg
             = OP.name(EXP_1.AReg, ... ,EXP_n.AReg)"
}
```

D. This is easily seen to be strongly non-circular!

2.2 Implementing the attribute system:

The standard way to implement this is using mutually recursive code: by now with a little bit of work you should be able to produce this code more or less automatically.

Rather than taking this approach here I shall illustrate an alternative "higher-order" approach which can be used to implement the function. This is much more closely related to the intent of the attribute system ... however, be warned, it uses the higher-order aspects of the functional style of programming and might be regarded a "tour de force" rather that something which is easily transferred into other paradigms. Despite this the ideas provide a good guide for what must be done in other languages.

(a) Requesting registers:

The first function we need is one for determining the next free register. We shall hold the "reserved" registers in an ordered list (of integers); when we ask for the next free register, we shall return not only the register number but also a revised reserved register list which includes the register just requested. The register number we return is the smallest number not on the reserve list.

```
next_free::[Int] -> (Int,[Int])
next_free res  = next 1 res where
     next::Int -> [Int] -> (Int,[Int])
     next m (n:ns)| m < n = (m,m:n:ns)
                  | otherwise = (\(x,xs) -> (x,n:xs))
                                  (next (m+1) ns)
     next m [] = (m,[m])
```

(b) The fold combinator for expressions:

The next function we shall program is the one which allows us to do a direct translation from the attribute grammar into the actual program. It reduces the overall computation to what must be done locally. This is a standard "combinator" which may be associated with any inductive datatype: it is called a fold (or sometimes a catamorphism) and is also used above:

```
data  Exp = OPN String [Exp]
          | VAR String
```

```
fold_exp:: (String -> [b] -> b) -> (String -> b) -> Exp -> b
fold_exp g v (OPN s ts)  = g s (map (fold_exp g v) ts)
fold_exp g v (VAR s)  = v s
```

This looks like a simple function but it is more complicated than it seems because of the use of the `map` combinator on lists which, recall, has the following effect:

```
map f [a1,a2,...,an] = [f(a1),f(a2),...,f(an)].
```

The `fold_exp` combinator allows us to write a function from our type for expressions to any type `a` in terms of two functions

```
g: string -> [a] -> a
v: String -> a
```

In this particular application we shall the type `a` is "code which is dependent on the output and reserved registers": thus, `a` is the higher-order (or function) type:

```
a = (Int,[Int])) -> code.
```

(c) Datatype for the instructions;

We shall need a datatype for the instructions. We shall us the same as above:

```
data Register  = REG Int | LOC String | FP

data Instr = APP   String Register [Register]
           | STORE Register Int Register
           | LOAD  Register Register Int
```

(d) Implementing the equations:

We need to implement functions

```
f::String -> [(Int,[Int]) -> code] -> (Int,[Int]) -> code
v::String -> (Int,[Int]) -> code
```

It is useful to think of this code backwards. We must produce a function which takes in a register number and some reserved registers and produces code. To achieve this we may use the functions already developed for the arguments. We do this in two steps the first creates the code for the arguments and is a recursion over the list of functions requiring the reserved registers:

```
create_code_seq _ [] = ([],[])
create_code_seq rsRegs (f::fs) = (aReg::args',(f(aReg,rsRegs)) ++ code') where
     (aReg,rsRegs') = next_free rsRegs;
```

```
        (args',code') = create_code_seq rsRegs' fs

    f_create_code str lf (aReg,rsRegs) = code ++ [APP str (REG aReg) (map (\n -> REG n) args)]
           (args,code) = create_code_seq rsRegs lf
    v_create_code str (aReg,rsRegs) = [LOAD (REG aReg) (LOC str) 0]
```

(e) Fitting it all together:

Finally we use the `fold_exp` and the above `create_code` functions to complete the translation to a function which we apply to `(0,[])` which means we want the final answer in register 0 and there are no reserved registers.

```
basic_reg_alloc t = fold_exp f_create_code v_create_code t (0,[]);
```

2.3 Optimal Register allocation

To do an optimal register allocation for an expression we will have to allow for the necessity of sometimes computing arguments out of order. Again we can view this as an attribute system; however, this time the evaluation at the nodes is sometimes out of order to allow those with greatest need to be evaluated earlier. In particular this means that we must pass as a synthesized attribute the need of the subexpressions as well as the function which generates the code.

As we are going to change the order of evaluation we must keep careful track of which register holds which value. One way to do this is to make an initial assignment of registers to arguments and then to reorder the arguments for evaluation (according to greatest need). This means, when the original function is evaluated, the registers are called in the "right order" (important for translating function calls).

If we write the algorithm, as before, as a fold over the rosetree we have to write two functions. This time type `A` is to be "code which is dependent on the output and reserved registers" together with the "register need": more formally, thus, `A` is the type:

```
A = ((Int,[Int]) -> Code),Int).
```

Our strategy will be to assign registers to hold each argument by running across the list of arguments left to right and assigning unused registers. We then reorder this list by need and generate code for each argument.

The code looks like:

```
create_argument_seq _ [] = []
create_argument_seq rsRegs ((n,f):fs)
   = (n,(aReg,f)):(create_argument_seq rsRegs' fs) where
     (aReg,rsRegs') = next_free rsRegs

create_code_args _ [] = []
create_code_args rsRegs ((_,(aReg,f)):fs) =
     (f(aReg,rsRegs)) ++ (create_code_args (aReg:rsRegs) fs)

f_create_code_opt s nLF = (rw,f)  where
```

```
    rw = reg_width "" (map fst nLF)
    f (aReg,rsRegs) = code ++  [APP s (REG aReg) (map REG args)] where
          arg_seq = create_argument_seq rsRegs nLF
          args = map (\(_,(x,_)) -> x) arg_seq
          arg_ord_seq = sort (\ (n,_) (m,_) -> m <= n) arg_seq
          code = create_code_args rsRegs arg_ord_seq

v_create_code_opt::String -> (Int,(Int,[Int]) -> [Instr])
v_create_code_opt str = (1,(\(aReg,rsReg) -> [LOAD (REG aReg) (LOC str) 0]))
```

Finally we use the `fold_exp` and the above `create_code_opt` functions to complete the translation to a function which we apply to `(0,[])` which means we want the final answer in register `0` and there are no reserved registers.

```
opt_reg_alloc t = f (0,[]) where
       (_,f) = fold_exp f_create_code_opt v_create_code_opt t
```

Here are some questions:

1. How does what this code does compare to our previus code?
2. Can you extend this code to account for spilling?