```
treeCG( T, regList )
T is the root of an expression tree, regList is a
list of machine registers for expression evaluation (|regList| ≥ 2).
This algorithm implements register targetting, the
result of T will be stored at head(regList).

// some utility list access definitions
head({l1,l2,l3,...}) → l1
tail({l1,l2,l3,...}) → {l2,l3,...}
head and tail do not modify their list argument.

r1 ← head(regList)
r2 ← head(tail(regList))

if ( T.kind is IntegerIdentifier ) then (
    LD r1,@<ident>
) else if ( T.kind is FloatIdentifier ) then (
    FLD r1,@<ident>    // float load
) else if ( T.kind is IntegerLiteral ) then (
    IMM r1,#<value>
) else if ( T.kind is FloatLiteral ) then (
    FIMM r1,#<value>    // immediate float initialize
) else if ( ··· other one-register ops ) then (
    ⋮
) else if ( T.kind is BinaryOp ) then
  left ← T.leftTree
  right ← T.rightTree
  if ( left.regCount ≥ |regList| AND right.regCount ≥ |regList| ) then (
    // Not enough registers for either tree, must use stack
    // space for temporary storage
    // Perhaps counterintuitive, we go ahead and eval the trees
    // with an "insufficient" regList anyway.
    treeCG( left, regList )      // left result in r1 due to register targeting
    PUSH r1          // left result on the stack now
    treeCG( right, regList )      // now right result in r1
    POP r2           // left result in r2 now
    <binaryOp> r1,r2,r1     // r1 ← r2 <binaryOp> r1
  ) else (
    // There are enough registers, but we must eval the "larger" first
    if ( left.regCount > right.regCount ) then (
      treeCG( left, regList )      // result in r1 due to register targeting
      treeCG( right, tail(regList) )      // result in r2
      <binaryOp> r1,r1,r2     // r1 ← r1 <binaryOp> r2
    ) else (
      treeCG( right, regList )
      treeCG( left, tail(regList) )
      <binaryOp> r1,r2,r1     // r1 ← r2 <binaryOp> r1
    )
  ) else if ( T.kind is FunctionCall ) then (
    callCG( T, regList, ··· )    // Future lecture :)
  )
)
```