Then

```
var x, y
init x = 20000, y = 1
while ~ (x == 0)
do (
    y := y * x ;
    x := x - 1
  ) ;
print(y)
```

Now

```
module Fact-Rec
 var y
 init y = 1
 proc fact(x) {
   if ~(x == 0)
   {
     y := y * x ;
     fact(x - 1)
   }
   else print(y)
 }
end
```

Compilers

Procedures

New pi Lib constructor

```
eq [prc] :
   < cnt : prc(I:Id, F:Formals, B:Blk) C, env : E, ... >
 =
   < cnt : C,
     env : insert(I:Id, abs(F:Formals, B:Blk), E), ... >
.

eq [prc] :
   < cnt : prc(I:Id, B:Blk) C, env : E, ... >
 =
   < cnt : C,
     env : insert(I:Id, abs(B:Blk), E), ... > .
```

Call

```
eq [cal] :
   < cnt : cal(I:Id, A:Actuals) C, ... >
 =
   < cnt : I:Id A:Actuals CAL C, ... > .

eq [cal] :
   < cnt : CAL C,
     val : V1:ValueStack
           val(abs(F:Formals, B:Blk)) V2:ValueStack, ... >
 =
   < cnt : addDec(match(F:Formals, V1:ValueStack), B:Blk)
C,
     val : V2:ValueStack, ... >.
```

Auxiliar functions

Compiling modules

```
op compileMod : Term -> Dec .

eq compileMod('module__end['token[I:Qid],
              '__['var_[T1:Term],
              '__['const_[T2:Term],
              '__['init_[T3:Term],
              T:Term]]]]) =
   dec(compileVar('var_[T1:Term],'init_[T3:Term]),
    dec(compileConst('const_[T2:Term], 'init_[T3:Term]),
       compileProc(T:Term))) .
```

Compiling procedures

```
op compileProc : TermList -> Dec .

eq compileProc(
   'proc_`(_`)_['token[O:Qid], TL:TermList, T:Term])
 =
   prc(compileId(‘token[O:Qid]),
    compileToFormals(TL:TermList), compileCmd(T:Term)) .
```

Compiling formals

```
op compileToFormals : TermList -> Formals .
eq compileToFormals(‘token[O:Qid])
 =
   par(compileId('token[O:Qid])) .

eq compileToFormals('_`,_[TL1:TermList, TL2:TermList])
 =
   for(compileToFormals(TL1:TermList),
    compileToFormals(TL2:TermList)) .
```

Compiling calls

```
op compileCmd : Term -> Cmd .

eq compileCmd('_`(_`)['token[I:Qid], 'bubble[Q:Qid]]) =
   cal(compileId('token[I:Qid]), compileActuals(Q:Qid)) .
```

Compiling actuals

```
op compileActuals : TermList -> Actuals .

eq compileActuals(Q:Qid) = compileId('token[Q:Qid]) .

eq compileActuals((TL1:TermList, '``,.Qid ,
                   TL2:TermList))
 =
   act(makeExp(TL1:TermList),
        compileActuals(TL2:TermList)) .

eq compileActuals(TL:TermList) = makeExp(TL:TermList)
[owise] .
```

Total Re Call

```
eq [cal] :
   < cnt : cal(I:Id, A:Actuals) C, ... >
 =
   < cnt : I:Id A:Actuals CAL C, ... > [variant] .

eq [cal] :
   < cnt : CAL C,
     val : V1:ValueStack
           val(abs(F:Formals, B:Blk)) V2:ValueStack, ... >
 =
   < cnt : addDec(match(F:Formals, V1:ValueStack), B:Blk) C,
     val : V2:ValueStack, ... >.
```

Match

```
op match : Formals ValueStack -> Dec .

eq match(par(I:Id), val(R:Rat)) = ref(I:Id, rat(R:Rat)) .

eq match(par(I:Id), val(B:Bool)) = ref(I:Id, boo(B:Bool)) .

eq match(for(F:Formal, L:Formals), (V:Value VS:ValueStack))
 =
   dec(match(F:Formal, V:Value),
    match(L:Formals, VS:ValueStack)) .
```

Fun

In Imp, proc are commands.

Implement **fun**, a (recursive) expression that when called returns a value.