**Efficiently Emitting Code**

**Efficiency Problem**

Consider concatenation-based implementation   
\begin{displaymath}
    [\![ ((e_1 + e_2) + e_3) + e_4 ]\!] = 
    (([\![e_1]\!]:::[\![e_2]\!]:::+):::[\![e_3]\!]:::+):::[\![e_4]\!]:::+
\end{displaymath}

Consider standard purely functional list implementation

EmptyList ::: ys = ys

(x::xs) ::: ys = x::(xs ::: ys)

How many times is $e_1$traversed in such implementation?

* quadratic behavior

Consider the related [StringBuilder](http://en.wikipedia.org/wiki/StringBuilder)

**Imperatively Producing Sequences**

Mutable variable contains code so far

Instead of appending lists of instructions, append each instruction to code so far

Just like pretty printing using ‘println’ instead of our [Printing Prefix Infix Postfix](http://lara.epfl.ch/w/cc09:printing_prefix_infix_postfix)

Advantage: very efficient

Disadvantage:

* less modular
* translation order is instruction order
  + must be careful
  + yet works well and is natural

[var](http://scala-lang.org) code : Array[Instruction]

[var](http://scala-lang.org) last : Int = 0

[def](http://scala-lang.org) emit(i : Instruction) = {

code(last) = i

last = last + 1

}

[def](http://scala-lang.org) translate(e : Expr) = e [match](http://scala-lang.org) {

[case](http://scala-lang.org) Mul(e1,e2) =>

translate(e1)

translate(e2)

emit(Imul())

[case](http://scala-lang.org) IntLiteral(c) =>

emit(Bipush(c))

...

}

**Functionally Producing Sequences**

Use *accumulating parameter*

* standard technique in functional programming

Instead of   
\begin{displaymath}
   [\![ \_ ]\!] : Tree \to List[Instruction]
\end{displaymath}  
have   
\begin{displaymath}
  transl : Tree \to List[Instruction] \to List[Instruction]
\end{displaymath}  
such that the following equation holds:   
\begin{displaymath}
  (transl\ prog\ L) = [\![ prog ]\!] ::: L
\end{displaymath}  
Here L denotes the existing list to which the translation will be added.

In particular, when existing=List(), we obtain:   
\begin{displaymath}
   [\![ prog ]\!] = transl\ prog\ List()
\end{displaymath}

Definition of transl:   
\begin{displaymath}
\begin{array}{l}
   transl\ (e1 * e2)\ L = transl\ e1\ (transl\ e2\ (* :: L)) \\
   transl\ c\ L = \textbf{bipush}(c) :: L 
\end{array}
\end{displaymath}  
No concatenation anywhere!

Task:

* consider [Printing Prefix Infix Postfix](http://lara.epfl.ch/w/cc09:printing_prefix_infix_postfix)
* rewrite prefix, infix, and postfix tree traversal so that they produce list without using concatenation, only ::

Example for infix(e : Expr)

def infixA(e : Expr, acc : List[Token]) : List[Token] = e match {

...

case Plus(lhs,rhs) => infixA(lhs, Add() :: infixA(rhs, acc))

...

}