Towards Compiling Expressions: Prefix, Infix, and Postfix Notation

Overview of Prefix, Infix, Postfix

Let f be a binary operation, e_1 e_2 two expressions

We can denote application $f(e_1,e_2)$ as follows

- in **prefix** notation $f\ e_1\ e_2$
- in **infix** notation $e_1\ f\ e_2$
- in **postfix** notation $e_1\ e_2\ f$

- Suppose that each operator (like *f*) has a known number of arguments. For nested expressions
  - infix requires parentheses in general
  - prefix and postfix do not require any parantheses!

Expressions in Different Notation

For infix, assume * binds stronger than +

There is no need for priorities or parens in the other notations

arg.list	+(x,y)	+(*(x,y),z)	+(x,*(y,z))	*(x,+(y,z))
prefix	+ x y	+ * x y z	+ x * y z	* x + y z
infix	x + y	x*y + z	x + y*z	x*(y + z)
postfix	x y +	x y * z +	x y z * +	x y z + *

Infix is the only problematic notation and leads to ambiguity

Why is it used in math? *Ambiguity* reminds us of algebraic laws:

x + y	looks same from left and from right (commutative)
x + y + z	parse trees mathematically equivalent (associative)

Convert into Prefix and Postfix

prefix

infix $((x + y) + z) + u$ $\qquad x + (y + (z + u))$

postfix

draw the trees:

Terminology:

prefix = Polish notation
(attributed to Jan Lukasiewicz from Poland)

postfix = Reverse Polish notation (RPN)

Is the sequence of characters in postfix opposite to one in prefix if we have binary operations?

What if we have only unary operations?

Compare Notation and Trees

arg.list	+(x,y)	+(*(x,y),z)	+(x,*(y,z))	*(x,+(y,z))
prefix	+ x y	+ * x y z	+ x * y z	* x + y z
infix	x + y	x*y + z	x + y*z	x*(y + z)
postfix	x y +	x y * z +	x y z * +	x y z + *

draw ASTs for each expression

How would you pretty print AST into a given form?

Simple Expressions and Tokens

```
sealed abstract class Expr
case class Var(varID: String) extends Expr
case class Plus(lhs: Expr, rhs: Expr) extends Expr
case class Times(lhs: Expr, rhs: Expr) extends Expr

sealed abstract class Token
case class ID(str : String) extends Token
case class Add extends Token
case class Mul extends Token
case class O extends Token   // (
case class C extends Token   // )
```

Printing Trees into Lists of Tokens

```
def prefix(e : Expr) : List[Token] = e match {
  case Var(id) => List(ID(id))
  case Plus(e1,e2)   => List(Add()) ::: prefix(e1) ::: prefix(e2)
  case Times(e1,e2) => List(Mul()) ::: prefix(e1) ::: prefix(e2)
 }
 def infix(e : Expr) : List[Token] = e match { // needs to emit parantheses
  case Var(id) => List(ID(id))
  case Plus(e1,e2) => List(O())::: infix(e1) ::: List(Add()) ::: infix(e2) :::List(C())
  case Times(e1,e2) => List(O())::: infix(e1) ::: List(Mul()) ::: infix(e2) :::List(C())
 }
 def postfix(e : Expr) : List[Token] = e match {
  case Var(id) => List(ID(id))
  case Plus(e1,e2)  => postfix(e1) ::: postfix(e2) ::: List(Add())
  case Times(e1,e2) => postfix(e1) ::: postfix(e2) ::: List(Mul())
 }
```

LISP: Language with Prefix Notation

- 1958 – pioneering language
- Syntax was meant to be abstract syntax
- Treats all operators as user-defined ones, so syntax does not assume the number of arguments is known
  - use parantheses in prefix notation: write f(x,y) as (f x y)

```
(defun factorial (n)
 (if (<= n 1)
     1
     (* n (factorial (- n 1)))))
```

PostScript: Language using Postfix

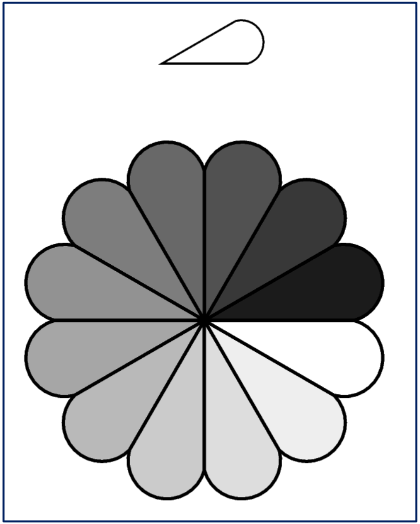
- .ps are ASCII files given to PostScript-compliant printers
- Each file is a program whose execution prints the desired pages
- http://en.wikipedia.org/wiki/PostScript%20programming%20language

PostScript language tutorial and cookbook
Adobe Systems Incorporated
Reading, MA : Addison Wesley, 1985
ISBN 0-201-10179-3 (pbk.)

A PostScript Program

```
/inch {72 mul} def
/wedge
        { newpath
          0 0 moveto
          1 0 translate
          15 rotate
          0 15 sin translate
          0 0 15 sin -90 90 arc
          closepath
        } def
gsave
  3.75 inch 7.25 inch translate
  1 inch 1 inch scale
  wedge 0.02 setlinewidth stroke
grestore
gsave
  4.25 inch 4.25 inch translate
  1.75 inch 1.75 inch scale
  0.02 setlinewidth
  1 1 12
        { 12 div setgray
          gsave
            wedge
            gsave fill grestore
            0 setgray stroke
          grestore
          30 rotate
        } for
grestore
showpage
```

If we send it to printer (or run GhostView viewer gv) we get



```
4.25 inch 4.25 inch translate
1.75 inch 1.75 inch scale
0.02 setlinewidth
1 1 12
        { 12 div setgray
          gsave
            wedge
            gsave fill grestore
            0 setgray stroke
          grestore
          30 rotate
        } for
grestore
showpage
```

Why postfix? Can evaluate it using stack

```
def postEval(env   : Map[String,Int], pexpr : Array[Token]) : Int = {  // no recursion!
   var stack : Array[Int] = new Array[Int](512)
   var top : Int = 0;   var pos : Int = 0
   while (pos < pexpr.length) {
    pexpr(pos) match {
       case ID(v) =>  top = top + 1
                      stack(top) = env(v)
       case Add() => stack(top - 1) = stack(top - 1) + stack(top)
                      top = top - 1
       case Mul() => stack(top - 1) = stack(top - 1) * stack(top)
                      top = top - 1
    }
    pos = pos + 1
   }
   stack(top)
}
```

x -> 3, y -> 4, z -> 5

infix: x*(y+z)

postfix: x y z + *

Run 'postfix' for this env

Evaluating Infix Needs Recursion

The recursive interpreter:

```
def infixEval(env : Map[String,Int], expr : Expr) : Int =
expr match {
   case Var(id) => env(id)
   case Plus(e1,e2) => infix(env,e1) + infix(env,e2)
   case Times(e1,e2) => infix(env,e1) * infix(env,e2)
}
```

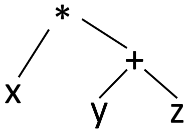
Maximal stack depth in interpreter = expression height

Compiling Expressions

- Evaluating postfix expressions is like running a stack-based virtual machine on compiled code

- Compiling expressions for stack machine is like translating expressions into postfix form

Expression, Tree, Postfix, Code



infix: x*(y+z)

postfix: x y z + *

bytecode:

get_local 1	x
get_local 2	y
get_local 3	z
i32.add	+
i32.mul	*

Show Tree, Postfix, Code

infix: (x*y + y*z + x*z)*2 tree:

postfix: bytecode:

“Printing” Trees into Bytecodes

To evaluate $e_1 * e_2$ interpreter

- evaluates e_1
- evaluates e_2
- combines the result using *

Compiler for $e_1 * e_2$ emits:

- code for e_1 that leaves result on the stack, followed by
- code for e_2 that leaves result on the stack, followed by
- arithmetic instruction that takes values from the stack and leaves the result on the stack

```
def compile(e : Expr) : List[Bytecode] = e match { // ~ postfix printer
  case Var(id) => List(Igetlocal(slotFor(id)))
  case Plus(e1,e2)  => compile(e1) ::: compile(e2) ::: List(Iadd())
  case Times(e1,e2) => compile(e1) ::: compile(e2) ::: List(Imul())
}
```

Local Variables

- Assigning indices (called *slots*) to local variables using function
  slotOf : VarSymbol → {0,1,2,3,…}
- How to compute the indices?
  - assign them in the order in which they appear in the tree

```
def compile(e : Expr) : List[Bytecode] = e match {
  case Var(id) => List(Igetlocal(slotFor(id)))
  …
}
def compileStmt(s : Statmt) : List[Bytecode] = s match {
 // id=e
 case Assign(id,e) => compile(e) ::: List(Iset_local(slotFor(id)))
  …
}
```

Shorthand Notation for Translation

[$e_1 + e_2$ **]** =

[e_1 **]**

[e_2 **]**

add

[$e_1 * e_2$ **]** =

[e_1 **]**

[e_2 **]**

mul