CS 316   3/26/2014   ①

e is $\underset{\text{SVIE}}{e_1}$ op $\underset{\text{SVIE}}{e_2}$

$a\underset{op}{+}b\underset{op}{*}c$ $\overset{e_1}{\overbrace{a+b}}\overset{op}{*}\overset{e_2}{c}$ ← **bad** decomposition ✗

$\underset{e_1}{a}\underset{op}{+}\underbrace{b*c}_{e_2}$ ← **good** decomposition. ✓

decomposition $a+b*c$

$a\underset{op}{+}(b*c)\underset{op}{/}d$

How precedence and associativity Rules of an infix Notation are used to determine which operator is Applied Last.

warming Precedence and associativity rules don't always uniquely determine which operator is applied first.

Ex: $y\overset{\downarrow}{/}z\overset{\text{applied last}}{\overset{\downarrow}{*}}--y$

Assuming -- blongs to a higher procedence class than * and / and that * and / belong to the same precedence class which is a left associative class.

The precedence and associativity rules imply * is applied last, but do not say which operator is applied first.

Sone languages may have additional rules that do determine which operator is applied first. Ex: Java has a left to right evaluation rule, which imples that / is applied first in the above expression. [C++, on the other hand, does not have this rule in C++ it is upto the compiler whether -- or / is applied first in the above expr.

int y=4
System.out.println(Y/z* --y) prints 6 in java
[would print 3 of -- were applyed first).

Operators are partitioned into ranked precedence classes.

Each precedence class is specified as being left-associative or right-associative.

To determine which operator in an expression e is applied first:

① Let e' = e without ~~its~~ ~~sam~~ surrounding parentheses if any.

Ex. $e = (x+(3*(y-7/w))*u)$
$e' = x\underset{\uparrow}{+}(3*(y-7/w))\underset{\uparrow}{*}u.$

top level means not in parentheses.

② Find all the <u>top-level</u> operators in e' [<u>top-level</u> means "<u>not</u> surrounded by parentheses"]

③ Among the operator is found at top ②, find the operators of <u>lowest</u> precedence.

④ if just one operator is found at step ③, then that operator is the operator that is applied last. Otherwise, the operator that is applied last is the left most (right most) of the operators found at step ③ / according to whether their precedence class is ~~rigt~~ right- (left-)associative.

Highest

	pretix unary	postfix unary	binary	associativity r.
class 1	~			right
class 2	+ −		+ −	left
class 3			& ^ @	right
class 4.			# $	left

lowest

which operator is applied last in

left associativity

① $\overset{\downarrow}{-} x \overset{\downarrow}{+} z \overset{\Downarrow}{\$} (\sim y \wedge z) \overset{\downarrow}{\&} a \overset{\downarrow}{@} \overset{\downarrow}{\sim} z \overset{\downarrow}{\wedge} x \underline{\underline{\overset{\Downarrow}{\#}}} y \overset{\downarrow}{-} 1$

② $\overset{\downarrow}{+} x \underline{\underline{\overset{\Downarrow}{@}}} (z \,\&\, \sim y \wedge z) \overset{\downarrow}{+} (a @ \sim z \wedge x) \overset{\Downarrow}{\&} y \overset{\downarrow}{-} 1.$

right associativity

<u>Postfix and prefix Notations</u>

prefix notation = Lisp notation without parentheses.

postfix notation* = "anti-lisp" notation without parentheses.

* also called <u>reverse Polish notation</u> or <u>RPN</u>.

In these notation you need to know the arity of each operator. You cannot use the same symbol to denote operators of different arities.

$--\ x\ y$ could mean $(-(-x)\,y)$ or $(-(-\ x\ y))$

$(-_2(-_1 x)\,y)$ $\quad(-_1(-_2\ x\ y))$

$-_2 -_1 x\ y$ $\quad(-_1 -_2\ x\ y)$

We should use <u>different symbols</u> for binary and unary − to avoid this problem.

<u>Differences between Postfix/Prefix Notations and infix Notation</u>

CS316 3/26/2014

① Operator may have any arity in postfix/prefix notation [they may only have arity 1 or 2 in infix notation]

② No parentheses in postfix/prefix notation

③ No precedence classes are used with postfix/prefix notation.

④ In postfix/prefix notation the same symbol should <u>not</u> be used to denote 2 operators of different arities.

Syntactically valid prefix/postfix expression can be defined as follows:

① A constant (5) or a variable (w) is a syntactically valid postfix/prefix expression

② If op is an operator of arity k, then.

(a) op $e_1 \cdots e_k$ is a syntactically valid prefix expr if each of the e's is such an expr.

(b) $e_1 \cdots e_k$ op is a syntactically valid postfix if each of the e's is such an expr.

Semantics In case ②, the expr. is evaluated by evaluating each of the e's and applying op ~~e~~ to the values of the e's

Postfix expression can be evaluated using a stack ; ~~Reading the expr. from left to right.~~

- Reading the expr from left to right
- whenever you see a constant or variable, push its value
- whenever you see an operator of arity k, pop k values, apply the operator to those k values and push the result.

When the entire expr has been processed, its value will be the only thing on the stack.

*i^{th} → *~~k^{th}~~ value to be popped is the ~~k^{th}~~ i^{th}-last operand.

Prefix expression can be evaluated in the same way. ~~expe~~ except that the expression is read from right to left and when k operands are popped the i^{th} operand to be ~~opp~~ popped is the ith operand of op (not the i^{th}_last)

You can use essentially the same algrithms to "put the parentheses back in a postfix /prefix expr. (to produce anti-lisp/lisp notation).

Lisp $(+_3\ (-_2\ (+_4\ x\ y\ (*_2\ z\ x)\ w)\ u)\ v\ (*_3\ x\ y\ z))$

prefix. $\boxed{+_3\ \boxed{-_2\ \boxed{+_4\ x\ y\ \boxed{*_2\ z\ x}\ w}\ u}\ v\ \boxed{*_3\ x\ y\ z}}$