**Report**

**For Data Structures # 7**

**학과 :**

**학번 :**

**이름 :**

컴퓨터공학과

20184071

김도현

**문제**

* 문제
* 다음과 같이 수식을 평가하는 프로그램을 작성하고 테스트하라.
  + 사용자로부터 임의 수식을 입력받는다.
  + 수식을 후위 식으로 변환한다.
  + 후위 식으로부터 수식 트리를 구성한다.
  + 수식 트리를 평가하고, 그 결과를 출력한다.
  + 가급적 일반적인 계산기와 유사하게 하라. (수치의 자리수/연산자 등)

1. 문제의 요구사항

- 사용자에게 숫자 및 기호 등을 입력받는다. (중위 형태)

- 사용자에게 입력받은 수식인지 정상인지 검사한다

- 정상이 아니면, 계산 실패를 출력한다.

- 정상이면 입력받은 중위 형태의 식을 후위 형태로 바꾼다

- 바꾼 후위 형태의 식을 가지고 트리를 만든다

- 위에서 생성된 트리로 수식 평가를 진행한다

- 또한, 문제에서 요구한 대로 스택을 사용하여 반복 버전의 중위 후위 탐색 결과를 출력한다.

2. 문제 분석

|  |  |
| --- | --- |
| Input | 중위 형태의 수식 |
| Output | 후위 형태의 수식 및, 계산 결과, 중위순회 후위순회 결과 |
| Process | **중위식 검사 방법**  1. 첫번째 문자가 부호인지 확인합니다. 부호면 올바르지 않은 수식 입니다  2. 마지막 문자가 부호인지 확인합니다. 부호면 올바르지 않은 수식 입니다.  3. 1 2 가 정상적으로 수행되었으면, 괄호쌍을 검사합니다. 올바르지 않은 괄호쌍이면, 올바르지 않은 수식입니다.  4. 1 2 3 이 문제 없이 수행되었으면, 후위식으로 변환합니다.  **후위식 변환 방법**  1. 숫자가 나오면 문자열에 숫자 + 공백을 추가합니다 > 2자리 이상 구분을 위함  2. 닫는 괄호나 여는 괄호가 아닌 기호가 들어오면 문자에 공백을 추가합니다 > 부호 구분을 위함  3. 스택이 비지 않고, 연산자 우선순위를 비교해서 스택에서 pop 을 수행 한 값을 문자열에 넣을지 말지 정합니다.  3 - 1문자열에 넣었으면, 마지막에 공백을 추가합니다.  4. 그 후, 스택에 받아온 문자를 넣습니다.  5. 1~4 까지 작업이 다 끝났으면, 스택이 비지 않았을 때 까지 문자열에 공백 추가후 부부호 추가하는 것을 반복합니다.  **후위식으로 노드 생성 방법**  생성된 후위식은 “공백” 을 기준으로 나누어 져 있습니다.  1. 그렇기 때문에 String 헤더를 포함하고, strtok 함수를 사용하여 공백을 기준으로 받아온 후위식을 자릅니다.  2. 자른 값이 null 이 아닐 때 까지 반복하여 진행합니다. While 문을 통하여 진행하는데, 시작될때 마다 빈 노드를 생성합니다.  3. 자른 문자열이 부호가 아니면, atoi 함수를 사용하여 문자열에서 숫자를 추출합니다. 그 후 생성된 노드의 data 를 atoi 함수의 반환값으로 설정합니다.  4. 자른 문자열이 부호면, 스택에 있는 값을 두개를 pop 을 수행합니다. 처음 pop 을 수행한 값을 노드의 오른쪽 노드로설정해주고, 두번째 pop 수행 한 값을 노드의 왼쪽 노드로 설정해줍니다.  5. 3번과 4번에서 수행된 노드는 포인터를 가지고 있으므로 스택에 포인터를 넣습니다.  6. 자른 문자열이 NULL 이여서 while 문이 종료 되면, 스택엔 루트 노드만 있을 것이므로, 그 노드를 pop 해서 반환합니다.  1 2 \* 33 + 2 – 를 그림으로 그리면 아래와 같습니다.    그 후 이 생성된 트리를 가지고 평가를 진행하면 됩니다.  트리 평가 진행 방법  재귀를 사용하여 진행합니다.  1. 입력된 root 가 NULL 이면 0을 반환합니다.  2. 입력된 root 가 잎노드면, 숫자이므로, 가지고 있는 숫자를 반환합니다.  3. 잎노드가 아니면, 잎노드를 찾기 위해 root 의 left 를 재귀적으로 자기 자신 함수를 반환 한 값을 op1 으로 할당합니다.  4. op1 으로 할당 되었으면, root 의 right 를 재귀적으로 자기 자신 함수를 호출하여 반환한 값을 op2 로 할당합니다.  5. op1 과 op2 가 할당되었으면, + - \* /등의 부호를 확인하고 그 계산된 값을 반환합니다.  반복으로 중위 탐색 진행 방법  1. 스택을 생성합니다.  2. 무한 반복문을 실행합니다.  3. 무한 반복문 안에 노드가 null 이 아닐떄 까지 while 문을 수행합니다  4 . 3번의 while 문을 수행하면서, 스택에 노드를 넣습니다.  5. 노드를 넣었으면 노드는 노드가 가진 left 노드로 이동하게 됩니다.  6. 스택이 비지 않았으면, 노드는 스택에 있는 값을 pop 울 수행하여 가져옵니다.  7. 이 노드가 잎노드면, 숫자를 잎노드가 아니면 기호를 출력합니다.  8. 이 작업이 끝났으면, 노드는 노드가 가진 right 로 이동합니다.  9. 만약 스택이 비었으면, 무한 반복문을 종료합니다.  반복으로 후위 탐색 진행 방법  1. 스택을 생성합니다.  2. 방문된 노드를 저장할 visted 노드를 생성합니다.  3. 무한 반복문을 실행합니다  4. 만약 노드가 비지 않았고 방문된 노드가 아닌 경우 노드를 스택에 넣습니다.  5. 스택에 넣었으면, 노드가 NULL 이 아닐때까지 우측 노드와 좌측 노드를 삽입합니다.  6. 삽입이 끝났으면 노드는 좌측 노드로 이동합니다.  7. 왜 그렇게 진행하냐면, 후위탐색은 좌측 우측 루트 이렇게 출력되는데, 스택은 후입선출 방식을 따르므로, 루트 우측 좌측 순으로 삽입하면, 좌측 우측 루트 순으로 출력되기 때문입니다.  8. 4의 if 문이 끝났으면, 스택이 비었나 확인합니다  9. 스택이 비지 않았으면, 노드는 스택에 저장된 값을 반환합니다.  10. 만약 노드의 왼쪽이 NULL 이 아니고 노드의 오른쪽이 NULL 이 아니고 방문된 노드가 아니라면, 우측이 없으므로 좌측 루트 만 출력되야 하므로, 현재 노드를 스택에 넣고 현재 노드는 스택의 좌측으로 이동합니다.  11. 만약 노드의 우측이 NULL 이거나 오른쪽 노드가 방문된 경우, 데이터를 출력합니다.  잎노드면, 숫자 잎노드가 아니면 부호를 출력합니다. 또한 이 노드는 방문된 노드이므로 방문 노드는 현재 노드로 설정합니다.  12. 10번에서 스택이 비었으면 ( else 문) while 문을 종료합니다. |
| Example | 1. 수식이 올바른 경우  2. 수식이 올바르지 않은 경우  3. 괄호쌍이 올바르고 수식이 올바른 경우  4. 괄호쌍이 올바르고 수식이 올바르지 않은 경우  5. 괄호쌍이 올바르지 않고 수식이 올바른 경우  6. 괄호쌍이 올바르지 않고 수식이 올바르지 않은 경우  5 번 항목의 실행결과 및 검증에서 확인 가능합니다. |

3. 의사 코드를 사용하여 알고리즘 작성

괄호쌍 검사 알고리즘

stack <- Stack()

Char LastSymbol // 스트링의 마지막 값

String InputString // 사용자에게 입력 받은 스트링

IsTrue <- True

If (LastSymbol!= '+' && LastSymbol!= '-' && LastSymbol!= '÷'&& LastSymbol!= '\*') then

For I <-0 to LastIndex by i++ do //스트링의 마지막 인덱스 까지 반복

Sym <- InputString[i]

If(stack.is\_empty() and sym = ‘)’ then

IsTrue <-false

Break

Endif

If (sym = ‘)’ ) then

Stack.pop()

Else if (sym = ‘(‘) then

Stack.push(“(“)

Endif

Repeat

Endif

If (!isTrue and !stack.is\_empty() )then

Print(“괄호가 올바르지 않습니다”)

Else then

ShowString <- “”

Stack <- Stack()

For I <-0 to LastIndex by i++ do

Sym <- InputString[i]

If sym = ‘)’ then

While( (left = stack.pop() )!= ‘(‘) do

showString <- ShowString + sym

Repeat

Else if (pie (sym) == -1) then

showString <- ShowString + sym

else

while(!stack.is\_empty() and pis(stack.peek()) >= pie(sym) )do

showString <- ShowString + sym

repeat

stack.push(sym)

endif

repeat

while(!stack.is\_empty()) do

ShowString <- ShowString + stack.pop()

Repeat

Stack <- Stack()

Strings <- ShowString.split(“ “) //공백을 기준으로 자른다

For I <- 0 to String.length i++ do

If I = 숫자 then

Stack.push(i)

Else then

Left <- stack.pop

Now <- stack.pop

Switch(i)

"+" then stack.push(now + left) //현재 i 가 + 면 덧셈 수행후 push

"-" then stack.push(now - left) // 현재 i 가 - 면 뺄셈 수행후 push

"\*" then stack.push(now \*left) //현재 i 가 \* 면 곱셈 수행후 push

"^"then stack.push(now^left) //현재 i 가 ^면 제곱 수행후 push

"÷" -> stack.push(now/left) //현재 i 가 ÷ 면 나누기 수행후 pus

endSwitch

endif

print(stack.pop())

1. Kotlin 으로 작성한 프로그램

------------------------------------------------Calculate.kt--------------------------------------

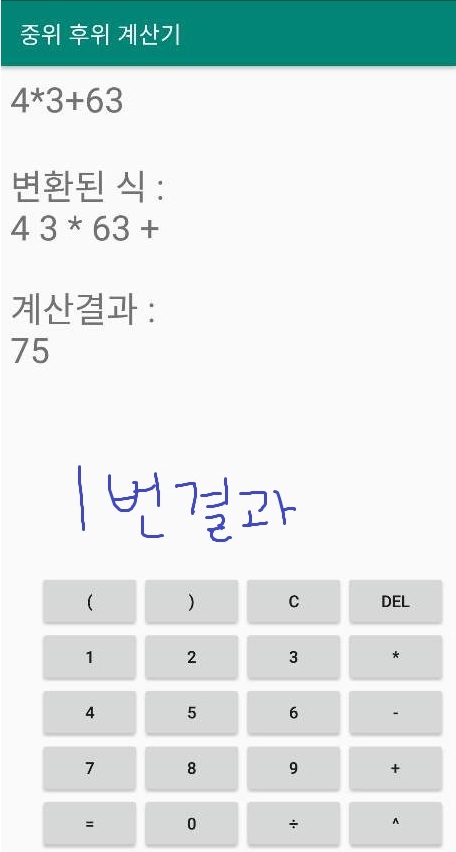
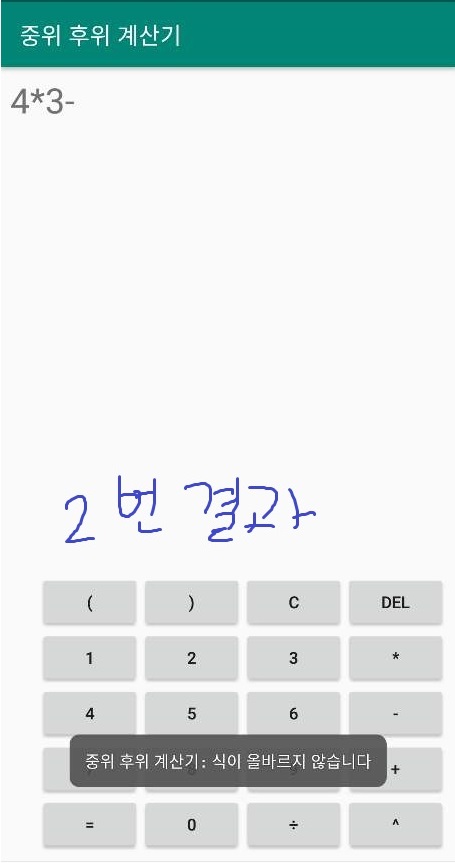
**package** am.ze.wookoo.infixnotation  
  
**import** android.support.v7.app.AppCompatActivity  
**import** android.os.Bundle  
**import** android.util.Log  
**import** android.widget.Toast  
**import** kotlinx.android.synthetic.main.activity\_calculate.\*  
**import** kotlinx.android.synthetic.main.activity\_calculate.view.\*  
  
**class** Calculate : AppCompatActivity() {  
  
 **override fun** onCreate(savedInstanceState: Bundle?) {  
 **super**.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_calculate*)  
 **val** Numbers = *arrayOf*(button0,button1,button2,button3,button4,button5,button6,button7,button8,button9) *//0~9 버튼을 배열로 만든다* **for** (i **in** Numbers){ *//0~ 9버튼을 눌렀을때 하는 일 for 문으로 돌려버림* i.setOnClickListener **{** *//클릭 리스너 설정* **var** StringTemp:String = Display.*text*.toString()  
  
 **if** (StringTemp.**length** <20 ){ *//직접 지정한 숫자가 20 이 안 넘었을 경우* Display.*text* = Display.*text*.toString() + i.*text //누른 버튼에 맞는 숫자를 띄워준다.* }  
  
 **}** }  
  
  
  
 buttonBracketEnd.setOnClickListener **{** *// ) 버튼을 눌렀을때 할 작업* **var** StringTemp:String = Display.*text*.toString()  
 **if**(Display.*text*.toString().**length** <20){  
 **try**{ *//try 문을 사용하여 StringTemp.lastIndex 가 -1 이 아닐 경우에 수행된다.* **val** X = StringTemp[StringTemp.*lastIndex*] *//마지막 문자를 X 변수에 저장한다.* **when**(X){  
 **'0'**,**'1'**,**'2'**,**'3'**,**'4'**,**'5'**,**'6'**,**'7'**,**'8'**,**'9'**,**'('**,**')'**-> Display.*text* = Display.*text*.toString() + buttonBracketEnd.*text  
 //마지막 변수가 위와 같다면, 즉, 기호가 아니라면 누른 버튼에 맞는 ) 를 띄워준다.* }  
 }**catch** (e : Exception) {*//StringTemp.lastIndex 가 -1 일때 수행된다* Display.*text* = Display.*text*.toString() + **")"** *//사용자가 누른 ) 를 띄워준다.* }  
 }  
 **}** buttonBracketStart.setOnClickListener **{** *//( 버튼을 눌렀을때 작업* **var** StringTemp:String = Display.*text*.toString()  
 **if**(Display.*text*.toString().**length** <20) {  
 **try** {*//try 문을 사용하여 StringTemp.lastIndex 가 -1 이 아닐 경우에 수행된다.* **val** X = StringTemp[StringTemp.*lastIndex*]  
 **when** (X) {  
 **'+'**, **'-'**, **'÷'**, **'\*'** ,**'('**,**')'**-> Display.*text* = Display.*text*.toString() + buttonBracketStart.*text  
 //마지막 변수가 위와 같다면, 즉, 마지막 변수가 숫자가 아니라면, 누른 버튼에 맞는 (를 띄워준다.* }  
 } **catch** (e: Exception) {*//StringTemp.lastIndex 가 -1 일때 수행된다* Display.*text* = Display.*text*.toString() + **"("** *//사용자가 누른 ( 를 띄워준다.* }  
 }  
 **}  
  
  
 val** Signs = *arrayOf*(buttonPlus,buttonMinus,buttonMultiply,buttonDivde,buttonSquare) *//기호 버튼을 눌렀을때 공통된 작업을 하므로 배열로 만든다.* **for** (i **in** Signs){ *//for each 문으로 배열 각각 요소인 버튼에 접근한다* i.setOnClickListener **{** *//버튼마다 작업을 지정해준다.* Log.d(**"기호 버튼"**,**"${**i.*text***}"**) *//디버그용 로그를 출력* **var** StringTemp = Display.*text*.toString() *//띄워져있는 text 를 가져온다.* **if** (StringTemp.**length**> 0 && StringTemp.**length** <20 ){ *//띄워져있는 text 의 길이가 0보다 크고 내가 설정한 20보다 작으면 수행한다.* **val** X = StringTemp[StringTemp.*lastIndex*]  
 **when**(X){  
 **'+'**,**'-'**,**'÷'**,**'\*'** -> Display.*text* = Display.*text*.toString().*substring*(0,StringTemp.*lastIndex*) + i.*text*.toString()  
 *//마지막 변수가 위와 같으면, 즉, 마지막 변수가 기호면  
 //예를들어 이전이 456+ 고 내가 누른게 - 면, 456- 로 바뀐다.* **'('** -> Display.*text* = Display.*text //이전이 (면 괄호 시작에 기호가 올수 없으므로 아무 작업도 하지 않는다.* **else** -> Display.*text* = Display.*text*.toString() + i.*text //그 외의 경우는 ) 나 숫자인 경우이므로 누른 버튼을 추가해준다.* }  
 }  
 **}** }  
  
 buttonClear.setOnClickListener **{** *//클리어 버튼누르면* Display.*text* = **""** *//디스플레이의 텍스트 초기화* Changed.*text* = **""** Toast.makeText(**this**,**"값이 삭제되었습니다"**,Toast.*LENGTH\_SHORT*).show() *//값이 삭제됬음을 띄워준다.* **}** buttonDel.setOnClickListener **{** *//Del 버튼을 누르면* **if**(Display.*text*.toString().*lastIndex* >=0){ *//글자가 1개 이상이면* Display.*text* = Display.*text*.toString().*substring*(0,Display.*text*.toString().*lastIndex*) *//스트링을 하나 날려버린다* }  
 Changed.*text* = **""  
 }** buttonEqual.setOnClickListener **{** *//계산 결과 버튼을 누르면 할 작업* **val** BracketStack = Stack() *//괄호쌍 검사를 위한 Stack 객체 생성* **val** DisPlayString = Display.*text*.toString() *//사용자가 입력한 String 들* **val** LastIndex = DisPlayString.*lastIndex* **var** IsTrue = **true** *//괄호가 올바른지. 기본값은 true* **var** lastSymbol = DisPlayString[LastIndex]  
  
 **if** (LastIndex != -1 && lastSymbol != **'+'** &&lastSymbol != **'-'** && lastSymbol != **'÷'**&& lastSymbol != **'\*'** ){ *//스트링에 값이 있으면* **for**(i **in** 0..LastIndex){ *//for each 문으로 스트링의 요소 하나하나를 가져온다* **val** sym = DisPlayString[i]  
 **if**(BracketStack.is\_empty() && sym == **')'**){ *//스택이 비었는데 들어온 sym 이 ) 면* IsTrue = **false** *//그 괄호쌍은 올바르지 않으므로* **break** *//for 문 종료* }  
 **if**(sym == **')'**){ *//스택이 비지 않았고, 들어온 sym 이 ) 면* BracketStack.pop() *//pop 을 수행한다.* }  
 **else if**(sym ==**'('**){ *//sym 값이 ( 인 경우 일단 값을 넣어본다.* BracketStack.push(3) *//아무값이나 push 를 수행한다.* }  
  
 }  
 **if**(!IsTrue || !BracketStack.is\_empty()){ *//isTrue 가 false 거나, BracketStack 이 비지 않았을 경우* Toast.makeText(**this**,**"괄호가 올바르지 않습니다"**,Toast.*LENGTH\_SHORT*).show()  
 *//괄호쌍이 올바르지 않을 경우 출력* }  
 **else**{  
 *//괄호가 올바를 경우 할 작업 > 1. 중위식 후위식 변환* **var** ShowString =**""** *//사용자에게 변환된 중위식을 저장할 String 변수* **val** ChangeStack = Stack() *//중위식을 후위식으로 변환 하는 것은 Stack 을 통해 구현을 한다.* **for**(i **in** 0..LastIndex){ *//0~ 마지막 인덱스 만큼 for 문을 반복한다.* **val** sym = DisPlayString[i] *//sym 변수는 DisPlayString 의 i 번째 index와 같다.* **if** (sym == **')'**){ *//sym 이 ) 인 경우* **var** left:Char *//left 변수는 char 로 설정한다. 값 대입이 없으면, Kotlin 컴파일러가 무슨 타입인지 예측할수 없으므로, Char 로 이번엔 설정해준다.* **while**(**true**){ *//코틀린은 while((변수=할당) == (조건)) 문이 수행이 안된다 그렇기 때문에 while True 로 수행한다.* left = ChangeStack.pop().toChar() *//pop 을수행한 값을 left 에 넣는다.* **if**(left == **'('**){ *//만약 left 가 ( 면* **break** *//while 이 종료된다.* }  
 ShowString += **" ${**left**}"** *//그게 아니면 left 를 String 으로 변환후 ShowString 에 추가한다. 그전에, 앞에 공백을 추가하여 숫자와 기호를 구분한다.* }  
  
 }  
 **else if** (pie(sym) == -1){ *//sym 의 pie 를 수행한 값이 -1 이면 숫자이므로* ShowString += sym.toString()*//ShowString 에 추가한다.* }  
 **else**{ *//그 외의 경우* **if**(sym != **'('** && sym != **')'**){ *//그 외의 경우 중 ( 와 ) 가 아니면* ShowString += **" "** *//기호와 숫자를 구분해야 하므로 ShowString 에 " "로 공백을 추가한다.* }  
 **while**(!ChangeStack.is\_empty() && pis(ChangeStack.peek().toChar() )>= pie(sym)){ *//스택이 비지 않았고, 연산자 우선순위등을 고려한다* ShowString += **"${**ChangeStack.pop().toChar()**} "** *//연산자 또는 숫자는 공백으로 비교하므로 마지막에 " " 를 추가한다.* }  
  
 ChangeStack.push(sym.toInt()) *//while 문이 끝났으면, 현재 연산자를 push 를 하여 스택에 저장한다.* }  
  
 }  
 **while**(!ChangeStack.is\_empty()){ *//for 문이 끝났으면, 스택이 빌때 까지 반복한다.* ShowString += **" ${**ChangeStack.pop().toChar()**}"** *//공백 문자를 통해 연산자가 끝남을 확인시켜준다.* }  
 Log.d(**"변환된 식"**,ShowString) *//디버그용 출력  
  
 //이제 결과값을 띄워주면된다.  
  
 //2. 중위식 후위식 평가진행하면 됨.* **val** result = CaculatePostfix(ShowString) *//중위식 평가하는 함수를 호출한다. , 위 작업을 통하여 ShowString 은 입력된 중위식을 후위식으로 바꾼 상태다.* Changed.*text* = **"변환된 식 : \n$**ShowString**\n\n계산결과 :\n$**result**"** }  
 }  
 **else**{*// String 의 끝에 들어있는 부호가 계산 기호일 경우* Toast.makeText(**this**,**"식이 올바르지 않습니다"**,Toast.*LENGTH\_SHORT*).show() *//Toast Message 로 식이 올바르지 않다는것을 보여준다.* }  
 **}** }  
  
 **fun** pis(sym: Char): Int { *//중위식 , 후위식을 변환할때 확인하는 함수, 스택 안의 값을 확인해본다.* **when** (sym) {  
 **'('** -> **return** 0 *//190 번째줄과 return 순위가 달라져있다.* **'^'** -> **return** 4 *//제곱은 항상 제일 먼저 수행하게 된다.* **')'** -> **return** 3  
 **'+'**, **'-'** -> **return** 1  
 **'\*'**, **'%'**, **'÷'** -> **return** 2  
 }  
 **return** -1  
 }  
  
 **fun** pie(sym: Char): Int { *//중의식, 후위식을 변환할떄 확인하는 함수, 스택 안의 값을 확인해본다.* **when** (sym) {  
 **'('** -> **return** 3  
 **'^'** -> **return** 4  
 **')'** -> **return** 0  
 **'+'**, **'-'** -> **return** 1  
 **'\*'**, **'%'**, **'÷'** -> **return** 2  
 }  
 **return** -1  
 }  
  
 **fun** CaculatePostfix(tempString : String):Int { *//생성된 후위식을 출력하는 함수이다.* **val** stack = Stack() *//스택 객체를 생성해준다.* **val** Strings = tempString.*split*(**" "**) *//파라메터로 받아온 tempString 의 공백을 기준으로 split 을 한다.  
 //스플릿이 된 값은 베열형태로 반환되서 Strings 에 저장된다* **for** (i **in** Strings){ *//Kotlin 에서 지원하는 for each 문을 사용한다.* Log.d(**"스트링스"**,**"작업중인 값 : ${**i**}"**) *//디버그를 해보기 위해 Log.d 메소드를 호출하여 로그를 찍어본다.  
 // for each 문으로 돌리는 i 값이 출력된다.* **try**{ *//toInt 그러니까, i를 숫자로 바꿀수 있으면* **var** temp = i.*toInt*()*//자바에서 Integer.parseInt(String) 과 같다. > 스트링을 숫자만 추출하고* stack.push(temp) *//그 값을 stack 에 넣는다.* Log.d(**"인트형으로 바꾼 것"**,**"$**temp**"**)  
 }  
 **catch** (e : Exception){ *//catch 문에선 String 형태의 기호를 Int 형으로 못바꾼 것들이 실행된다.* Log.d(**"인트형으로 못 바꾼것"**,**"$**i**"**)  
 *//이 아래는 적절히 후위식 평가를 진행하는 코드다.* **val** left = stack.pop() *//pop 을 두번 수행하여 left 와 now 를 가져온다.* **val** now = stack.pop()  
 **when**(i){  
 **"+"** -> stack.push(now + left) *//현재 i 가 + 면 덧셈 수행후 push* **"-"** -> stack.push(now - left) *// 현재 i 가 - 면 뺄셈 수행후 push* **"\*"** -> stack.push(now \*left) *//현재 i 가 \* 면 곱셈 수행후 push* **"^"**-> stack.push(Math.pow(now.toDouble(), left.toDouble()).toInt()) *//현재 i 가 ^면 Math 메소드의 pow 함수를 사용하여 제곱을 구한 후 push* **"÷"** -> stack.push(now/left) *//현재 i 가 ÷ 면 나누기 수행후 push* }  
  
 }  
  
 }  
 **return** stack.pop() *//for 문이 끝났으면 pop 을 수행하여 반환한다.* }  
  
}

---------------------------------------------Caculate.kt 끝------------------------------------

---------------------------------------------Stack.kt-------------------------------------

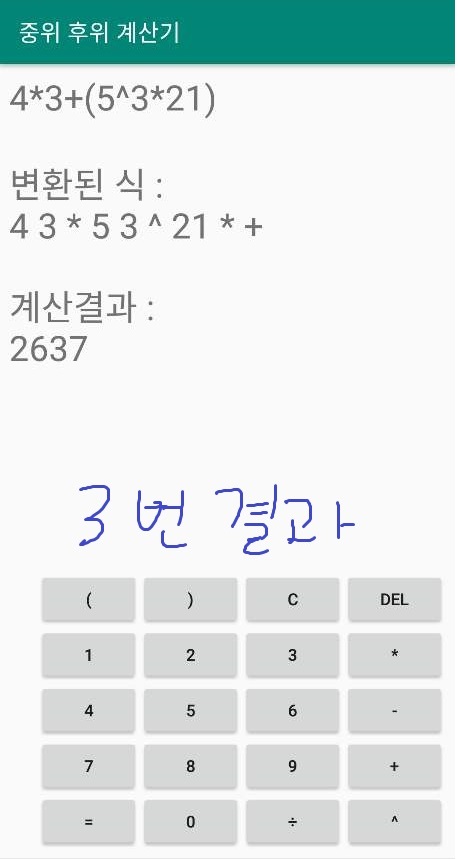
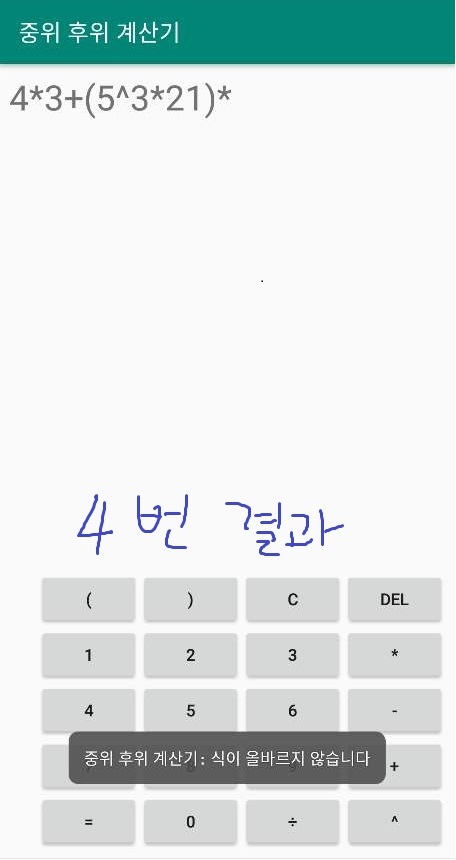
**package** am.ze.wookoo.infixnotation  
  
**class** Stack{  
 **var top** = -1  
 **var data** = ArrayList<Int>()  
  
  
 **fun** is\_empty():Boolean{ *//값이 비었는지 확인* **return this**.**top** == -1   
 }  
 **fun** push(n:Int){  
  
 **this**.**data**.add(n)  
 **this**.**top**+=1  
 }  
 **fun** pop():Int{  
 **if**(!**this**.is\_empty()){ *//값이 안비었으면* **this**.**top**--; *//top 을 감소하고* **val** temp = **this**.**data**.get(**this**.**data**.*lastIndex*) *//마지막 인덱스의 요소를 가져오고 저장한다음* **this**.**data**.removeAt(**this**.**data**.*lastIndex*) *//마지막 인덱스의 값을 지우고* **return** (temp) *//반환한다* }  
 **return** -1; *//안비었으면 -1 을 반환한다* }  
 **fun** peek():Int{  
 **return this**.**data**.get(**this**.**top**) *//마지막 인덱스의 값을 반환한다.* }  
  
}

1. 실행 결과 및 검증



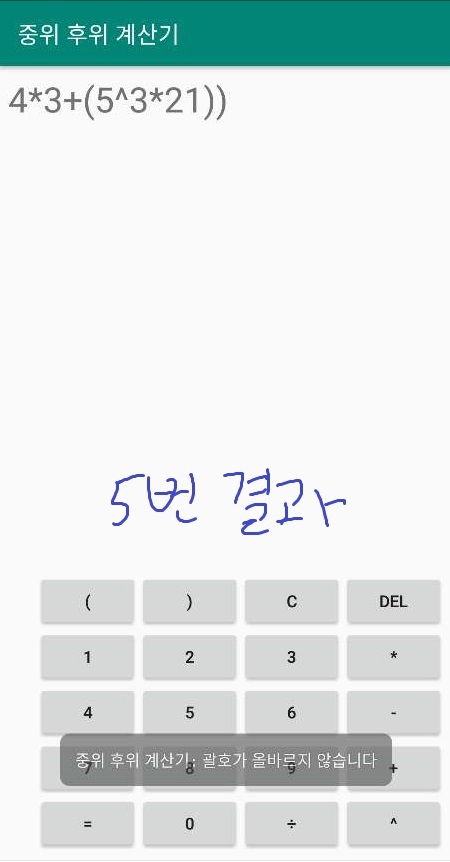
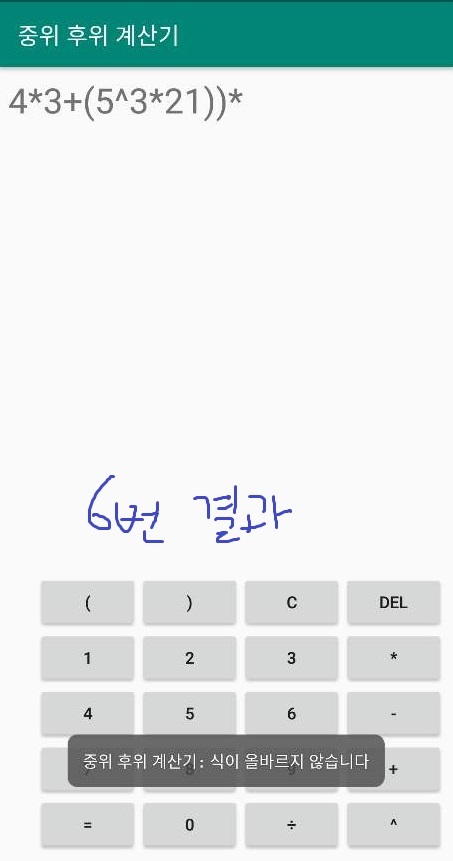
1번 경우인 수식이 올바른 경우는, 정상적으로 후위식 변환 및 평가가 이루어진다.

2번 경우인 수식이 올바르지 않는 경우는, 식이 올바르지 않다는 에러 메시지를 보여줌을 알 수 있다.



3번 경우인 괄호쌍이 올바르고 수식이 올바른 경우엔, 정상적으로 후위식 변환 및 평가가 이루어진다.

4번 경우인 괄호쌍이 올바르고 수식이 올바르지 않은 경우, 식이 올바르지 않다는 에러가 출력된다.



5번 경우인 괄호쌍이 올바르지 않고 수식이 올바른 경우엔, 괄호가 올바르지 않다는 에러 메시지가 출력된다.

6번 경우인 괄호쌍이 올바르지 않고 수식이 올바르지 않는 경우엔, 식이 올바르지 않다는 에러 메세지가 출력된다.

1 ~ 6 번 경우를 보면서, 올바르지 않았을 때 작동이 되는 경우는 없고, 올바른 경우에서 작동이 안되는 결과는 존재하지 않았다. 즉, 정상적으로 알고리즘이 구현됬음을 보여준다.

1. 의견

신생 언어인 Kotlin 을 공부하기 위한 목적으로 소스를 짜서 아직 내가 봐도 미흡한 소스인 것 같다.

Strong type 의 언어가 아니기 때문에, 변수 선언이 비교적 쉬웠지만 역효과로 변수의 타입을 까먹을 때도 있다. 또한, Kotlin에서 지원하는 ArrayList 로 보다 간편하게 동적 Stack 을 구현할수 있었다.

자바로 컴파일 했을때와 Kotlin 으로 컴파일 했을 때 속도차이가 얼마나 나나 궁금해서검색을 해봤는데, 속도는 비슷하다고 했다. Kotlin 이 애초에 긴 java 문법을 대체하기 위한 ( 사실 오라클과 구글의 분쟁 때문이 더 맞는 표현인거 같긴 하지만.. ) 목적으로 개발되었는데, 컴파일 속도가 비슷하다는것에 놀라 찾아보니, 자바의 컴파일 속도가 미친듯이 빠른것이라고 하였다.. 참으로 허무하였다.

절차지향 언어도 좋지만, 여러가지 스택을 생성하고, 메소드를 묶고 기타 편한 작업을 하기 위해선 개인적인 생각이지만, 객체 지향 언어가 조금 더 쉽게 구현이 가능하다 생각된다. 다만 속도는 보장할수 없는 문제가 있긴 하지만 말이다..

여담으로 제작한 소스 전체 코드는 <https://github.com/wookoo/InfixNotation> 에서 확인 하실 수 있습니다.

아래의 QR 을 스캔하시면 위의 링크로 이동됩니다.



<< 원본 소스 및 프로젝트로 이동하는 QR 코드

또한 업로드된 어플리케이션은 <https://play.google.com/store/apps/details?id=am.ze.wookoo.infixnotation> 에서 확인 하실 수 있습니다.

아래의 QR 을 스캔하시면 위의 링크로 이동됩니다.

플레이스토어에서 확인하는 QR 코드 >>