# 보고서

## - 직접 조작 계산기 -

| | | |
|---|---|---|
| 과목명 | | 휴먼컴퓨터인터페이스 |
| 담당교수 | | 이강훈 |
| 제출일 | | 2019년 5월 25일 |
| 소속 | | 소프트웨어학부 |
| 학번 | | 2015202065 |
| 이름 | | 윤홍찬 |

광운대학교
KwangWoon University

# < 목 차 >

## 1 - 1. 기능적 요구조건에 대한 구현 완성도 요약

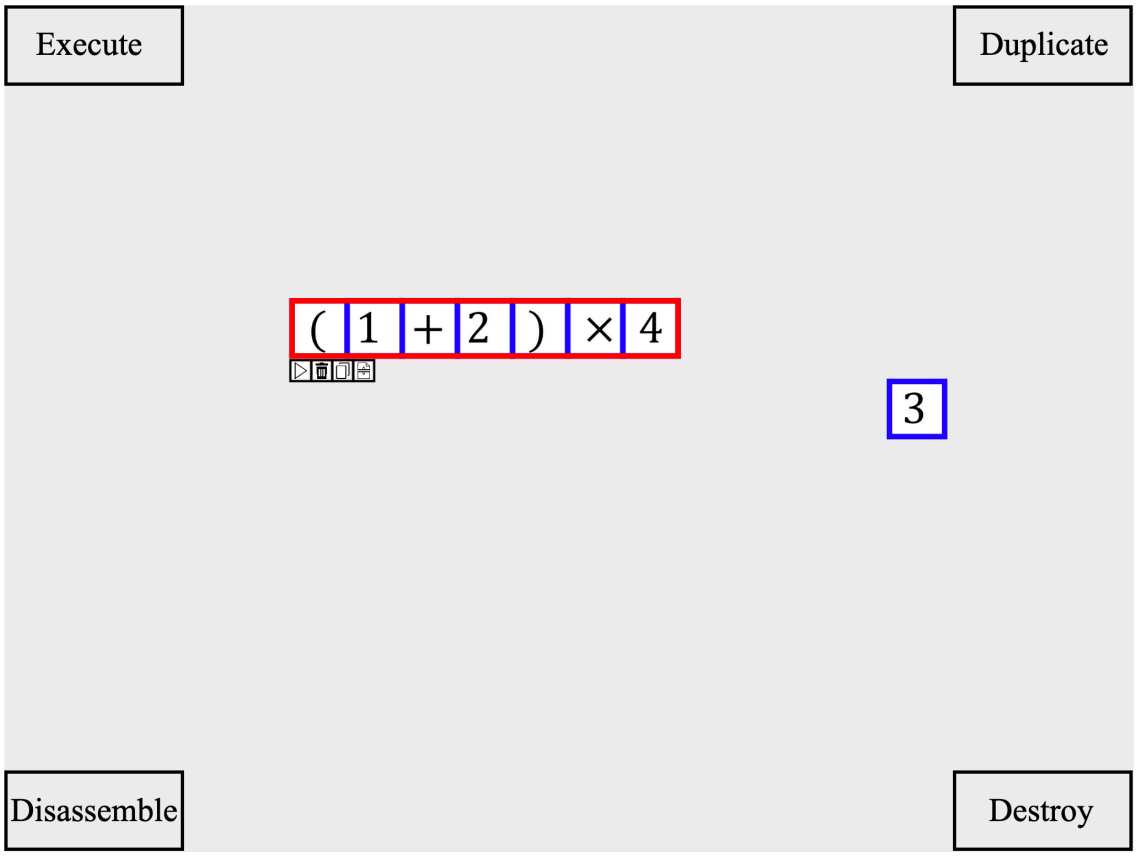
| 분류 | 기능 | 구현 |
|---|---|---|
| 요구 조건 | **정수, 실수, 복소수의 표현과 그 기본 연산**<br>- 산술연산(+, -, *, /, %, ^), 비교연산(==, !=, >, <, >=, <=) | O |
| | **벡터, 행렬의 표현과 그 기본 연산**<br>- 백터: 내적(n 차원), 외적(3 차원) / 행렬: 곱셈, 역행렬, 행렬식 | O |
| | **자주 사용되는 상수 및 함수 지원**<br>- 상수: pi, e / 함수: sin, cos, tan, exp, log, sqrt | O |
| | **결과 출력**<br>- 올바른 입력 -> 수식의 결과 값 / 잘못된 입력 -> 오류 메시지 | O |
| | **변수, 함수 정의 및 사용**<br>- 변수 : 개수 제한 없음 / 함수 : 개수 제한 없음 | O |

## 1 - 2. 인터페이스 요구조건에 대한 구현 완성도 요약

| 분류 | 기능 | 구현 |
|---|---|---|
| 요구 조건 | **생성**<br>- 키보드 숫자, 문자 키 | O |
| | **이동**<br>- 마우스 드래그 | O |
| | **조립**<br>- 마우스 드래드-앤-드롭 | O |
| | **실행**<br>- 팝업 메뉴 (Canvas 로 구현) | O |
| | **분해**<br>- 팝업 메뉴 (Canvas 로 구현) | O |
| | **복제**<br>- 팝업 메뉴 (Canvas 로 구현) | O |
| | **소멸**<br>- 팝업 메뉴 (Canvas 로 구현) | O |
| 추가 구현 | **실행, 분해, 복제, 소멸**<br>- 직접 조작 방식 | O |

## 2 사용자 인터페이스의 구성 요소 및 사용 방법

다음 사진은 최종적으로 구현한 직접 조작 계산기의 화면이다.

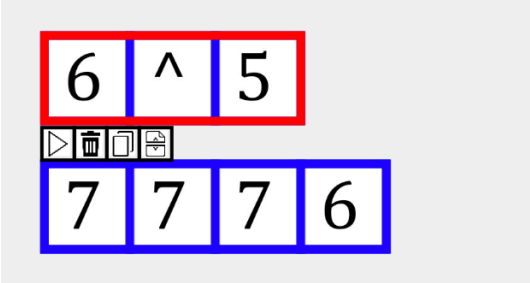


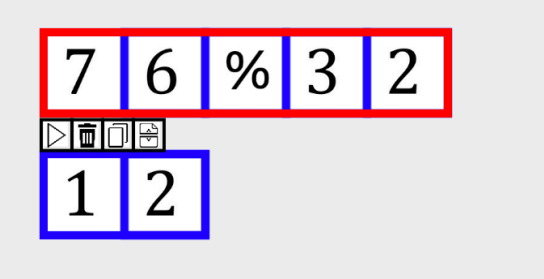

각 세부적인 인터페이스를 설명하기 전 간단히 설명하자면, 사용자는 숫자, 문자 키를 입력하여 블록을 **생성**하고, 블록을 마우스로 끌어 **이동 및 조립**할 수 있으며, 클릭 시 나타나게 되는 팝업 메뉴를 클릭하여 블록 **실행, 분해, 복제, 소멸**을 할 수 있다. 추가적으로, 각 모서리에 **실행, 분해, 복제, 소멸 영역**을 만들어서 해당 연산을 하고자 하는 블록을 각 부분에 **직접 조작 방식**으로 겹치게 두면 해당 연산이 실행된다. 이제 각 인터페이스에 대해 세부적으로 살펴보겠다.

### 1) 기능적 요구조건 추가

인터페이스 각 구성 요소에 대해 설명하기 전, 제일 먼저 간단하게 기능적 요구조건에 대해 미흡한 사항을 추가한 부분에 대해 설명하고자 한다. 과제의 기능적 요구조건을 보면 산술 연산에 **^(거듭제곱)과 %(나머지 연산)을 구현해야 한다고 명시되어 있다.** 예제로 주어진 코드에서는 나머지 연산에 대해서 모두 구현이 완료되

었으나, ^와 %는 구현이 되어있지 않았다. ^는 이미지 파일이 있어 간단히 추가 구현하였지만, %는 이미지 파일 조차 주어지지 않아 직접 이미지를 만들고 기능을 추가하였다. 다음은 이에 대한 구현 결과이다.



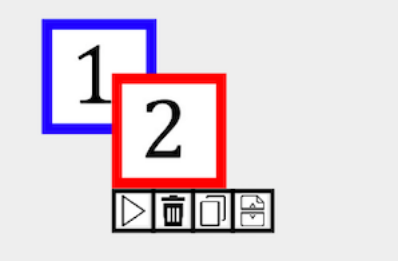

### 2) 생성 및 이동

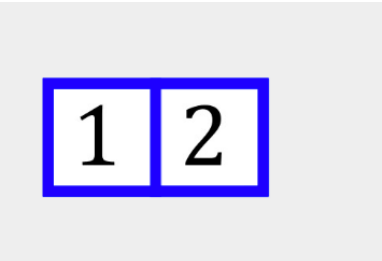
제일 먼저 살펴볼 사용자 인터페이스는 생성 및 이동이다. 먼저 **생성**에 관한 부분이다. 사용자가 블록을 생성하기 위해서는 **키보드 숫자 또는 문자 키를 눌러야 한다**. 생성된 블록은 캔버스 영역내의 무작위 위치에 표시된다.

두 번째로 **이동**이다. 블록을 이동시키기 위해서는 단순히 **블록을 마우스로 끌면 (드래그) 된다**. 만약 블록이 여러 개의 블록으로 이루어져 있다면, 전체의 블록이 함께 이동된다.

### 3) 조립

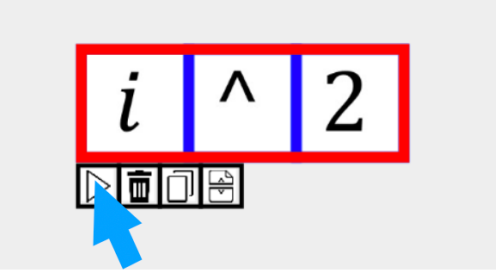
여러 개의 블록을 합치기 위한 **조립** 인터페이스에 대한 설명이다. 블록을 조립하기 위해서는 **블록을 마우스로 끌어(드래그) 결합하고자 하는 블록과 중첩되도록 블록을 이동(드롭)시키면 된다**. 만약 이미 여러 개의 블록으로 구성 되어있는 블록이라면, 어느 곳이 중첩되든 블록의 오른쪽 맨 마지막에 조립된다. 또한 여러 개의 블록으로 구성되어있는 블록들끼리의 결합 또한 가능하다.

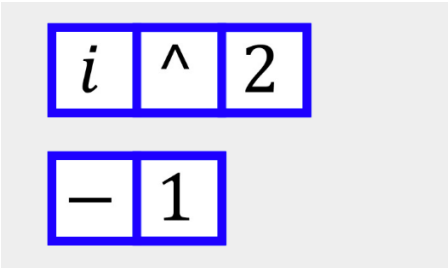


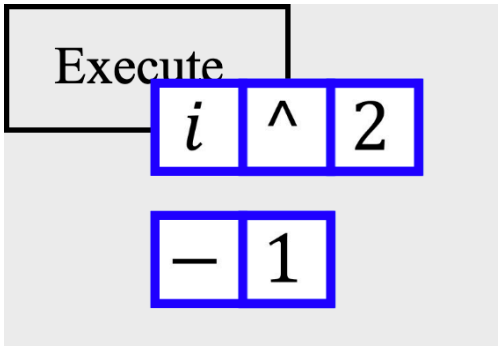
### 4) 실행

**실행**에 대한 설명이다. 블록을 실행하기 위한 인터페이스로 두 가지 방식을 제공하고 있다. **첫 번째는, 블록 클릭 시 나타나는 팝업 메뉴 중 실행 버튼을 클릭하는 것**이고, **두 번째는 직접 조작 방식으로 블록을 실행 영역에 끌어 놓는 것**이다. 다음은 첫 번째 방법이다.



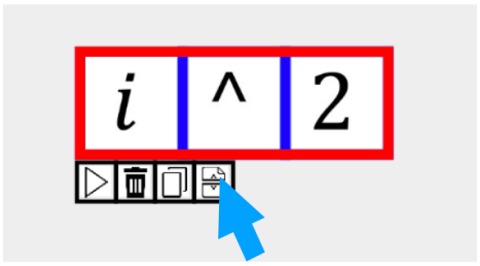

왼쪽 사진과 같이 블록 클릭 시 나타나는 팝업 메뉴에서 첫 번째인 실행 버튼을 클릭하면 오른쪽과 같이 실행 결과가 블록 바로 밑에 생성된다. 이어서 두 번째 방법이다.

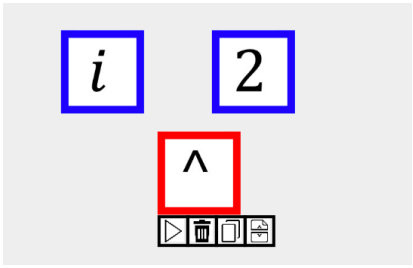



이는 단순히 실행하고자 하는 블록을 끌어 왼쪽 위 모서리의 실행 영역에 놓음으로써 블록의 연산이 실행되어, 바로 밑에 새로운 블록이 생성된다. 이는 팝업 메뉴보다 더 간단하고 빠르게 작업을 수행할 수 있다.

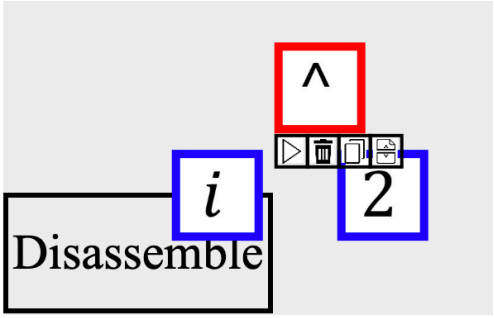
### 5) 분해

**분해** 또한 두 가지 인터페이스로 제공하고 있다. 첫 번째는 실행과 마찬가지로 **블록 클릭 시 나타나는 팝업 메뉴 중 분해 버튼을 클릭하는 것**이고, **두 번째는 직접 조작 방식으로 블록을 분해 영역에 끌어 놓는 것**이다. 다음은 첫 번째 방법이다.





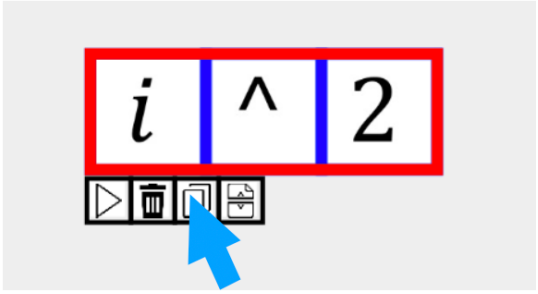

왼쪽 사진과 같이 블록 클릭 시 나타나는 팝업 메뉴에서 네 번째 분해 버튼을 클릭하면 분해가 된다. 따라서 오른쪽과 같이 블록을 개별적으로 조작할 수 있는 모습을 볼 수 있다. 이어서 두 번째 방법이다.

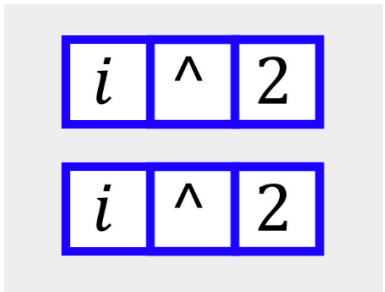



단순히 분해하고자 하는 블록을 끌어 왼쪽 아래 모서리의 분해 영역에 놓음으로써 블록이 분해되어 개별적으로 조작할 수 있게 된다.

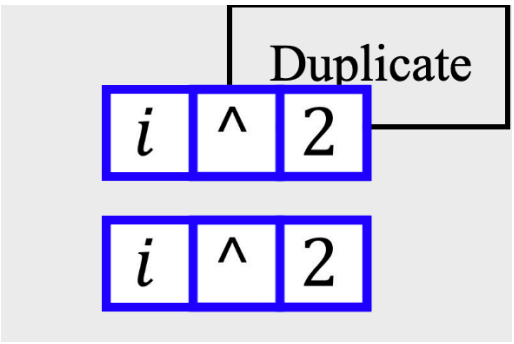
### 6) 복제

복제도 두 가지의 인터페이스를 제공한다. 첫 번째는 **블록 클릭 시 나타나는 팝업 메뉴 중 복제 버튼을 클릭하는 것**이고, **두 번째는 직접 조작 방식으로 블록을 복제 영역에 끌어 놓는 것**이다. 다음은 첫 번째 방법이다.





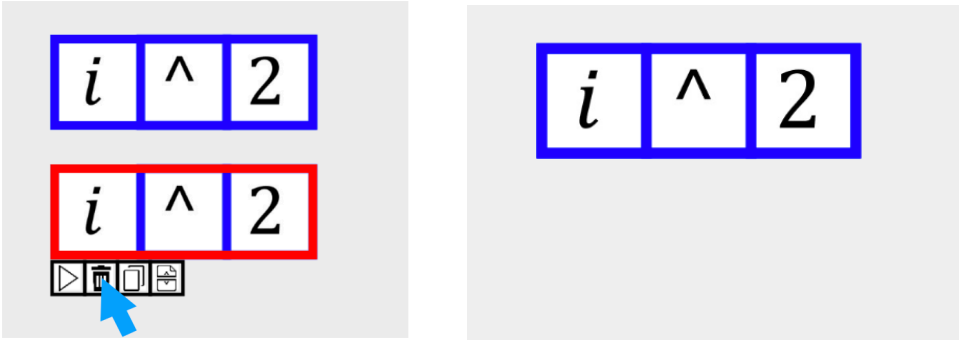

왼쪽 사진과 같이 블록 클릭 시 나타나는 팝업 메뉴에서 세 번째인 복제 버튼을 클릭하면 오른쪽과 같이 복제된 블록이 바로 밑에 생성된다. 이어서 두 번째 방법이다.




단순히 복제하고자 하는 블록을 끌어 오른쪽 위 모서리의 복제 영역에 놓음으로써 블록 바로 밑에 새롭게 블록이 복제 된 것을 볼 수 있다.

### 7) 소멸

소멸도 위의 것 들과 마찬가지로 두 가지의 인터페이스를 제공한다. 첫 번째는 **블록 클릭 시 나타나는 팝업 메뉴 중 소멸 버튼을 클릭하는 것**이고, **두 번째는 직접 조작 방식으로 블록을 소멸 영역에 끌어 놓는 것**이다. 다음은 첫 번째 방법이다.




왼쪽 사진과 같이 블록 클릭 시 나타나는 팝업 메뉴에서 두 번째인 소멸 버튼을 클릭하면 오른쪽과 같이 해당 블록이 삭제 된 것을 볼 수 있다. 이어서 두 번째 방법이다.


단순히 삭제하고자 하는 블록을 끌어 오른쪽 아래 모서리의 소멸 영역에 놓음으로써 블록을 제거할 수 있다. 왼쪽 사진은 블록을 끌어 소멸 영역에 놓기 전(마우스의 버튼을 떼기 전)을 캡처한 것이고, 오른쪽 사진을 보면 마우스의 버튼을 떼자 블록이 소멸된 것을 볼 수 있다.

## 3. 특징적인 상호작용 방식들에 대한 세부 구현 방법

### 1) 생성 및 이동

사용자가 키보드 숫자 또는 문자 키를 눌러 블록을 **생성**하는 것과 생성된 블록을 마우스로 끌어(드래그) **이동**하는 상호작용 방식은 예제 코드로 주어진 것과 같이 구현하였다. 따라서 이에 대한 세부 구현 방법은 생략하겠다.

### 2) 조립

블록이 조립되는 과정을 살펴보면, 제일 먼저 드래그를 마쳤을 때(마우스를 땠을 때) 겹치는 블록이 있는지 검사한 후 겹치는 블록이 있으면, 이를 하나로 합쳐야 한다. 이를 구현한 코드를 순서대로 살펴보겠다.

```
302         let overlapBlock = MathApp.findOverlapBlock();
303         if(overlapBlock != null){
304             // 블록이 겹쳐진 상태
305             // 인자로 들어오는 왼쪽 블록에 오른쪽 블록을 결합
306             MathApp.assembleBlock(overlapBlock, MathApp.selected_block);
307         }
308     }
```

제일 먼저 마우스를 땠을 때 처리되는 이벤트 핸들러인 MathApp.handleMouseUP() 메소드내의 코드 일부분을 위와 같이 작성하였다. 302 행에서 호출하는 **MathApp.findOverlapBlock()** 메소드는 현재 선택 중인 블록과 중첩되는 블록 객체를 반환한다. 303 행 ~ 307 행은 겹치는 블록이 있으면 수행되는 부분으로, 306 행의 **MathApp.assembleBlock()** 메소드는 왼쪽으로 전달되는 블록에 오른쪽으로 전달되는 블록을 합친다. 이제 위의 두 메소드를 차례대로 살펴보겠다.

```
371         for(let i = 0; i < this.blocks.length; i++)
372         {
373             let block = this.blocks[i];
374 
375             // 선택한 블럭이라면 넘어감
376             if(block == MathApp.selected_block)
377                 continue;
378 
379             if( x - width/2 >= block.position.x - block.size.width/2 &&
380                 x - width/2 <= block.position.x + block.size.width/2 &&
381                 y - height/2 >= block.position.y - block.size.height/2 &&
382                 y - height/2 <= block.position.y + block.size.height/2 ){
383                 // 왼쪽 위
384                 return block;
385             }
```

위의 코드는 **MathApp.findOverlapBlock()** 메소드의 일부분이다. **이 메소드는 현재 선택 중인 블록과 중첩되는 블록을 찾아 반환**한다. 371 행 반복문으로 시작하여 모든 블록 객체와 비교를 하게 되고, 379 행 ~ 385 행은 블록 간 경계 검사를 하여 중첩되는지 검사를 하게 된다. 이는 왼쪽 위 x, y 좌표에 대해서만 검사를 하며, 왼쪽 아래, 오른쪽 위, 오른쪽 아래 모든 부분에 대해 같은 작업을 수행하며, 코드의 중복이 있어 이 부분은 설명을 생략하였다.

```
576         // 인자로 들어오는 왼쪽 블록에 오른쪽 블록을 결합
577         MathApp.assembleBlock = function(leftBlock, rightBlock){
578             let rightCnt = rightBlock.blockCnt;
579             let beforeX = leftBlock.position.x;
580             let beforeWidth = leftBlock.size.width;
581 
582             // 왼쪽 블록에 이름, 위치, 크기 새롭게 지정
583             leftBlock.blockCnt += rightCnt;
584             leftBlock.name += rightBlock.name;
585             leftBlock.position = {
586                 x : leftBlock.position.x + rightBlock.size.width / 2,
587                 y : leftBlock.position.y
588             };
589 
590             leftBlock.size = {
591                 width : leftBlock.size.width + rightBlock.size.width,
592                 height : leftBlock.size.height
593             };
```

다음으로 살펴볼 코드는 **MathApp.assembleBlock()** 메소드 내의 코드이다. **이 메소드는 인자로 두 개의 블록 객체를 받아 왼쪽 인자로 들어온 블록 객체에 오른쪽 인자로 들어온 블록 객체를 합친다.** 582 행 ~ 593 행을 보면 블록을 합치기 위해 왼쪽 블록의 이름, 위치, 크기, 블록의 개수 등을 새롭게 지정한다.

```
595             // 왼쪽 블록의 전체 boundary 제거
596             leftBlock.visual_items.forEach(item => {
597                 if(item.boundary){
598                     MathApp.canvas.remove(item);
599 
600                     let index = leftBlock.visual_items.indexOf(item);
601                     if(index > -1)
602                     {
603                         leftBlock.visual_items.splice(index, 1);
604                     }
605                 }
606             })
```

이후 블록을 합치게 되면, 전체 경계선이 새로 그려져야 하므로, 왼쪽 블록의 **기존 경계선을 찾아** 화면에서 해당하는 요소를 **제거**하고, visual_items 배열에서도 제거를 한다.

```
608         // 오른쪽의 시각적 요소를 왼쪽에 결합 및 캔버스 출력
609         var i = 0;
610         var additionAdd = 0;
611         rightBlock.visual_items.forEach(item => {
612             i++;
613             if(item.boundary){
614                 // 전체 bondary일 경우 무시
615                 return;
616             }
617             else if(i == 4){
618                 i = 1;
619                 additionAdd += SYMBOL_WIDTH;
620             }
621 
622             if(item.image){
623                 // 이미지 요소일 경우
624                 item.left = beforeX + beforeWidth / 2 + SYMBOL_WIDTH / 2 - img_w / 2 + additionAdd;
625             }
626             else{
627                 item.left = beforeX + beforeWidth / 2 + additionAdd;
628             }
629             item.top = leftBlock.position.y - leftBlock.size.height / 2;
630 
631             leftBlock.visual_items.push(item);
632             MathApp.canvas.add(item);
633         })
```

이후 블록을 결합하기 위해 **오른쪽 블록의 시각적 요소(visual_items)를 화면에 출력(캔버스에 추가) 및 왼쪽 블록 시각적 요소에 추가**한다. 이때, 오른쪽 요소의 전체 경계선은 무시를 하고 (613 행), 한 블록당 시각적 요소의 개수가 3 개(경계선, 사진, 배경)임을 감안하여, 617 행 ~ 619 행에서 i 변수를 사용해 additionAdd 에 블록 너비를 누적해 나가면서 622 행 ~ 629 행에서 적절한 위치를 설정하고 캔버스에 추가 및 왼쪽 블록 시각적 요소에 추가를 한다.

```
635             // 전체 boundary 생성과 추가 및 캔버스 출력
636             let boundary = new fabric.Rect({
637                 left: leftBlock.position.x - leftBlock.size.width/2,
638                 top: leftBlock.position.y - leftBlock.size.height/2,
639                 width: leftBlock.size.width,
640                 height: leftBlock.size.height,
641                 fill: "rgba(0,0,0,0)",
642                 stroke: "rgba(0,0,255,1)",
643                 strokeWidth: 5,
644                 selectable: false,
645                 image: false,
646                 boundary: true
647             });
648             leftBlock.visual_items.push(boundary);
649             MathApp.canvas.add(boundary);
650 
651             // select 블록 삭제
652             rightBlock.destroy();
653         }
```

조립의 마지막 과정이다. 이제 결합된 블록의 전체 경계선을 추가하고(635 행 ~ 649 행), 오른쪽 블록 객체를 destroy() 메소드 호출을 통해 삭제하여 마무리한다.

앞으로 실행, 분해, 복제, 소멸에 대한 구현을 설명한다. 앞에서 봤듯이 이 네 가지 기능은 두 가지 방법에 대한 인터페이스를 제공한다. 따라서 이제 **실행, 분해, 복제, 소멸 기능에 대한 메소드**에 대해 설명하고 **이를 모두 설명한 후 이 메소드를 호출하게 되는 팝업 메뉴 방식과 직접 조작 방식(모서리에 각 영역 존재)에 대해 마지막에 설명**하겠다.

### 4) 실행

사용자가 팝업 메뉴의 실행 버튼을 클릭하거나 모서리의 실행 영역에 블록을 끌면 제일 먼저 **executeFunc()** 메소드가 실행된다. 이 메소드는 Math.js 를 사용하여 계산을 수행하고 **makeResultBlock()** 메소드를 호출하여 해당하는 블록들을 만든다. 마지막으로 **assembleResultBlock()** 메소드가 호출되어 생성된 블록들을 하나로 합친다. 이제 이 메소드들에 대해 하나씩 살펴보겠다.

```
933         // 계산에 관한 코드 (Math.js)
934         function executeFunc(){
935             let result = 0;
936 
937             try
938             {
939                 expression = MathApp.selected_block.name;
940                 result = parser.eval(expression).toString();
941                 var tokens = result.split(' ');
942                 if(tokens[0] == 'function')
943                 {
944                     result = 'function';
945                 }
946 
947                 // 결과를 하나하나 블록으로 쪼갬
948                 makeResultBlock(result);
949             }
950             catch (e)
951             {
952                 if(result != 'function')
953                 {
954                     alert(e);
```

제일 먼저 호출되는 **executeFunc() 메소드**이다. 939 행 ~ 945 행은 Math.js 를 이용하여 선택된 블록의 연산을 수행하고, 결과를 인자로 담아 makeResultBlock() 메소드를 호출한다. 950 행 ~ 954 행은 에러가 발생했을 때 alert() 형식으로 에러 메시지를 전달한다.

```
959     // 스트링을 하나하나 블록으로 쪼개어 전역 배열인 currentResultBlocks에 넣음
960     function makeResultBlock(result){
961         for(let i = 0; i < result.length; i++){
962             key = result[i];
963             if (key in MathApp.symbol_paths)
964             {
965                 let size = {
966                     width : SYMBOL_WIDTH,
967                     height : SYMBOL_HEIGHT
968                 };
969                 let position = {
970                     // 선택중인 블록 바로 밑에 생성되도록 함
971                     x : MathApp.selected_block.position.x - MathApp.selected_block.size.width/2 + size.width
972                     y : MathApp.selected_block.position.y + size.height/2 + size.height
973                 };
974 
975                 resultBlockCnt++;
976                 var resultSymbol = new MathApp.Symbol(position, size, key);
977                 currentResultBlocks.push(resultSymbol);
978             }
979         }
980     }
```

위에서 호출하는 **makeResultBlock() 메소드**이다. 이 메소드는 **인자로 들어오는 연산의 결과를 하나하나 블록으로 만들어 currentResultBlocks 전역 배열에 넣게 된다**. 961 행에서 결과의 길이만큼 반복문이 돌면서 블록이 생성된다. 이때, 블록의 위치는 현재 연산을 수행한 블록 바로 밑에 출력되도록 하였으며, 차례대로 생성되도록 위치를 설정하였다.(969 행 ~ 973 행) 설정한 위치, 크기, 이름을 가지고 Symbol 객체를 생성했으며, 생성된 객체를 currentResultBlocks 에 넣었다.

블록 객체를 생성하고 바로 블록을 조립하는 함수를 호출하면 되는데, 배열에 넣는 등 복잡하게 구현을 하였다. 왜냐하면 Symbol 생성자 내에서 **fabric.Image.fromURL 과 관련한 부분** 때문이다. 이 부분은 해당 이미지가 로드 될 때 실행되므로, 객체를 생성하고 바로 조립하는 함수를 호출하면 이미지를 제외한 부분들만 결합이 되게 된다. **따라서 생성된 블록들을 배열 안에 넣어두고 이미지가 모두 로드될 때 조립하는 함수를 호출하도록 구현하였다.**

```
821     if(resultBlockCnt != 0){
822         // 이미지 로드를 기다리고 있으면
823         imageCompCnt++;
824         if(imageCompCnt == resultBlockCnt){
825             // 모든 블록의 이미지가 로드되면
826             assembleResultBlock();
827         }
828     }
```

위의 코드는 Symbol 생성자에서 이미지를 로드하는 부분 마지막에 있는 코드이다. 블록을 currentResultBlocks 전역 배열에 넣을 때 카운트 변수인 resultBlockCnt 또한 증가하게 된다. 따라서 821 행 if 문이 의미하는 바는 이미지 로드를 기다리고

있으면 다음의 문장을 수행하라는 것이다. 그리고 각 블록당 하나의 이미지를 가지고 있으니 처리한 이미지 수와 전역 배열 요소의 개수가 같다는 것은 모든 블록의 이미지가 로드 되었다는 것이다. 이때 **assembleResultBlock() 메소드를 호출**한다.

```
982    // 전역 배열인 currentResultBlocks에 있는 블록들을 하나로 합침
983    function assembleResultBlock(){
984        for(let i = 1; i < currentResultBlocks.length; i++){
985            MathApp.assembleBlock(currentResultBlocks[0], currentResultBlocks[i]);
986        }
987
988        // 이미지 로드를 기다리는 블록이 없음을 표시
989        currentResultBlocks = [];
990        resultBlockCnt = 0;
991        imageCompCnt = 0;
992    }
```

위에서 호출하는 **assembleResultBlock() 메소드이다.** 이 메소드는 반복문을 돌면서 위의 블록 조립할 때 설명한 메소드인 **MathApp.assembleBlock()을 호출**하며 블록들을 하나로 결합한다. 이후 전역 배열과 카운트 변수 초기화를 진행하며 마무리한다.

### 4) 분해

사용자가 팝업 메뉴의 분해 버튼을 클릭하거나 모서리의 분해 영역에 블록을 끌면 **disassembleFunc()** 메소드가 실행된다. 다음은 이 메소드 부분이다.

```
994    // 선택한 것 가져와서 이름 저장 후 없애고, 새롭게 블럭들 생성
995    function disassembleFunc(){
996        let str = MathApp.selected_block.name;
997        let newX = MathApp.selected_block.position.x;
998        let newY = MathApp.selected_block.position.y;
999        let originWidth = MathApp.selected_block.size.width;
1000       MathApp.selected_block.destroy();
1001
1002       for(let i = 0; i < str.length; i++){
1003           key = str[i];
1004           if (key in MathApp.symbol_paths)
1005           {
1006               let size = {
1007                   width : SYMBOL_WIDTH,
1008                   height : SYMBOL_HEIGHT
1009               };
1010               let position = {
1011                   // 변수 i를 사용하여 원래 자리에 그대로 생성
1012                   x : newX - originWidth/2 + size.width/2 + i * size.width,
1013                   y : newY
1014               };
1015               var resultSymbol = new MathApp.Symbol(position, size, key);
1016           }
1017       }
1018   }
```

이 메소드는 먼저 선택한 블록의 위치, 크기, 이름(name)을 저장한 후 destroy() 메소드를 호출해 삭제한다. (996 행 ~ 1000 행) 이후 반복문을 돌면서 적절히 변수 i 값을 사용해 위치를 지정하며 원래 위치에 이름을 사용해 블록들을 차례대로 생성한다. (1002 행 ~ 1018 행)

### 5) 복제

사용자가 팝업 메뉴의 복제 버튼을 클릭하거나 모서리의 복제 영역에 블록을 끌면 **duplicateFunc()** 메소드가 실행된다. 다음은 이 메소드 부분이다.

```
1020  // 선택한 것 이름 저장 후 낱개의 블록 생성 후 합침
1021  function duplicateFunc(){
1022      let str = MathApp.selected_block.name;
1023
1024      for(let i = 0; i < str.length; i++){
1025          key = str[i];
1026          if (key in MathApp.symbol_paths)
1027          {
1028              let size = {
1029                  width : SYMBOL_WIDTH,
1030                  height : SYMBOL_HEIGHT
1031              };
1032              let position = {
1033                  // 선택중인 블록 바로 밑에 생성되도록 함
1034                  x : MathApp.selected_block.position.x - MathApp.selected_block.size.width/2
1035                  y : MathApp.selected_block.position.y + size.height/2 + size.height
1036              };
1037
1038              resultBlockCnt++;
1039              var resultSymbol = new MathApp.Symbol(position, size, key);
1040              currentResultBlocks.push(resultSymbol);
1041          }
1042      }
1043  }
```

위의 메소드는 먼저 선택한 블록의 이름을 저장하고, 1024 행 ~ 1042 행 반복문을 돌면서 위치를 적절히 설정해 이름으로 블록들을 생성한 후 1038 행 ~ 1040 행에서 블록들을 합치게 된다. (이 부분의 대한 설명은 '조립'에서 했으므로 생략한다.)

### 6) 소멸

사용자가 팝업 메뉴의 소멸 버튼을 클릭하거나 모서리의 소멸 영역에 블록을 끌면 **destroyFunc()** 메소드가 실행된다. 다음은 이 메소드 부분이다.

```
1045    function destroyFunc(){
1046        MathApp.selected_block.destroy();
1047    }
```

위 함수는 예제 코드로 구현되어 있는 destroy() 함수를 호출하여 선택한 블록을 제거한다.

### 7) 팝업 메뉴 구성

블록들을 클릭하면 팝업 메뉴가 활성화 되어, 사용자가 실행, 소멸, 복제, 분해를 할 수 있도록 하였다.

```
661        this.visual_items = []; // 그림 객체 저장을 위한 배열
662        this.popup_items = [];  // popup 객체 저장
```

먼저 예제 코드에 있던 블록 객체에 팝업 메뉴를 담는 배열을 다음과 같이 선언하였다.

```
788    // 팝업 메뉴 생성
789    makePopup(block, position, size)
```

이는 symbol 객체 생성자 중 일부이다. 다음과 같이 makePopup() 함수를 호출하도록 하여 팝업 메뉴를 생성하도록 하였다.

```
836    // 팝업을 생성
837    function makePopup(block, position, size){
838        // Make Boundary
839        let left = position.x - size.width / 2;
840        let top = position.y + size.height - size.height / 3 - 3;
841        for(let i = 0; i < 4; i++){
842            let boundary = makePopupBoundary(left, top);
843            block.popup_items.push(boundary);
844            left += (POPUP_WIDTH + 4);
845        }
```

위에서 호출하는 makePopup() 메소드이다. 먼저 위와 같이 for 반복문을 돌면서 makePopupBoundary() 메소드를 호출해 4 개의 경계선을 만들어낸다. 이후 화면에 출력하고, popup_items 배열에 넣는다.

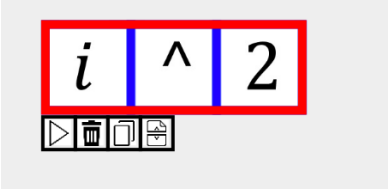
makePopupBoundary() 메소드는 **fabric.js 를 사용하여 구성**했으며, 이후 배경 또한 위와 비슷하게 만들어낸다. 이에 대한 설명들은 생략하겠다,

```
856     // Make Image
857     left = position.x - size.width / 2 + 3;
858     top = position.y + size.height - size.height / 3;
859     makePopupImage("./img/play.png", left, top, block);
860     left += (POPUP_WIDTH + 4);
861     makePopupImage("./img/trash.png", left, top, block);
862     left += (POPUP_WIDTH + 4);
863     makePopupImage("./img/copy.png", left, top, block);
864     left += (POPUP_WIDTH + 4);
865     makePopupImage("./img/split.png", left, top, block);
```

이미지를 생성하는 부분이다. makePopupImage() 메소드를 호출해 4 개의 이미지를 만들게 된다.

```
914     // Set Click Event Handler
915     if(path == "./img/play.png")
916         img.on('mousedown', function(e){
917             executeFunc();
918         });
919     else if(path == "./img/trash.png")
920         img.on('mousedown', function(e){
921             destroyFunc();
922         });
923     else if(path == "./img/copy.png")
924         img.on('mousedown', function(e){
925             duplicateFunc();
926         });
927     else if(path == "./img/split.png")
928         img.on('mousedown', function(e){
929             disassembleFunc();
930         });
```

makePopupImage() 메소드의 일부분이다. 앞 부분은 fabric.js 를 사용해 구성하였으며, 위의 코드 부분은 각 **이미지에 대해 클릭 이벤트를 설정**하는 부분이다.




최종적으로 구현한 팝업 메뉴이다. 각 이미지는 무료 이미지를 다운 받아 사용하였으며, 다운 받은 페이지에 명시된 대로 다음의 저작권 표시를 하겠다.

**실행 ICON**
Icon made by Smashicons(https://www.flaticon.com/authors/smashicons) from www.flaticon.com

**휴지통 ICON**
Icon made by Freepik(https://www.freepik.com/) from www.flaticon.com

**복사 ICON**
Icon made by Gregor Cresnar(https://www.flaticon.com/authors/gregor-cresnar)
from www.flaticon.com

**분할 ICON**
Icon made by Those Icons(https://www.flaticon.com/authors/those-icons) from www.flaticon.com

## 8) 직접 조작 방식 구현

왼쪽 위, 왼쪽 아래, 오른쪽 위, 오른쪽 아래 각 모서리에 실행, 분해, 복제, 삭제에 대한 영역이 있어 이곳에 **블록을 끌어 놓으면 해당 기능을 수행하도록 구현**하였다.

```
21    // 모서리 영역 너비와 높이
22    const ADDI_WIDTH = 160;
23    const ADDI_HEIGHT = 70;
```

제일 먼저 다음과 같이 영역의 너비와 높이를 상수로 지정하였다.

```
106    // x : 0 ~ ADDI_WIDTH
107    // y : 0 ~ ADDI_HEIGHT
108    let playText = new fabric.Text('Execute',{
109        left: 28,
110        top: 18,
111        fontSize: 30
112    });
113    let playRect = new fabric.Rect({
114        left: 0,
115        top: 0,
116        width: ADDI_WIDTH,
117        height: ADDI_HEIGHT,
118        fill: "rgba(0,0,0,0)",
119        stroke: "rgba(0,0,0,1)",
120        strokeWidth: 3,
121        selectable: false
122    });
123    MathApp.canvas.add(playText);
124    MathApp.canvas.add(playRect);
```

MathApp.Initialize() 메소드의 일부분이다. 여기서 **fabric.js 의 Text, Rect 를 사용해 각 모서리의 영역을 만들어낸다.** 위의 코드는 실행 영역을 만들어내는 코드로 실행 뿐만 아니라 다른 영역도 이와 비슷하게 만들어내었다.

```
270     // 각 모서리에 블록이 겹쳐지는지 확인
271     let num = MathApp.findOverlapAdditional();
272     // 해당하는 함수 호출
273     if(num == 1){
274         executeFunc();
275         MathApp.is_mouse_dragging = false;
276         MathApp.mouse_drag_prev = {x:0, y:0};
277         MathApp.selected_block.onDeselected();
278         return;
279     }
280     else if(num == 2){
```

이 부분은 마우스 클릭 버튼을 땠을 때 호출되는 함수인 MathApp.handleMouseUp() 메소드 중 일부이다. 271 행에서 호출하는 메소드는 선택한 블록이 어느 영역에 겹치게 되는지 반환하는 메소드이다. 1 을 반환하면 실행 영역에 블록이 겹친 것이므로 273 행 ~ 279 행은 이와 알맞게 처리를 수행한다.

```
440   MathApp.findOverlapAdditional = function(){
441       let x = MathApp.selected_block.position.x;
442       let y = MathApp.selected_block.position.y;
443       let width = MathApp.selected_block.size.width;
444       let height = MathApp.selected_block.size.height;
445
446       // Execute Boundary
447       // x : 0 ~ ADDI_WIDTH
448       // y : 0 ~ ADDI_HEIGHT
449       if( x - width/2 >= 0 &&
450           x - width/2 <= ADDI_WIDTH &&
451           y - height/2 >= 0 &&
452           y - height/2 <= ADDI_HEIGHT ){
453           // 왼쪽 위
454           return 1;
455       }
```

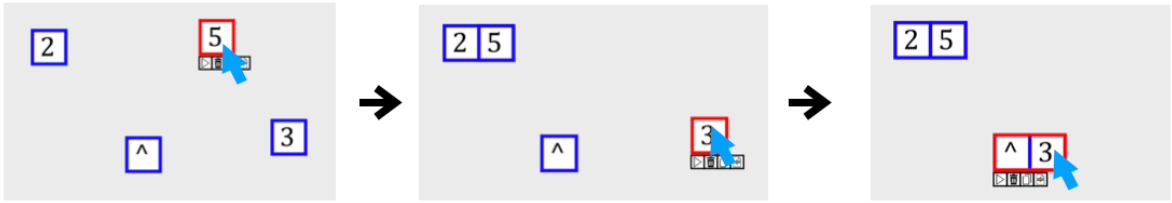
위에서 호출하는 findOverlapAdditional() 메소드 중 일부이다. **이 메소드는 선택한 블록이 어떤 영역과 중첩되는지 검사**한다. 449 행 ~ 455 행은 실행 영역과 선택한 블록 왼쪽 위 모서리와 비교하는 것으로 모든 모서리, 모든 영역에 대해 이를 수행한다. 만약 실행 영역과 겹친다면 1, 복제 영역과 겹친다면 2, 분해 영역과 겹친다면 3, 소멸 영역과 겹친다면 4 를 반환한다.


위의 사진은 구현한 직접 조작 계산기의 초기 화면이다.

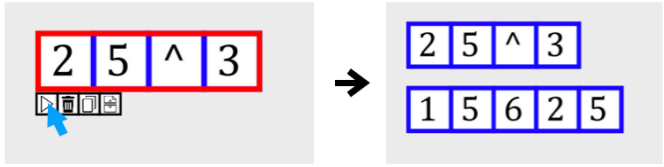
## 4. 실제 문제에 대한 사용 예시

(YouTube 링크 : https://youtu.be/nYcy6bmGSo8)

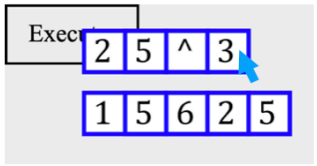
1. 25 의 세제곱을 구하시오.



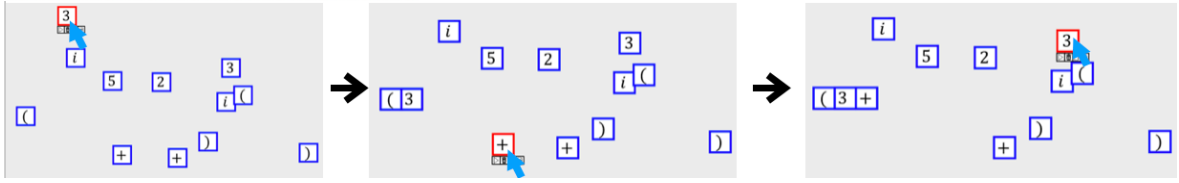

첫 번째 방법

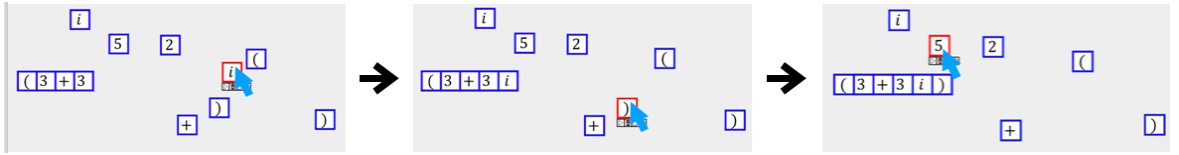


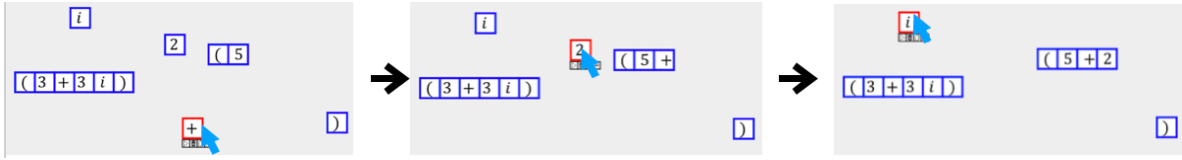

두 번째 방법

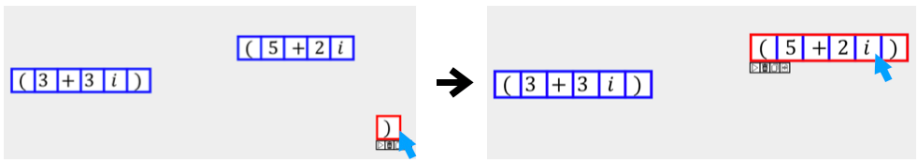



2. $(3+3i)(5+2i) = ?$

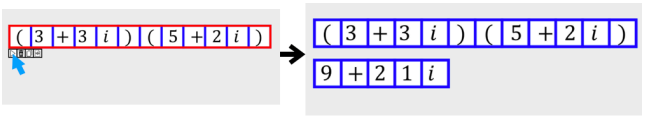




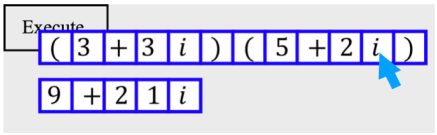




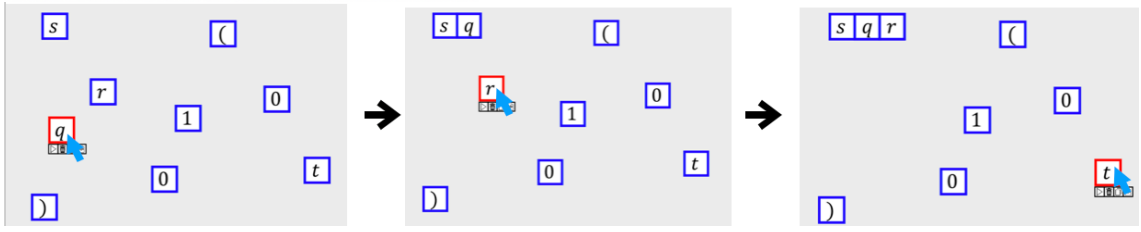
첫 번째 방법



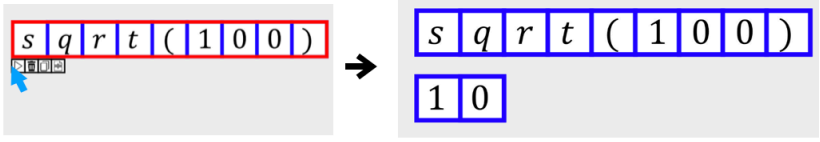

두 번째 방법



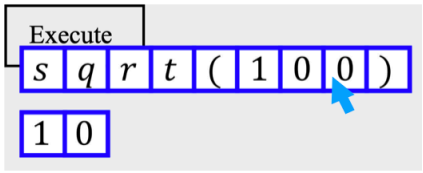

## 3. 100 의 제곱근을 구하시오.




첫 번째 방법




두 번째 방법



## 5. 구현 측면에서 성공적인 부분과 실패한 부분

가장 성공적인 부분은 **블록이 중첩되었을 때, 조립하는 부분**이라고 생각한다. 이 부분을 구현하는데 있어 많은 어려움이 있었기 때문이다. 위에서 구현 파트에서 언급했듯이 블록을 조립할 때, 복사할 블록 객체의 시각적 요소들을 모두 새로 생성하고 이를 합친 후 원본을 삭제하는 방식으로 구현하였는데, 블록 생성자 중 Image 를 로드한 후 실행되는 코드 부분이 있어 새로 생성하고 나서 바로 조립을 해버리면 Image 를 제외한 부분들만 조립이 되어, 화면에 제대로 표시되지 않는 문제가 있었다. 따라서 생성자 중 Image 가 로드된 후 실행되는 코드를 모두 수행하고 나서 조립하는 메소드를 호출하도록 **직접 콜백을 구현하여 해결**하였다.

이제 실패한 부분에 대해 언급하겠다. 제일 먼저 팝업 메뉴를 통한 실행 연산을 항상 제공하는 것이 아니라, **연산이 가능할 때만 표시되도록 구현할 계획**이었다. 따라서 사용자가 현재 조립한 블록들이 유효한 값인지 쉽게 확인을 하도록 하려고 했으나, 구현 상의 어려움이 있어 이 부분은 구현하지 못한 채 완성하였다.

두 번째로, 1 차 과제처럼 **도움말 기능**을 추가적으로 구현할 계획에 있었다. 도움말 기능을 제공하면, 사용자가 이러한 직접 조작 계산기를 처음 접했을 때 더 쉽게 사용할 수 있을 거라 생각했기 때문이다. 그러나 과제의 기본 요구조건을 수행하는데 있어 많은 시간을 투자해 결국 시간이 모자라 구현을 하지 못한 채 과제를 제출하게 되었다. 이 부분을 보완한 점에 대해 **사용성의 긍정적인 측면에서 다시 언급하겠다**.

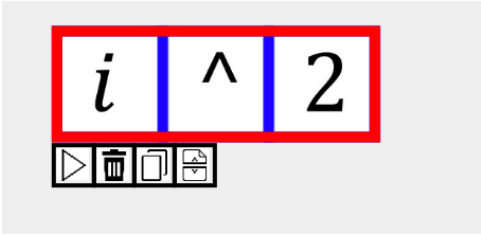
## 6. 사용성 측면에서 긍정적인 측면과 부정적인 측면

사용성 측면에서 가장 긍정적인 측면은 **사용자에게 실행, 복제, 분해, 소멸 기능을 두 가지 방법으로 제공한 것이라 생각**한다. 기본 요구조건에 있었던 팝업 메뉴로 제공하는 것 뿐만 아니라, 사용자는 **직접 조작 방식으로 블록을 끌어 해당 영역에 놓으면 해당하는 기능이 수행**된다. 이는 사용자에게 여러 가지 선택지를 줌으로서 사용자가 더 편리한 방법대로 사용하도록 하였다.

위에서 언급한 블록을 끌어 해당 영역에 놓아 해당하는 기능이 수행되도록 하는 **직접 조작 방식**에 대해 더 언급하겠다. 이 기능을 추가하는데 있어 어떻게 하면 사용자가 더 편리하게 실행, 복제, 분해, 소멸 기능들을 수행할 수 있을까 생각하게 되었다. 먼저, 블록들을 끌어 조립하고 이동하는 등의 행위에서 착안하여, 다양한 기능들 또한 이러한 직접 조작 방식으로 구현할 수 있을까? 하고 다시 생각해 보았

다. 이렇게 모든 행위를 직접 조작으로 통일함으로서 사용자는 더 편리하게 사용할 수 있을 것이며, 직접 사용해 본 결과 기능을 수행하는데 있어 더 빠르게 작업할 수 있다는 점에서 상당히 긍정적이다.

세 번째로, **사용자가 도움말 없이 직관적으로 계산기의 기능을 이용할 수 있도록 팝업 메뉴 이미지를 사용자가 한 눈에 알아볼 수 있도록 직관적으로 선택하였다.** 아래 사진은 블록 클릭 시 나타나는 팝업 메뉴이다.




위의 이미지들을 보면 각 이미지들이 기능에 맞게 직관적으로 표시되고 있음을 볼 수 있다. 실행 기능은 플레이 버튼으로, 소멸 기능은 휴지통 버튼으로, 복제 기능은 복사 버튼으로, 분해 버튼은 분할 된 버튼으로 구성하였다.

사용성의 부정적인 측면이 없도록 고려하여 구현하였기 때문에 부정적인 측면은 없다고 생각한다. 물론 개인적인 의견이므로 추후, 사용자의 사용 패턴을 파악하여 개선해 나가도록 하겠다.

## 7. 과제 결과에 대한 전반적인 자체 평가 및 향후 개선 계획

이번 2 차 과제도 1 차 과제와 마찬가지로 계산기를 구현할 때, 어떻게 하면 사용자가 쉽고 편리하게 기능들을 이용할 수 있을까를 중점으로 효율적인 인터페이스들을 설계하려 노력하였다. 따라서 위의 긍정적인 측면들을 보듯이 사용자에게 편리한 인터페이스를 추가적으로 구현하는 등 과제를 매우 성공적으로 수행했다고 생각한다. 다만, 사용자의 경험을 위한 조건으로 유용성, 사용성, 감성이 있다면, 유용성과 사용성은 위의 긍정적인 측면들로 인해 충족이 되었지만, 여전히 사용자에게 즐거움을 주는 감성적인 부분은 부족하다고 생각한다. 이 감성적인 부분과 위에서 언급한 사용성의 부정적인 측면이 발견된다면, 추후에 이 부분을 해결할 계획이다.

**YouTube 링크 : https://youtu.be/nYcy6bmGSo8**