Compiler Construction Project IV

(Code Generator)

제출자 : 26조

20120121 이희락

제출일 : 2019년 6월 25일

**프로젝트 제목 : Design and Development of Compiler for C- Language**

**Phase 4 : A code Generator**

**제출일 : 2019년 6월 25일**

**참여조원: 26조, 20120121 이희락**

**1. 개발목표**

|  |
| --- |
| Project2에서 구현한 AST와 Project 3에서 추가한 각 symbol들의 정보를 이용하여 SPIM machine에서 수행될 수 있는 Code를 생성하는 프로그램을 만든다. |

**2. 개발 범위 및 내용**

**(1) cgen.h / cgen.c**

|  |
| --- |
| project4에서 새롭게 추가된 파일. Semantic Error Check가 끝난 후 Code생성을 위해 호출될 함수를 선언/정의 하고 있다. codeGen()함수는 main에서 호출되며, SPIM Instruction code를 생성하여 \*.tm파일에 저장한다 |

**(2) main.c**

|  |
| --- |
| 생성된 code를 저장할 .tm파일을 만들고 codeGen()함수를 호출하는 코드를 덧붙인다. addIO()함수를 통해 표준입출력에 대한 함수 “input”과 “output”를 추가한다. |

**3. 추진 일정 및 개발 방법**

**가. 추진일정**

6월 6일 ~ 6월 8일 : code.h, code.c 파일 완성

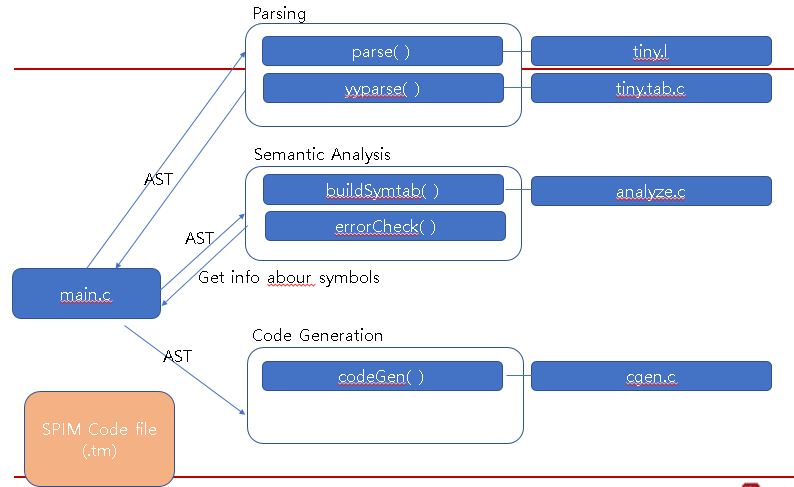
6월 24일 ~6월 25일: 최종 디버깅 및 test case 수행

**나. 개발 방법**

cspro9 서버에서 vi editor를 이용하여 프로그램을 작성했다. 수행 여부는 미리 설치되어있는 SPIM simulator를 사용하여 확인했다.

**4. 연구 결과**

**I) 합성 내용**



**<그림 1. 프로그램의 개략적인 flow>**

Project4에서 추가한 내용은 Code Generation 부분에 해당한다. <그림1>에서 확인할 수 있듯이 main함수는 Semantic Error check가 끝난 후 AST와 더불어 C- program에서 사용된 각 symbol에 대한 추가적인 정보를 얻는다. 이에 해당하는 정보들은 다음과 같다.

1) 각 프로시져의 stack에 대한 정보를 담고 있는 stack\_instance

2) 각 symbol의 memory location($fp로부터의 offset)

3) 각 symbol의 전역, 지역 변수 여부, array 변수 여부

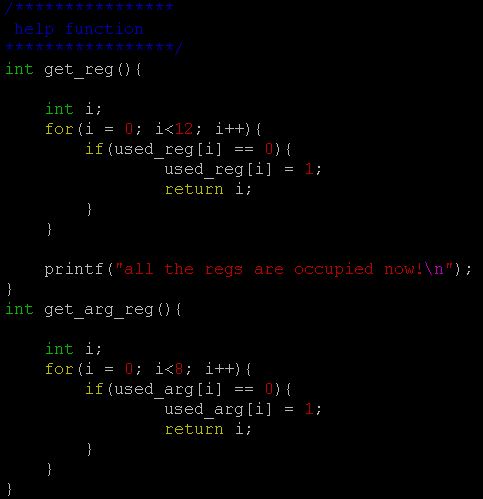
codeGen()함수에 1,2,3의 정보를 담고 있는 AST와 stack\_instance 및 결과인 code를 담을file의 descriptor를 넘겨 주어, 최종적으로 .tm파일을 생성하게 된다.

**II) 분석 내용**

가장 중요하게 사용되는 자료구조는 AST이며, 미리 정해놓은 AST의 각 NODE의 특징(이를 테면 child의 개수 및 NODE의 종류)에 맞추어 AST 순회 도중 이에 각 NODE가 나타내는 C- code statement에 적합한 SPIM code를 생성하는 것이 관건이 되었다. 또한 각 procedure마다 할당될 STACK의 구조를 미리 알고 이에 맞게 $sp, $fp의 위치를 정해주어야 했기에 각 프로시져의 메모리 정보를 담고 있는 stack\_instance의 활용 또한 매우 중요했다. 그렇기에 stack\_instance에 대한 pointer 역시 AST 순회 과정에서 계속해서 parameter로 넘겨주어 callK node를 처리하는 코드에서 사용했다. 또한 Code Generation을 위한 AST 순회과정 중에서는 “ASSIGN”을 나타내는 NODE를 만날 시에 L-value와 R-value 처리를 위해 기존의 pre-order순회 방식을 멈추고, 이를 따로 처리해준 뒤 바로 sibling node로 넘어갈 필요가 있었으므로 기존 AST 순회 알고리즘을 수정할 필요가 있었다. 이에 대한 설명은 제작의 codeGen() 함수에서 더 구체적으로 서술한다.

**III)제작 내용**

**register 관리**



|  |
| --- |
| Three-address code 방식을 사용하는 SPIM에서는 사칙/논리 연산, argument passing, procedure call, memory access 등 모든 부분에서 register를 사용하므로 이들에 대한 관리가 매우 중요하다. 따라서 이들을 관리할 수 있는 자료구조와 함수를 사용했다.  그림에서 보이는 used\_reg 및 used\_arg는 reg\_name 및 arg\_name 배열과 연관되어 사용된다. used\_reg 및 used\_arg의 값이 0인 register는 사용 가능함을 나타낸다. get\_reg() 및 get\_arg()함수는 이 배열을 linear하게 검색하여 처음으로 0인 regster의 index를 return하며 해당 element를 1로 set한다. 코드 생성시 넘겨받은 index를 이용하여 reg\_name과 arg\_name에 hard-coding된 각 register를 이용할 수 있게 만들었다. |

**codeGen()**

|  |
| --- |
| **return** : void  **parameter** : TreeNode \* AST, FILE \* codefile, runtime\_stack \* stack\_instance  **functionality** : AST와 code를 담을 file descriptor 및 각 procedure의 memory정보를 담고있는 stack\_instance를 인자로 받는다. AST를 pre-order로 순회하며 **switch-case**문을 통해 Statement구문과, ASSIGN, Call, DeclarK 구문을 처리하며 나머지 Expression 구문은 직접 처리하지 않고 ASSIGN 구문의 처리과정에서 genExp()함수를 호출하여 처리한다. expression을 별도로 처리한 이유는 “ASSIGN” 연산에서 l-value와 r-value를 처리하기 위한 memory 접근 방식이 다르기 때문이다. l-value는 연산 결과를 “sw”를 통해 해당 메모리 영역에 저장해야 하며 r-value는 메모리 영역에 저장된 값을 register로 가져오는 특수성을 고려해야했다. 또한 Expression에는 논리연산도 포함되어있는데, 이들의 경우 IF,WHILE에서 사용되므로 statement 처리과정에서 일반화하여 이들을 모두 포괄하는 code를 생성할 수 없다는 점에서 이러한 방식을 채택했다. 논리연산의 경우는 genStmt()함수에서 호출되는 genExpK()함수에서 처리되어 register에 논리연산의 결과를 넘겨준다. |

**genStmt()**

|  |
| --- |
| **return :** void  **parameter :** TreeNode \* AST, FILE \* codefile, runtime\_stack \* stack\_instance  **functionality :** codeGen에서 호출되어 Select, Iteration 구문에 대한 code를 생성하기위해사용되는 help function이다. 주요하게 사용되는 함수로는 **gen\_label( )** 함수가 있다. selection의 경우 if문과 if-else 구문에 대한 처리를 따로 해주었다.  **if statement 처리 과정**  1) gen\_label로 label 번호 생성.  2) genExpK()를 논리 연산의 결과를 저장하고 있는 register번호 획득  3) “beqz” 코드를 통해 2)에서 얻은 register의 결과값 확인하여 branch하는 code생성  4) True일시 진행되는 code 생성(여기서 다시 codeGen()을 호출한다.)  5) 3)의 결과가 False일 시 branch되는 label code 생성.  **if-else statement 처리 과정**  1) genExpK()호출을 통해 조건문의 논리 연산의 결과를 저장하고 있는 register 번호 획득  2) “beqz” 코드를 통해 1)에서 얻은 register의 결과값을 확인하여 branch하는 code생성  3) 조건이 true인 경우 수행되는 코드를 codeGen() 호출을 통해 생성  4) 3)에서 수행된 코드 진행 이후 else 구문을 수행하지 않도록 branch하는 코드생성  5) 2)결과가 False일시 else 구문 코드 생성을 위해 branch할 label 코드 생성  6) else 구문에 대한 코드 생성, 이때도 역시 codeGen()을 호출한다.  7) if문 수행 이후 branch할 label코드 생성  8) 사용한 register들 회수 (used\_reg의 값을 0으로 다시 만들어준다.)  **while statement 처리 과정**  1)조건 check를 하는 구문으로 jump하는 branch코드 생성  2)while문 내부의 statement를 위한 code branch label 생성  3)while문 내부의 statement에 대한 실제 code 생성(codeGen()함수 호출)  4)1)에서 조건 check를위해 branch하는 label 생성  5)조건에 대한 code 생성, genExpK()함수를 이용한다.  6)4)에서의 결과값을 저장하고 있는 register의 값을 확인하여 2)에서 만든 label로 branch할지 결정하는 코드 생성(“bnez 사용”) |

**genExpK()**

|  |
| --- |
| **return : int (연산 혹은 메모리 접근 결과를 저장하고 있는 register의 index값)**  **parameter :** TreeNode \* AST, FILE \* codefile, runtime\_stack \* stack\_instance, int lr\_value  **functionality :** genCode()에서 ‘ASSIGN’ node를 만났을 때 혹은 genStmt()에서 논리연산의 결과를 저장할 register를 반환한다. 논리/사칙 연산의 경우 genExpK()를 recursive하게 두 번 호출하여 연산의 좌측 식과 우측 식의 값을 저장한 register를 받은 뒤 이를 적절한 SPIM instruction을 선택하여 다시금 그 둘의 연산 결과를 다른 register에 저장한다. 이후 각 식의 값을 저장하고 있던 register는 0의 값을 주어 다시 회수하며 연산의 결과를 저장하고 있는 register의 번호는 회수하지 않고 반환 값으로 caller에서 넘겨준다. genExpK()에서 주요하게 고려해야 할 점은 바로 ‘Id’ Node에 대한 처리이다. Id의 경우는 memory access를 통해 해당 memory의 값을 register에 저장시켜야 하기 때문에 코드 생성에 있어서 레지스터 관리가 특히나 더 엄격해야 했다. 특히 array 변수에 대한 접근을 위해서는 해당 array의 base register를 먼저 참조해야 했는데, 이는 global, local을 나누어 고려해야 했으며 특히 local일 경우 parameter로 받은 array이인지 혹은 자신의 procedure에서 선언한 array인지에 따라 base address를 접근하는 방식에 차이가 있으므로 모든 경우를 고려하여 code를 생성해줄 필요가 있었다.  **1) global\_array에 대한 접근**  - get\_reg()함수를 통해 base\_address를 point할 임의의 register를 선택한다.  - AST node에 저장된 memloc이 base를 가리키고 있으므로 $gp함수에서 해당 memory 크기만큼 ‘위로’ 떨어져 있을 memory 위치를 앞서 구한 임의의 register가 가리키게한다. 이는 $gp의 내용에서 memory 위치만큼을 더한 결과값을 해당 register에 넣어줌으로써 가능하다.  **2) local이면서 현재 procedure에서 선언된 array에 대한 접근**  - get\_reg()함수를 통해 base\_address를 point할 임의의 register를 선택한다.  - memory 위치가 음수인 경우에 해당한다. local의 경우 $fp 아래에 위치하도록 설정했기 때문에 memory 위치를 나타내는 값의 부호를 통해 구분 가능하다. fp로부터 해당 값의 절대값 만큼 subu해주어 base에 대한 접근이 가능하다.  **3) 다른 procedure에서 선언된 array변수에 대한 접근**  - get\_reg()함수를 통해 base\_address를 point할 register를 선택한다.  - 같은 방법으로 caller의 fp에 접근할 임의의 register를 선택한다.  - parameter로 받은 array의 경우 AST에 array에 대한 flag가 set되어있으며 동시에 memory 위치를 나타내는 변수가 ‘양수’가 되도록 설정했기 때문에 이를 파악할 수 있다.  - caller로 이동하여 기존 parameter의 위치에 저장된 값만큼 offset을 이동하여 해당 메모리의 값을 읽는다. 이것이 음수라면 해당 procedure에서 선언된 array이고, 다시 양수라면 이 역시 다른 procedure에서 선언되고 paramter로 넘겨진 변수라는 뜻으로, 다시금 caller의 caller의 fp로 이동한다.  - 앞선 step을 반복하다가 offset을 나타내는 메모리 공간의 값이 음수인 경우에 해당 fp의 값에서 offset만큼을 아래로 이동하여 base address에 접근할 수 있다.  🡺1,2,3의 경우 index만큼의 element에 대한 접근 방식은 동일하다. array 변수를 나타내는 node의 경우 첫번째 자식 노드에 해당 subscript에 대한 expression이 저장되어있으므로 해당 값을 얻도록 genExpK()호출을 통해 특정 레지스터에 해당 값을 저장하고 이에 4만큼을 곱하여 base로부터 원하는 index의 element가 저장된 memory에 접근할 수 있다.  다만 주의해야했던 점은 global의 경우 base가 맨위에 있고 local의 경우 base가 맨 아래에 위치했기에 subu를 해야할지 addu를 해야할지 잘 구분해야했다. |

**genDeclar()함수와 procedure stack 관리**

|  |
| --- |
| **return :** void  **parameter :** TreeNode \* AST, FILE \* codefile, runtime\_stack \* stack\_instance  **functionality :** 함수 선언 Node를 만났을 경우 codeGen()에서 호출되는 함수이다.  AST 순회 과정에서 계속해서 넘겨주었던 stack\_instacne 자료구조가 본격적으로 이용되는 부문이며, 각 프로시져의 stack구조를 결정하여 할당하는데 사용된다.  stack\_instance는 stack의 크기, parameter를 위한 메모리 공간의 크기, local variable을 위한 메모리 크기 정보가 들어있으므로 sp를 procedure stack의 크기만큼 아래로 내리고, fp를 sp 에서 (local읠 위한 크기 + 4)만큼의 위치에 고정시킨다. 4는 return address를 위한 공간 때문에 생긴 값이다. 또한 $a0~ $a4등에 이르는 argument 값이 자신의 parameter영역에 저장되어 있음이 항상 보장 되어야 하므로, 각각의 register를 자신의 parameter 영역에 “sw”하는 코드를 마지막에 반드시 추가한다. 이어지는 코드에서는 자신의 stack영역을 회수하는 **DeallocStack()**을 호출한다**.** 해당 함수는 jr $ra를 통해 caller로 jump하는 코드와, sp를 stack 크기만큼 다시 줄이는 코드를 생성하여 메모리 영역이 procedure수행과 동시에 회수될 수 있도록 한다. |

**Return value 회수**

return의 경우 항상 $V0 레지스터를 이용하여 caller로 넘어가므로 return statement의 값을 특정 register에 저장한뒤 이를 다시 v0으로 move하는 코드를 생성한다.

**addIO**

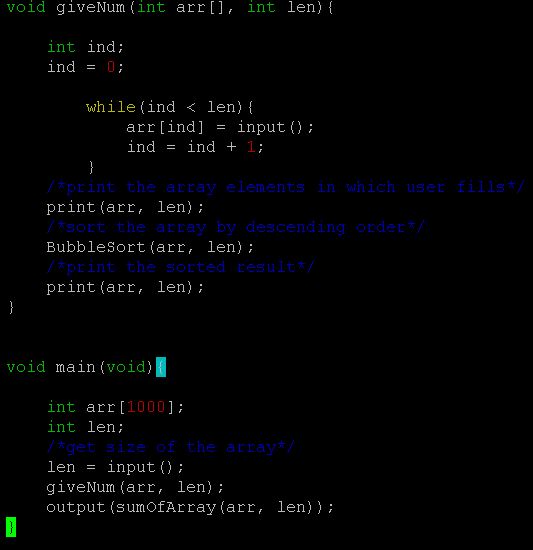
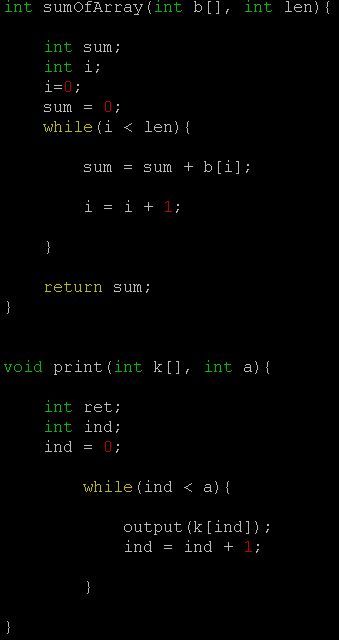
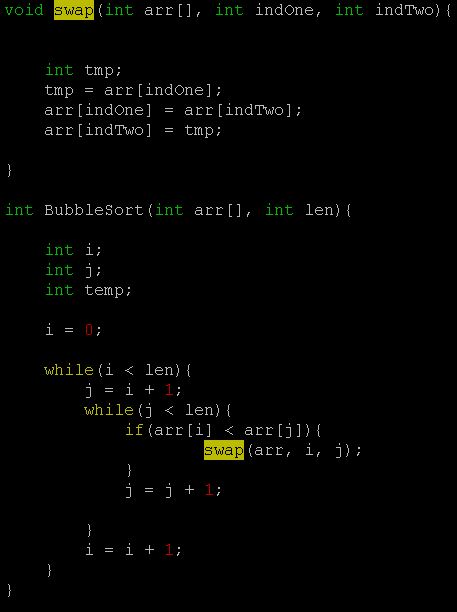
output : intput: 의 prompt를 띄어줄 수 있도록 file의 .data 영역에 해당 string을 저장해주는 함수. input과 output은 semantic error check과정에서 선언되어있지 않아도 넘어가도록 설정해주어 사용할 수 있도록 했다.

**IV)시험 내용**

결과 확인을 위해 두 가지 test case를 이용했으며 프로젝트 파일에 각각 test1.c test2.c로 첨부되어있다.

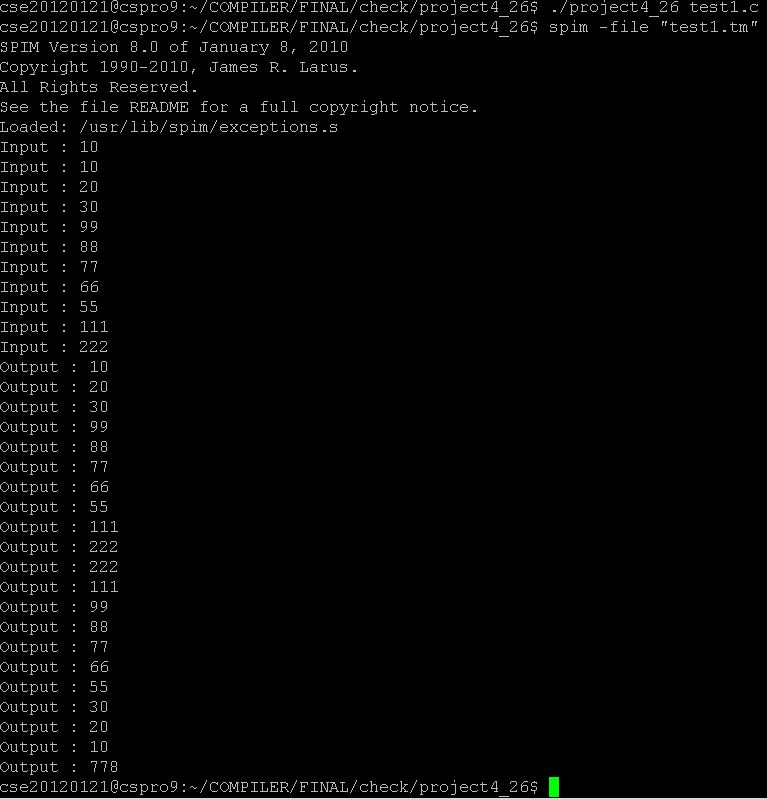
**test1.c**

|  |
| --- |
| **test case to check:**  (1) Accessing Local array which is declared in other procedure and passed as argument  (2) User input and Output  **Flow**  (1) main() : get Array length from user(Cannot exceed 1000)  [and call giveNUM() with passing array and length as argument]  (2) giveNum(int arr[], int len) : get User inputs by "length" times, and each value is stored in th e array. [and call print]  (3) print(int arr[], len) : print the array element  (4) BubbleSort(int arr[], len) : Sort the array by descending order  (5) sumOfArray(int arr[], len) : get total sum of the array element  (6) each step is processed different procedure |



<test1.c의 코드>

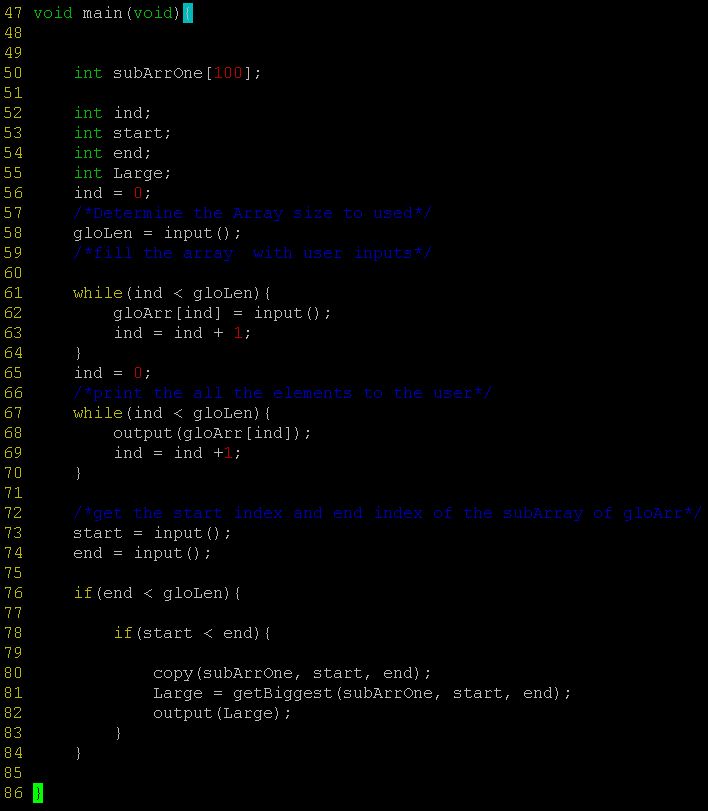
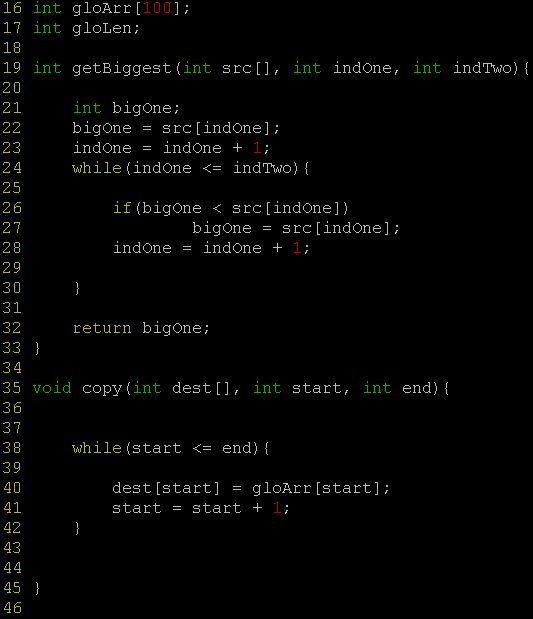
🡺parameter로 받은 array에접근하여 input()함수를 통해 값을 정하고 이를 다시 출력,다시금 해당 array를 parameter로 넘겨 sort한 후 모든 element의 합을 출력해주는 C-코드



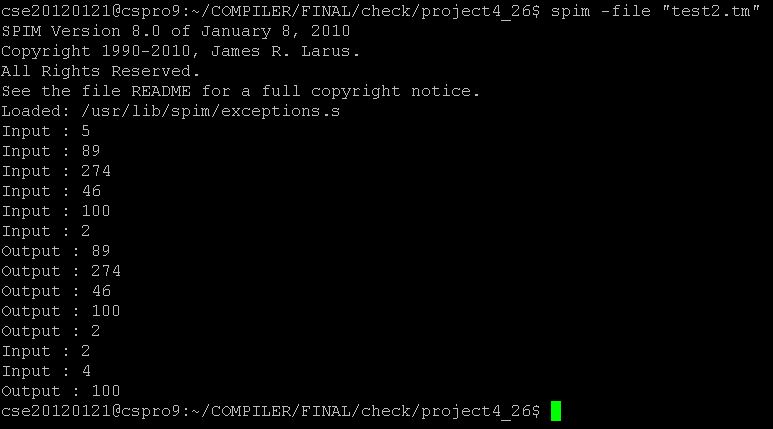
🡺10의 입력을 받은 후, 10개의 사용자 입력을 저장하고, 저장된 결과를 보여준다. 이후 sorting된 결과를 차례대로 보여주며 마지막으로는 모든 element의 합을 출력한다.

**test2.c**

|  |
| --- |
| **Test C- program to check**  (1) Accessing global Array  (2) Accessing local Array which is declared in caller proceduere and passed as argument  (3) Check while statement and if statment  (4) Evaluate the Relative Operation    **FLOW**  get Length of the global array  -> fill the array with user input  -> get the start index and end index of the subArray of the globalArray  -> copy  -> get largest # of the subArray |



* main 함수에서 사용자로부터 global array의 사용할만큼의 size를 input으로 입력받아 gloLEN에 저장. 다시금 subarray의 시작, 끝 index를 입력 받아, 해당 부분을 local로 copy한 후, 그 중 가장 큰 element를 출력하는 C- 코드



* 사용자로부터 global varable의 앞선 5개의 칸만을 이용함을 입력받고, 이후 5번의 추가 입력을 받아 global variable의 각 위치에 저장. 이후 저장된 결과를 차례대로 보여준다.
* sub로 자신의 local로 복사할 index 2,4를 입력 받아 local array에 옮긴 뒤, 그 안에서 가장 큰 element인 100을 출력하는 모습