Basic Interpreter 분석 보고서

성결대학교 컴퓨터공학과 20220854 윤소영

목차

---

1. 자료구조

1.1 Node 구조체

```
struct node {
    int type;         // 1:변수, 2:함수, 3:함수호출, 4:begin, 5:end
    char exp_data;    // 심볼 이름 (1글자)
    int val;          // 변수 값
    int line;         // 함수 정의/호출 라인
    struct node* next;
};
```

역할: 스택의 기본 단위. 변수, 함수, 제어 흐름 정보 저장

시각화:

```
┌──────────────────┐
│ type = 1         │
│ exp_data = 'a'   │  ← 변수 a
│ val = 5          │
│ line = 0         │
│ next ────────────┼──→ [다음 Node]
└──────────────────┘
```

1.2 Stack 구조체 (심볼 테이블)

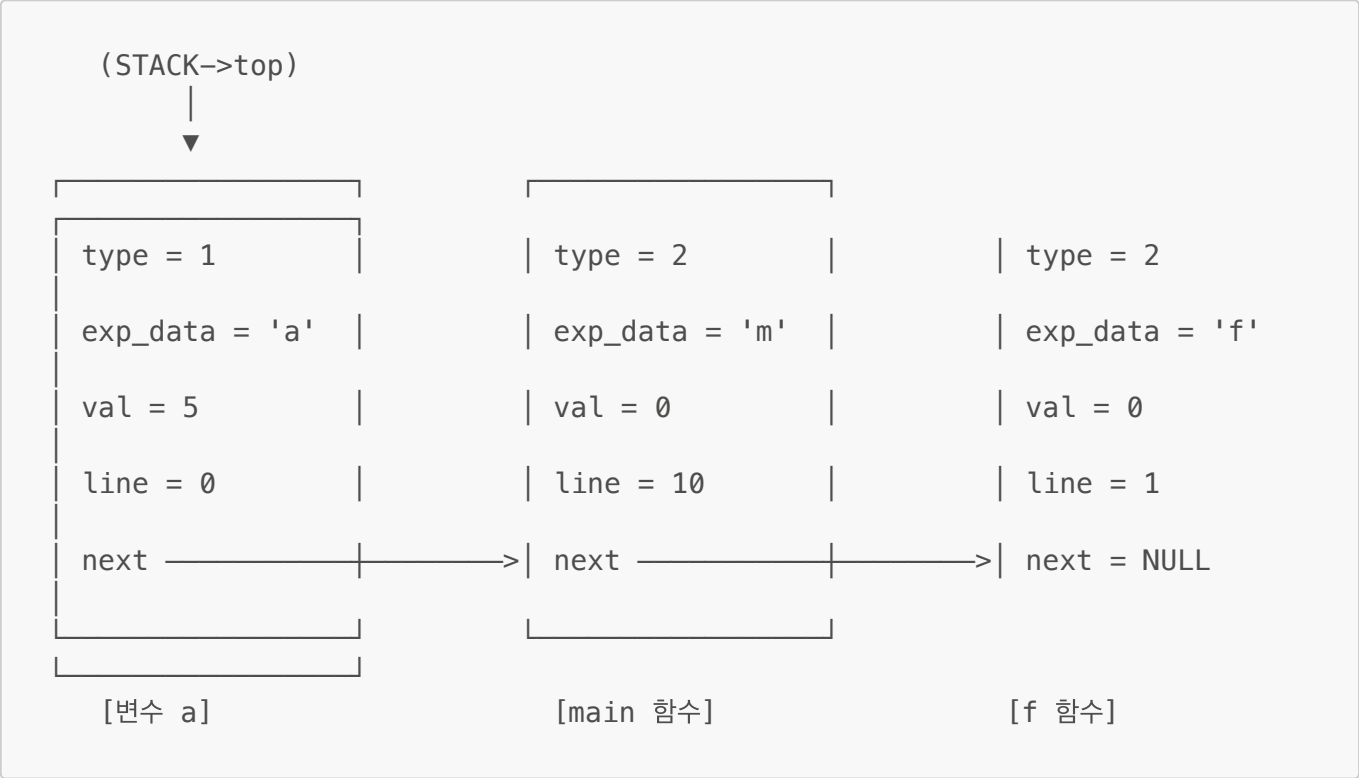
```
struct stack { Node* top; };
```

역할: 변수/함수 관리, LIFO 방식

주요 연산:

- `Push()`: 노드 추가 (O(1))
- `Pop()`: 노드 제거 (O(1))
- `GetVal()`: 심볼 검색 (O(n))

시각화:



1.3 OpStack 구조체 (중위→후위 변환)

```
struct opstack { opNode* top; };
struct opnode { char op; struct opnode* next; };
```

역할: 수식 변환 시 연산자 임시 저장

예시: `(a + b * c)` 파싱

```
단계 1: '+' push
[+] → NULL

단계 2: '*' push (우선순위 높음)
```

```
[*] → [+] → NULL

단계 3: ')' 만남
PopOp() → '*', '+' 순서로 후위 표기식에 추가
```

1.4 PostfixStack 구조체 (후위 표기식 계산)

```
struct postfixstack { Postfixnode* top; };
struct postfixnode { int val; struct postfixnode* next; };
```

역할: 후위 표기식 계산 시 피연산자 저장

예시: `5 3 +` 계산

```
1. 5 push: [5]
2. 3 push: [3]→[5]
3. '+' 만남:
   - pop: 3, 5
   - 계산: 5 + 3 = 8
   - push: [8]
```

---

2. 실행 흐름

입력 파일 `input1.spl`

```
function f(int a)
begin
   int b = 6;
   int c = 2;
   ((b+c)/a);
end

function main()
begin
   int a = 1;
   int b = 2;
   int c = 4;
   ((6 + f(c) ) / b);
end
```

Phase 1: 초기화 및 함수 등록

목표: main을 찾을 때까지 함수 정의만 스택에 등록

라인 1: `function f(int a)` 처리

```
// basic_interpreter.c 276-304 중 일부
if (my_stricmp("function", firstword) == 0) {
    firstword = strtok(NULL, " ");      // "f(int"
    tempNode.type = 2;                   // 함수 타입
    tempNode.exp_data = firstword[0];    // 'f'
    tempNode.line = curLine;             // 1
    STACK = Push(tempNode, STACK);
}
```

`f 함수`를 발견하여 스택에 Push.

스택 변화:

```
STACK
[top] → [type=2, exp_data='f', line=1]  ← f 함수
      → NULL
```

라인 2-6: 함수 본문 무시

```
// basic_interpreter.c L204-211
if (my_stricmp("begin", line) == 0) {
    if (foundMain) {  // foundMain = 0 → 무시
        // ...
    }
}
```

`spl` 파일의 2번째 줄부터 6번째 줄은 함수 f의 내부이다. main 함수를 찾지 못한 `foundMain = 0`상태이기 때문에 무시한다.

따라서 스택 상태는 여전히 f함수만 등록되어 있다.

```
STACK
┌──────────────┐
│ top ─────────┼──→ [type=2, exp_data='f', line=1] → NULL
└──────────────┘
```

라인 8: `function main()` 처리

```
// basic_interpreter.c L281-290
tempNode.type = 2;
tempNode.exp_data = 'm';
```

```
tempNode.line = 8;
STACK = Push(tempNode, STACK);

// main 감지
if (firstword[0] == 'm' && firstword[1] == 'a' &&
    firstword[2] == 'i' && firstword[3] == 'n') {
    foundMain = 1;  // <- 이제 코드 실행 시작
}
```

최종 스택:

```
STACK
[top] → [type=2, exp_data='m', line=8] ← main 함수
      → [type=2, exp_data='f', line=1] ← f 함수
      → NULL
```

Phase 2: main 함수 실행

라인 9: `begin` 마커 추가

```
// basic_interpreter.c L204-211
if (my_stricmp("begin", line) == 0)
        {
            if (foundMain)
            {
                tempNode.type = 4; // begin 마커
                STACK = Push(tempNode, STACK);
            }
        }
```

스택:

```
STACK
[top] → [type=4]                          ← begin 마커
      → [type=2, exp_data='m', line=8]    ← main 함수
      → [type=2, exp_data='f', line=1]    ← f 함수
      → NULL
```

라인 10: `int a = 1;` 처리

```
// basic_interpreter.c L253-275
if (my_stricmp("int", firstword) == 0)
{
```

```
    if (foundMain)
    {
        tempNode.type = 1;                  // 변수
        firstword = strtok(NULL, " ");      // "a"
        if (!firstword) continue;
        tempNode.exp_data = firstword[0];   // 'a'

        firstword = strtok(NULL, " ");      // "="
        if (!firstword) continue;

        if (my_stricmp("=", firstword) == 0)
        {
            firstword = strtok(NULL, " ");  // "1;"
            if (!firstword) continue;
        }

        tempNode.val = atoi(firstword);     // 1
        tempNode.line = 0;
        STACK = Push(tempNode, STACK);
    }
}
```

스택:

```
STACK
[top] → [type=1, exp_data='a', val=1]   ← a 변수
      → [type=4]                        ← begin
      → [type=2, exp_data='m', line=8]  ← main 함수
      → [type=2, exp_data='f', line=1]  ← f 함수
      → NULL
```

라인 11-12: `int b = 2;`, `int c = 4;` 처리

같은 방식으로 변수 b, c를 스택에 추가한다.

스택:

```
STACK
[top] → [type=1, exp_data='c', val=4]   ← c 변수
      → [type=1, exp_data='b', val=2]   ← b 변수
      → [type=1, exp_data='a', val=1]   ← a 변수
      → [type=4]                        ← begin
      → [type=2, exp_data='m', line=8]  ← main 함수
      → [type=2, exp_data='f', line=1]  ← f 함수
      → NULL
```

라인 13: `((6 + f(c)) / b);` 수식 처리

단계 1: 중위→후위 변환 시작

```
// basic_interpreter.c L305-329
else if (firstword[0] == '(')
{
    if (foundMain)
    {
        int i = 0;
        int y = 0;

        MathStack->top = NULL;

        while (lineyedek[i] != '\0')
        {
            if (isdigit((unsigned char)lineyedek[i]))
            {
                postfix[y] = lineyedek[i]; // 숫자는 바로 추가
                y++;
            }
            else if (lineyedek[i] == ')')
            {
                if (!isStackEmpty(MathStack))
                {
                    postfix[y] = PopOp(MathStack);
                    y++;
                }
            }
            else if (lineyedek[i] == '+' || lineyedek[i] == '-' ||
lineyedek[i] == '*' || lineyedek[i] == '/')
            // 연산자 우선순위 처리
```

진행 상황:

```
입력: ((6 + f(a)) / a)
현재: "6" 처리 완료
postfix = "6"
MathStack = [+]
```

단계 2: 함수 호출 발견

```
// basic_interpreter.c L349-362
else if (isalpha((unsigned char)lineyedek[i]) > 0)
{
    int codeline = 0;
    int dummyint = 0;
    int retVal = GetVal(lineyedek[i], &codeline, STACK);
```

```
    if ((retVal != -1) && (retVal != -999))
    {   // 변수인 경우
        postfix[y] = (char)(retVal + 48);
        y++;
    }
    else
    {
        if (LastFunctionReturn == -999) // 함수 발견
        { // 함수 호출 처리
```

GetVal() 동작:

```
// basic_interpreter.c L470-485
static int GetVal(char exp_name, int* line, Stack* stck)
{
    Node* head;
    *line = 0;
    if (stck->top == NULL) return -999;
    head = stck->top;
    while (head) {
        if (head->exp_data == exp_name) {
            if (head->type == 1) return head->val;      // 변수
            else if (head->type == 2) {                 // 함수
                *line = head->line;
                return -1;
            }
        }
        head = head->next;
    }
    return -999;  // 못 찾음
}
```

```
STACK
[top] → [type=1, exp_data='c', val=4]
      → [type=1, exp_data='b', val=2]
      → [type=1, exp_data='a', val=1]
      → [type=4]
      → [type=2, exp_data='m', line=8]
      → [type=2, exp_data='f', line=1]  ← 여기서 발견. codeline = 1
      → NULL
```

단계 3: 함수 호출 준비

```
// basic_interpreter.c L336-355
if (LastFunctionReturn == -999) // 아직 함수 실행 안 함
{
    int j;
```

```
    // 1. 호출 마커 추가
    tempNode.type = 3;
    tempNode.line = curLine; // 13
    STACK = Push(tempNode, STACK);

    // 2. 인자 값 추출
    CalingFunctionArgVal = GetVal(lineyedek[i + 2], &dummyint, STACK);
    // lineyedek[i+2] = 'c' -> GetVal('c') -> 4

    // 3. 파일 포인터 리셋
    fclose(filePtr);
    filePtr = fopen(argv[1], "r");
    curLine = 0;


    // 4. 함수 정의 라인으로 이동
    for (j = 1; j < codeline; j++)
    {
        fgets(dummy, 4096, filePtr);
        curLine++;
    }

    WillBreak = 1; // 현재 루프 중단
    break;
}
```

최종 스택:

```
[top] → [type=3, line=13]                ← 함수 호출 마커 (새로 추가)
      → [type=1, exp_data='c', val=4]    ← 인자로 전달될 값
      → [type=1, exp_data='b', val=2]
      → [type=1, exp_data='a', val=1]
      → [type=4]
      → [type=2, exp_data='m', line=8]
      → [type=2, exp_data='f', line=1]
      → NULL
```

상태 변화:

```
curLine: 13 → 0
filePtr: line 13 → line 1로 리셋
CalingFunctionArgVal = 4
WillBreak = 1 (다음 while문에서 Phase 3 시작)
```

Phase 3: 호출된 함수 실행

라인 1: `function f(int a)` 재처리

```
// basic_interpreter.c L276-304
else if (my_stricmp("function", firstword) == 0) {
    firstword = strtok(NULL, " ");  // "f(int"

    // 함수 등록 (중복이지만 처리됨)
    tempNode.type = 2;
    tempNode.exp_data = firstword[0];  // 'f'
    tempNode.line = curLine;
    tempNode.val = 0;
    STACK = Push(tempNode, STACK);

    // main이 아닌 경우 매개변수 처리
    if (firstword[0] == 'm' && firstword[1] == 'a' &&
        firstword[2] == 'i' && firstword[3] == 'n') {
        foundMain = 1;
    }
    else {
        if (foundMain) {
            firstword = strtok(NULL, " ");  // "a)"
            tempNode.type = 1;                // 변수로 등록
            tempNode.exp_data = firstword[0]; // 'a'
            tempNode.val = CalingFunctionArgVal; // 4 (전달받은 값)
            tempNode.line = 0;
            STACK = Push(tempNode, STACK);
        }
    }
}
```

스택 변화:

```
[top] → [type=1, exp_data='a', val=4]   ← f의 매개변수 a
      → [type=2, exp_data='f', line=1]
      → [type=3, line=13]
      → [type=1, exp_data='c', val=4]
      → [type=1, exp_data='b', val=2]   ← main의 변수들
      → [type=1, exp_data='a', val=1]
      → [type=4]
      → [type=2, exp_data='m', line=8]
      → [type=2, exp_data='f', line=1]
      → NULL
```

라인 2: `begin` 마커

```
tempNode.type = 4;
STACK = Push(tempNode, STACK);
```

스택:

```
[top] → [type=4]
      → [type=1, exp_data='a', val=4]   ← f의 매개변수 a
      → [type=2, exp_data='f', line=1]
      → [type=3, line=13]
        ...
      → NULL
```

라인 3-4: `int b = 6;`, `int c = 2;` 처리

```
tempNode.type = 1;
tempNode.exp_data = 'b';
tempNode.val = 6;
STACK = Push(tempNode, STACK);

tempNode.exp_data = 'c';
tempNode.val = 2;
STACK = Push(tempNode, STACK);
```

스택:

```
[top] → [type=1, exp_data='c', val=2]
      → [type=1, exp_data='b', val=6]
      → [type=4]
      → [type=1, exp_data='a', val=4]   ← f의 매개변수 a
      → [type=2, exp_data='f', line=1]
      → [type=3, line=13]
        ...
      → NULL
```

라인 5: `((b+c)/a);` 계산

중위→후위 변환

```
// basic_interpreter.c L314-394
while (lineyedek[i] != '\0') {
    // 'b' 처리
    if (isalpha((unsigned char)lineyedek[i]) > 0) {
        int codeline = 0;
        int dummyint = 0;
        int retVal = GetVal('b', &codeline, STACK);
        // GetVal('b') → 6 (스택에서 찾음)

        if ((retVal != -1) && (retVal != -999)) {
            postfix[y] = (char)(retVal + 48);  // '6'
            y++;
```

```
        }
    }
    // 'c' 처리
    // GetVal('c') → 2
    // postfix[y++] = '2'

    // '+', '/', 'a' 처리...
}
```

변환 결과: `"6 2 + 4 /"`

```
입력:  ((b+c)/a)
        ↓
후위:  6 2 + 4 /
```

후위 표기식 계산

```
// basic_interpreter.c L396-432
if (WillBreak == 0) {
    // 남은 연산자 처리
    while (!isStackEmpty(MathStack)) {
        postfix[y] = PopOp(MathStack);
        y++;
    }

    postfix[y] = '\0';

    // 후위 표기식 계산
    i = 0;
    CalcStack->top = NULL;
    while (postfix[i] != '\0') {
        if (isdigit((unsigned char)postfix[i])) {
            CalcStack = PushPostfix(postfix[i] - '0', CalcStack);
        }
        else if (postfix[i] == '+' || postfix[i] == '-' ||
                 postfix[i] == '*' || postfix[i] == '/') {
            val1 = PopPostfix(CalcStack);
            val2 = PopPostfix(CalcStack);

            switch (postfix[i]) {
                case '+': resultVal = val2 + val1; break;
                case '-': resultVal = val2 - val1; break;
                case '/': resultVal = val2 / val1; break;
                case '*': resultVal = val2 * val1; break;
            }
            CalcStack = PushPostfix(resultVal, CalcStack);
        }
        i++;
    }
```

```
    LastExpReturn = CalcStack->top->val;
}
```

계산 과정:

```
"6 2 + 4 /"
 ↓
1. push(6): [6]
2. push(2): [2]→[6]
3. '+': pop 2,6 → 6+2=8 → push(8): [8]
4. push(4): [4]→[8]
5. '/': pop 4,8 → 8/4=2 → push(2): [2]

최종: LastExpReturn = 2
```

라인 5: end 처리 (함수 반환)

```
// basic_interpreter.c L212-247
else if (my_stricmp("end", line) == 0) {
    if (foundMain) {
        int sline;
        int foundCall = 0;
        tempNode.type = 5;
        STACK = Push(tempNode, STACK);

        sline = GetLastFunctionCall(STACK);  // 13 반환

        if (sline == 0) {
            printf("Output=%d", LastExpReturn);  // main의 end
        }
        else {
            int j;
            // 함수 반환 처리
            LastFunctionReturn = LastExpReturn;  // 2 저장

            // 파일 포인터를 호출 위치로 복귀
            fclose(filePtr);
            filePtr = fopen(argv[1], "r");
            curLine = 0;
            for (j = 1; j < sline; j++) {  // line 13로 이동
                fgets(dummy, 4096, filePtr);
                curLine++;
            }

            // 스택 정리 (end ~ call 마커까지 pop)
            while (foundCall == 0) {
                Pop(&tempNode, STACK);
                if (tempNode.type == 3) foundCall = 1;
```

```
            }
        }
    }
}
```

GetLastFunctionCall() 동작:

```
// basic_interpreter.c L460-468
static int GetLastFunctionCall(Stack* stck) {
    Node* head = stck->top;
    while (head) {
        if (head->type == 3) return head->line;  // call 마커 찾기
        head = head->next;
    }
    return 0;  // 못 찾으면 main의 end
}
```

스택 탐색:

```
[top] → [type=1, exp_data='c', val=2]
      → [type=1, exp_data='b', val=6]
      → [type=4]
      → [type=1, exp_data='a', val=4]
      → [type=2, exp_data='f', line=1]
      → [type=3, line=13]   ← 여기 return 13
        ...
      → NULL
```

스택 정리 (Pop 반복):

```
[top] → [type=1, exp_data='b', val=2]   ← main의 변수들만 남는다.
      → [type=1, exp_data='a', val=1]
      → [type=4]
      → [type=2, exp_data='m', line=8]
      → [type=2, exp_data='f', line=1]
      → NULL
```

end 부터 함수 호출 마커까지 pop한다.

상태 변화:

```
LastFunctionReturn = 2
curLine = 12 (line 13로 이동 준비)
filePtr: line 1 → line 13로 리셋
```

Phase 4: main 복귀 및 종료

라인 13: 수식 계산 재개

함수 반환값 처리

```
// basic_interpreter.c 384-390
else {  // LastFunctionReturn != -999
    postfix[y] = (char)(LastFunctionReturn + 48);  // 2 + 48 = '2'
    y++;
    i = i + 3;  // "f(c)" 건너뛰기
    LastFunctionReturn = -999;  // 초기화
}
```

중위→후위 변환 완료:

```
입력: ((6 + f(c)) / b)
           ↓
후위: 6 2 + 2 /
      └─┘
    f(c)의 반환값 2가 삽입됨
```

후위 표기식 계산

```
// basic_interpreter.c L396-432
if (WillBreak == 0) {
    while (!isStackEmpty(MathStack)) {
        postfix[y] = PopOp(MathStack);
        y++;
    }

    postfix[y] = '\0';

    i = 0;
    CalcStack->top = NULL;
    while (postfix[i] != '\0') {
        if (isdigit((unsigned char)postfix[i])) {
            CalcStack = PushPostfix(postfix[i] - '0', CalcStack);
        }
        else if (postfix[i] == '+' || postfix[i] == '-' ||
                 postfix[i] == '*' || postfix[i] == '/') {
            val1 = PopPostfix(CalcStack);
            val2 = PopPostfix(CalcStack);

            switch (postfix[i]) {
                case '+': resultVal = val2 + val1; break;
```

```
                case '-': resultVal = val2 - val1; break;
                case '/': resultVal = val2 / val1; break;
                case '*': resultVal = val2 * val1; break;
            }
            CalcStack = PushPostfix(resultVal, CalcStack);
        }
        i++;
    }

    LastExpReturn = CalcStack->top->val;
}
```

계산 과정:

```
"6 2 + 2 /"

1. push(6): [6]
2. push(2): [2]→[6]
3. '+': pop 2,6 → 6+2=8 → push(8): [8]
4. push(2): [2]→[8]
5. '/': pop 2,8 → 8/2=4 → push(4): [4]

최종: LastExpReturn = 4
```

라인 14: `end` 처리 (main 종료)

```
// basic_interpreter.c L220-224
sline = GetLastFunctionCall(STACK);  // 0 반환 (call 마커 없음)

if (sline == 0) {
    printf("Output=%d", LastExpReturn);  // Output=4
}
```

GetLastFunctionCall() 동작:

```
[top] → [type=1, exp_data='b', val=2]
      → [type=1, exp_data='a', val=1]
      → [type=4]
      → [type=2, exp_data='m', line=8]
      → [type=2, exp_data='f', line=1]  ← type=3 없음. return 0
      → NULL
```

프로그램 종료:

```
// basic_interpreter.c L439-445
fclose(filePtr);
STACK = FreeAll(STACK);
printf("\nPress a key to exit...");
getchar();
return 0;
```

정리

실행 흐름 요약

```
Phase 1: 함수 등록              → foundMain = 0, 함수만 스택에 추가
Phase 2: main 실행              → foundMain = 1, 함수 호출 감지
Phase 3: 호출 함수 실행         → 파일 포인터 리셋, 매개변수 전달
Phase 4: main 복귀              → 반환값으로 계산 완료, 결과 출력
```

주요 변수 역할

변수	역할
`foundMain`	main 발견 여부 (0→1 전환점)
`curLine`	현재 라인 번호
`LastExpReturn`	마지막 수식 결과
`LastFunctionReturn`	함수 반환값 (-999: 미호출)
`CalingFunctionArgVal`	함수 인자 전달용
`WillBreak`	함수 호출 시 루프 중단 플래그

스택 활용

1. **심볼 테이블**: 변수/함수 저장 (LIFO 검색)
2. **실행 컨텍스트**: begin/call/end 마커로 스코프 관리
3. **함수 호출**: call 마커로 복귀 지점 저장

인터프리터 실행 방법

1. 소스 코드 다운로드

이 레포지토리를 로컬에 복제한다.

```
git clone https://github.com/[Your-Username]/[Your-Repository-Name].git
```

2. C 코드 컴파일

소스 코드가 준비되면, 터미널에서 gcc 컴파일러를 사용하여 `basic_interpreter.c` 파일을 컴파일한다. 다음 명령을 실행하여 `basic_interpreter`라는 이름의 실행 파일을 생성한다.

```
gcc basic_interpreter.c -o basic_interpreter
```

3. 인터프리터 실행

생성된 실행 파일을 통해 `.spl` 언어 파일을 분석할 수 있다. 프로그램의 첫 번째 인자로 `input1.spl` 파일의 경로를 전달하여 실행한다. 레포지토리 내에 있는 다른 파일도 실행 가능하다.

```
./basic_interpreter input1.spl
```

4. 실행 결과 확인

`input1.spl` 파일을 성공적으로 실행하면, 터미널에 다음과 같은 문구가 출력된다.

```
Output=4
Press a key to exit...
```

`Outputs.txt`에 나와있는 내용과 일치하다.