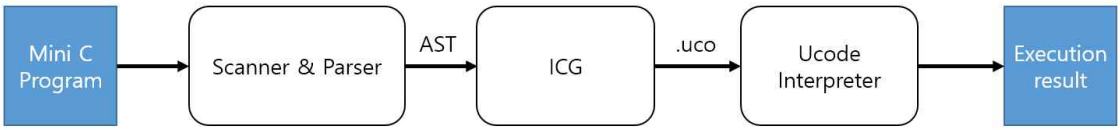
목차

1. 개요

파서(Parser) 과정에서는 주어진 소스 코드의 구문 분석 과정을 거쳤다. 이 과정에서 우리는 AST(Abstract Syntax Tree)를 생성하였고, 이를 출력하는 작업까지를 진행하였다. 이로서 구문 분석 단계까지 진행하면서 중간 코드 생성기(Intermediate Code Generator:ICG)에 필요한 AST를 넘겨줬다.

그 전에 중간 코드 생성기(Intermediate Code Generator:ICG)란 무엇인지 알고 넘어가자. ICG는 컴파일러 과정에서 심각하게 복잡한 과정을 중간에 거쳐 간다는 개념으로 만든 것이며, 이것을 만듦으로서 생기는 장점은 문법적 에러가 있는 소스 프로그램에 대해 불필요한 코드 생성 과정을 생략할 수 있으며, 트리 형태로 구문의 의미 정보를 모두 가지고 있으므로 파싱 순서와 무관하게 코드를 생성할 수 있어 코드 생성기의 구현이 유용하다는 장점이 있다.

이러한 역할을 하는 ICG는 아래와 같은 구조로 사용된다고 보면 되겠다.



▲ 그림1. Mini-C 컴파일러의 전체 과정

즉, ICG는 Input으로는 AST가 되겠고, 이를 이용해 중간 언어를 생성하면 Output으로 “.uco”파일에 결과를 넣어주면 된다. 그러면 Ucode-Interpreter는 이 파일(.uco)을 읽고 프로그램을 실행하는 구조로 될 것이다.

그리고 해당 과제를 수행하면서 이용하게 된 것은 Github다. Git과 Visual Studio Code의 터미널 기능을 이용해서 실시간으로 소스코드 버전 관리 및 전체적인 소스코드 저장, 문서화 등을 진행하였으며 밑에서 설명할 실행 방법 또한 본인의 Github Repository를 이용한 방법으로 설명을 하게 될 것이다.

https://github.com/KeonHeeLee/MiniC-Compiler

▲ 링크1. 해당 과제를 수행할 때 사용한 원격 저장소

전체적으로는 어떻게 ICG를 설계하고 구현하였으며, 또한 앞의 과제에서 수행한 추가 문법은 어떻게 처리하였으며, 이 소스 코드를 실행하는 방법, 결과 스냅샷 및 추가 문법에 대한 .uco파일 설명을 서술할 것이다.

2. 문제의 핵심 해결 방안 설계

위의 개요에서 설명했듯이, 중간 언어 생성기(ICG)의 역할은 파서로부터 얻어온 AST를 분석하고 이를 Ucode-Interpreter가 해석하고 실행할 수 있게 중간 언어를 생성하는 것이 목표다. 이러한 프로그램을 어떻게 설계하고 제작하였는 지에 대해서 설명할 것이다.

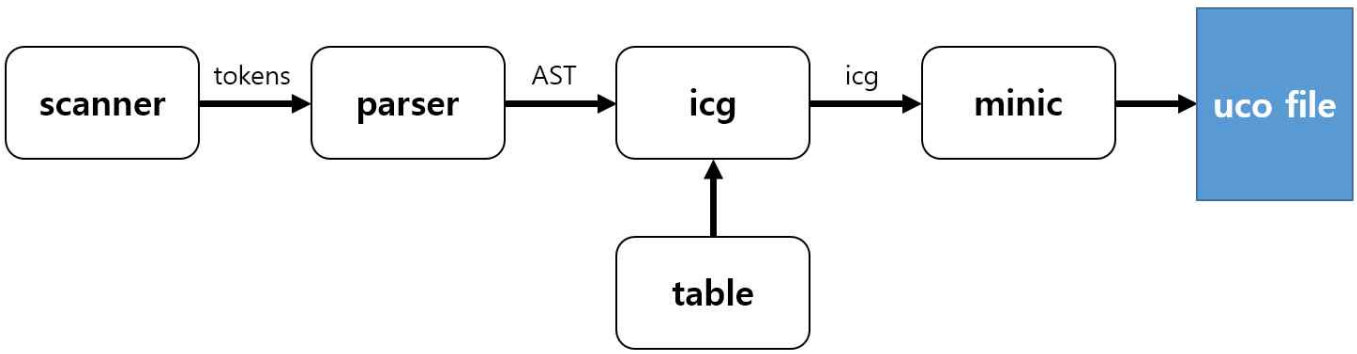
1) 중간 언어 생성기(ICG) 설계

가장 먼저 중간 언어를 생성할 때 고려했던 점은, AST와 기록할 .uco파일을 Input으로 넣었던 것이다. 그리고 중간 언어 기록이 다 된 .uco파일을 Output으로 둔다. 즉, 이러한 기능을 다루는 모듈이 바로 핵심 모듈으로 해야 할 것이다. (icg.h - codeGen 함수)

그리고 이제 AST를 분석해야한다. 트리를 탐색해야하기 때문에, root로부터 포인터를 이용해서 접근을 해야하며, 각 선언문이나 연산자, 흐름문 등의 내용들에 따라서도 실제로 트리를 읽는 흐름이 달라지기 때문에 여기서 각 기능군별로 나눠서 함수화시키는 것이 소스코드 가독성이든, 모듈별 테스트든 유리하게 될 것이다.

그 후에 이렇게 분석을 하였으면, 이에 따라서 uco파일에 분석 결과를 써줘야 할 것이다. U-code의 문법에 따라서도 오프셋을 3개로 할건지, 2개로 할건지 혹은 주소를 기록하는지, 라벨 정보를 기록하는지 각각 상이하기 때문에 이에 따라서 함수를 나눠주는 것이 중요하다.

마지막으로 흐름문이나 함수 선언에 따라서 라벨을 구분해줘야 하기 때문에 이에 따른 함수나 구조체도 선언해줘야 한다. 이렇게 구성을 하면 전체 모듈 구성은 아래와 같이 할 수 있다.



▲ 그림2. 전체 모듈 설계

parser 및 scanner부분은 지난 과제(과제3: 파서 작성)에서 수행했으므로 자세한 설명은 생략하도록 하겠으며, 여기서는 icg 관련 모듈과 table 모듈에 대해서 설명하도록 하겠다.

우선 table 모듈에선 SymbolTable과 FlowTable, 그리고 이것들을 다룰 수 있는 함수들로 구성 되어있다. 그리고 두 구조체(SymbolTable과 FlowTable)는 아래와 같이 정의할 수 있다.

```
typedef struct SymbolInfo {
        char *id;
        typeQualifier qual;
        typeSpecifier spec;
        int width;
        int offset;
        int initialValue;
} SymbolInfo;

typedef struct SymbolTable {
        int count;
        int offset;
        int base;
        SymbolInfo symbols[SYM_TABLE_SIZE];
        struct SymbolTable *parent;
} SymbolTable;
```

```
typedef struct FlowInfo{
        char* startLabel;
        char* endLabel;
        typeQualifier qual;
} FlowInfo;
typedef struct FlowTable {
        int count;
        FlowInfo flows[SYM_TABLE_SIZE];
} FlowTable;
```

▲ 표1. SymbolTable과 FlowTable 구조체 정의 및 그에 관련된 구조체 추가선언

먼저 SymbolTable을 보자. SymbolTable은 심벌의 이름, Qualifier 타입, Specifier 타입, offset 등의 정보를 포함한 SymbolInfo 구조체를 배열형태로 가지고 있으며, SymbolTable 자체로는 링크드 리스트 노드 형태로 설계하였다. 실제로 이 심벌 정보와 심벌들이 어느 블록에 속하며, offset을 계산하는 등의 작업을 여기서 해주게 설계하였다.

FlowTable의 경우는 Flow의 정보를 담는 FlowInfo 구조체를 배열형태로 가지고 있게 설계를 하였다. 이는 실제로 흐름문을 사용할 때, 어디가 시작이고 끝인지를 제어해주기 위해서 설계하였다.

그 외에 process(예 - processSimpleVariable())계열의 함수들은 AST를 읽어가면서 어떤 중간언어를 생성할 건지 규칙을 정하는 작업을 하고, 이에 따라서 각각 함수들로 나뉘어져있다. 그리고 emit계열 함수들은 uco파일에 process과정에서 만든 중간 언어를 써주는 역할을 하게 설계하였다.

그 후에 icg.c파일 내의 codeGen()이라는 함수를 통해서 uco파일에 중간언어를 모두 기록하고, minic.y 모듈이 종료하는 과정을 거치게끔 하였고, 이 파일(.uco)은 과제1(Ucode Interpreter)에서 분석한 소스코드를 컴파일 해서 나온 ucodei란 실행파일로 실행하면 프로그램이 실행되는 절차로 구성되어있다.

2) ICG 제작 및 고려해야할 부분

처음에는 부록에 서술된 참고자료[1]을 참고하면서 소스 코드를 그대로 적어가면서 틀을 마련하였다. 하지만 여기서는 구현이 안된 것이 있는데, 바로 심벌 테이블과 그것과 관련된 함수, 그리고 필자의 경우는 필요에 따라서 흐름문에 관련된 테이블이 구현되지 않았기 때문에 참고자료[3]을 참고하면서 추가로 구현하게 되었으며, 이는 실제 소스코드로는 (table.c와 table.h)다.

특히 SymbolTable의 경우는 설계 과정에서 링크드 리스트로 설계를 하였기 때문에 헤더 포인터의 선언과 초기화(createSymbolTable 함수) 노드들을 연결(insertSymbol 함수), 노드 찾는 연산(lookupSymbol)로 구성하였다.(참고로 FlowTable과의 구분을 위해서 교재(참고자료[1])에서 선언해둔 이름을 바꾼 것이다.)
그리고 무엇보다 링크드 리스트 구조는 포인터 관련해서 오류가 나올 수도 있기 때문에 if문으로 검사를 하면서 값이 없을 경우엔 return;이나 printf 함수로 잘못되었음을 알려주는 경고메세지도 띄우게 구현하였다.

FlowSymbol은 Flow초기화(initFlowTable 함수), 위쪽의 flow로 이동하는 topFlow함수, 그리고 flow의 추가와 제거에 관련된 pushFlow함수와 popFlow함수가 있다. 이는 실제로 흐름문을 발견할 경우, 흐름문의 시작과 끝이 어디인지 제어하기 위한 역할을 수행하는 함수들로 구현되었다. 그리고 count라는 구조체 멤버를 통해서 오류 처리 및 flow 이동을 할 수 있다.

이제는 icg계열 모듈을 설명하도록 하겠다. 먼저 실제 minic.y에서 사용되는 함수는 codeGen이다. 이 함수가 icg 모듈에선 핵심역할을 하게 된다. 이 함수는 AST와 FilePointer를 매개변수로 받아서, 먼저 AST의 root부터 트리들을 탐색하면서 AST의 노드들을 탐색하는 과정을 거친다. 여기서 내부의 process함수와 emit 함수를 같이 사용하면서 uco파일에 중간 언어를 기록하게 된다. 이러한 과정이 끝났으면, 아래의 중간언어를 선언 후 프로그램이 끝난다.

```
bgn     <number of static variable>
ldp
call     main
end
```

▲ 표2. codeGen 함수에서 마지막 부분에 붙이는 중간 언어 키워드들

위에서 말한건 codeGen에서는 process계열 함수를 통해서 AST트리르 전부 탐색한다고 서술하였다. 이 과정이 어떻게 구현되었는지 확인해보자. 먼저 processDeclaration 함수를 보자. 이 함수는 선언문과 관련된 함수이며, 선언문이 아닐 경우는 오류 출력 후 함수를 탈출하는 방법으로 구현되어있다. 여기서 typeSpecifier와 typeQualifier의 enum형 변수를 사용해서 void형인가 int형인가 그리고 const인가 변수인가 등의 구분을 하게 구현하였다.

그리고 여기서 AST를 자식 노드쪽으로 이동하면서 탐색을 하게 된다. 이렇게 하면 AST내에서는 brother와의 관계는 문장이 이어지게 구성이 되어있기 때문에, 변수명이나 배열이름을 파악할 수 있게 된다. 만약 이 둘중 아무것도 아니면 오류 메시지를 출력하게 된다.

processSimpleVariable함수와 processArrayVariable함수는 만약 여기서 요구하는 token.number가 아닐 경우 혹은 현재 AST brother가 없을 경우는 함수가 종료되며, 정상적일 경우는 symbolTable에 해당 정보를 저장하고 함수가 종료된다.

int; float; void a;	왼쪽과 같은 경우가 바로 위의 함수에서 오류로 처리하는 부분이다.

▲ 표3. processSimpleVariable함수와 processArrayVariable함수에서의 오류 case

함수 헤더(processFuncHeader 함수)의 경우 codeGen에서 변수타입인가 함수타입인가 알아볼 때, 같이 수행을 하게 된다. int형이나 void형이 아닐 경우는 miniC에서 언급하는 것이 아니므로 오류를 출력하게 하며, 그 외의 다른 키워드가 있는지를 AST 노드를 탐색하면서 검사를 한다. 그리고 insertSymbol함수를 수행한다.

processFunction 함수는 우선 SymbolTable부터 생성을 하게 된다. 그 후에는 AST를 쭉 탐색하면서 (paramter 처리) Parameter가 있을 경우는 processDeclaration() 함수를 사용해서 해당 블록에 변수나 배열들을 불러오게 구현하였다. 그리고 proc이란 키워드를 파일에 쓰고, 해당 함수에서 생성한 SymbolTable을 탐색하면서 매개변수 또한 emitSym함수를 통해서 sym 키워드를 파일에 써준다. processStatement함수가 수행된 이후에는 returnFlag를 통해서 오류처리를 하고, ret과 end 키워드를 파일에 써준다.

processStatement 함수가 수행된다. 문장 자체를 다룬다. 그래서 범위가 상당히 많은데, 크게는 Statement에서 Statement로 갈 경우에는 재귀적으로 함수를 호출, return 키워드를 발견했을 경우엔 반환 값이 있는지 보고 retv일지 ret일지 판단, 그 외에는 흐름문이나 제어문이 있다. 이것 외의 키워드일 때는 오류 메세지를 출력하게 하였다.

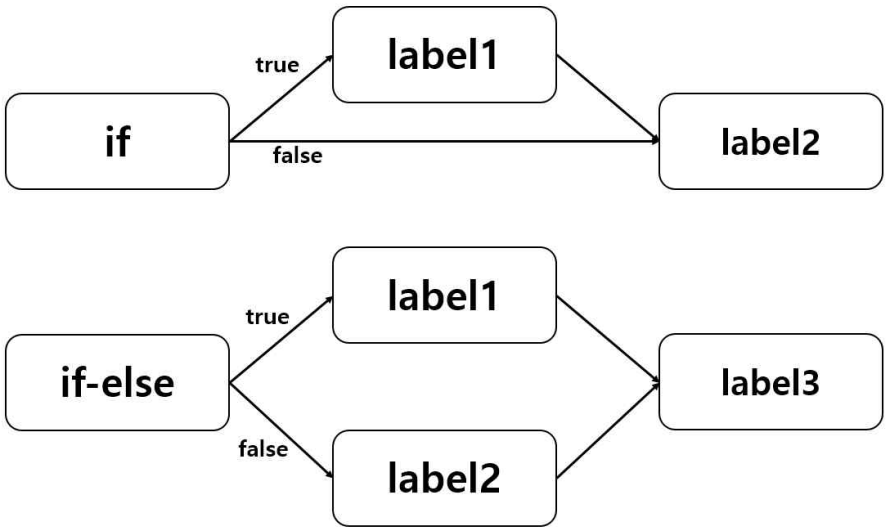
먼저 흐름문이나 제어문을 언급하기전에 processOperator 함수부터 알아보자. 여기서는 각 키워드 군마다 작동하는 방법이 다르기 때문에, 상당히 복잡할 수 있지만, 상당수는 lhs와 rhs를 통해서 선언문의 모호성 제거 및 우선순위를 정하고, 이를 emit 계열 함수로 uco 파일에 써주는 방법을 취하고 있다.(일반연산)

INDEX 키워드의 경우는 indexExp가 만약 nonterm일 경우는 재귀적으로 호출하고 아닐 경우는 index에 따라서 중간언어를 생성하게 끔, rv_emit함수를 호출하였다. 그리고 심벌을 통해 라벨 정보를 얻어와서 블록에 따라서 lda함수를 호출, 그리고 add를 함으로써, 배열내의 원소를 얻어올 수 있다.

++,-- 연산의 경우는 포인터가 없거나 IDENT 심벌이 없을 경우 오류로 걸러내고, 그것이 아닐 경우는 lookupSymbol 함수를 이용해서 연산 대상이 되는 변수의 블록정보를 얻어온 후에, emit계열 함수로 중간 코드를 기록한다.

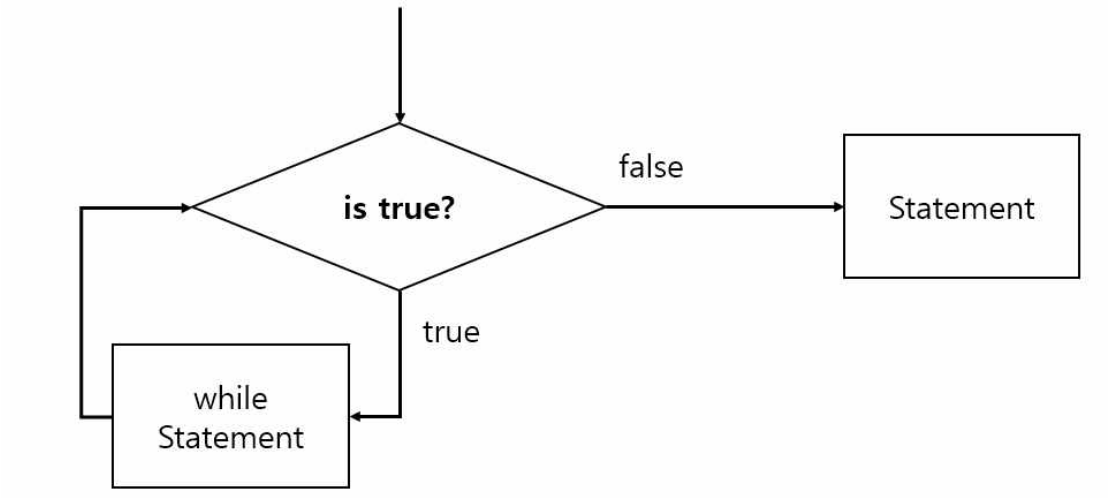
CALL 심벌의 경우는 미리 선언된 함수인지 먼저 찾아보고, ldp를 선언 후, 매개변수로 사용할 변수들을 채워준다. 그리고 noAruements 변수를 통해서 카운트해준다. 만약 noArguments가 0이 아닐 경우는 매개변수 수가 안맞다는 말이므로 오류처리를 하고 이러한 내용들을 uco파일에 써준다.

다시 processStatement로 넘어와서 제어문부터 설명하도록 하겠다. 우선 for문이나 switch, default, break, continue 같은 추가 구현할 키워드는 다음항목에서 설명하도록 하겠으며, 여기는 기본적인 키워드부터 설명하겠다. 우선 여기서 EXP_ST는 위에서 설명한 processOperator함수를 호출하고, 그 외의 키워드에는 IF_ST, IF_ELSE_ST, WHILE이 있다. IF의 경우는 우선 processCondition함수를 통해서 조건문에 해당하는 중간언어를 기록한 후에, fjp와 라벨명을 적어주면 된다. 그 후엔 조건문 내의 문장들을 적어주고 다음 라벨로 넘어가게 해주면 된다.(fjp일 경우는 다음 라벨로 바로 넘어간다.) IF_ELSE_ST의 경우는 IF_ST의 경우에 비해서 else부분이 if가 아닌 나머지를 뜻하기 때문에 ujp로 표기해주면 된다.



▲ 그림3. if-else문에서 고려해야할 사항

WHILE_ST 키워드는 while문을 나타내는 반복문을 말한다. 이 경우는 Flow문이기 때문에 pushFlow함수를 이용해서 FlowTable에 넣어준다. 그리고 라벨 2개를 얻어와서 사용하면 되는데, 하나는 조건문을 나타내는 while문의 loop 조건에 따라 이동할 라벨, 다른 하나는 탈출할 라벨을 기록한다. 그리고 popFlow함수를 사용함으로써, 흐름문을 탈출했다는 것을 명시, 그리고 테이블의 데이터를 비운다. 실제 코드에선 label1이 조건문 시작부분을 나타내는 라벨이고, label2가 조건문이 끝난 후의 라벨이다. 이에 따라서 소스코드를 구현하면 되고, 전체적인 구조는 아래와 같다.



▲ 그림4. while문에서 고려해야할 사항

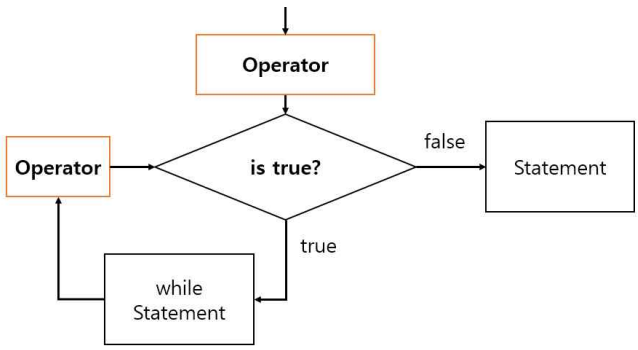
3) 추가 구현한 부분

3-1) for

for문의 경우는 총 3가지의 동작이 이루어진다. 변수 초기화, 조건문 검사, 연산 이렇게 3가지의 동작이 이루어지며, while문의 구조에서 단지 선언부분과 연산부분만 추가로 구현하면 된다는 것이다. 즉, while문이 시작되기전에 변수를 초기화하고, while문을 돌면서 연산을 하는 것만 추가하면 for문은 구현이 된 것이다. 구현 및 flowchart는 아래와 같다.

```
case FOR_ST:
{
        char label1[LABEL_SIZE], label2[LABEL_SIZE];

        for(p = ptr->son->son; p; p = p->brother)
                processOperator(table, p);
        genLabel(label1);
        genLabel(label2);
        pushFlow(flowTable, LOOP_QUAL, label1, label2);
        emitLabel(label1);
        processCondition(table, ptr->son->brother->son);
        emitJump("fjp", label2);
        processStatement(table,ptr->son->brother->brother->brother);
        for(p = ptr->son->brother->brother->son; p; p= p->brother)
                processOperator(table,p);
        emitJump("ujp",label1);
        emitLabel(label2);
        popFlow(flowTable);
}
break;
```

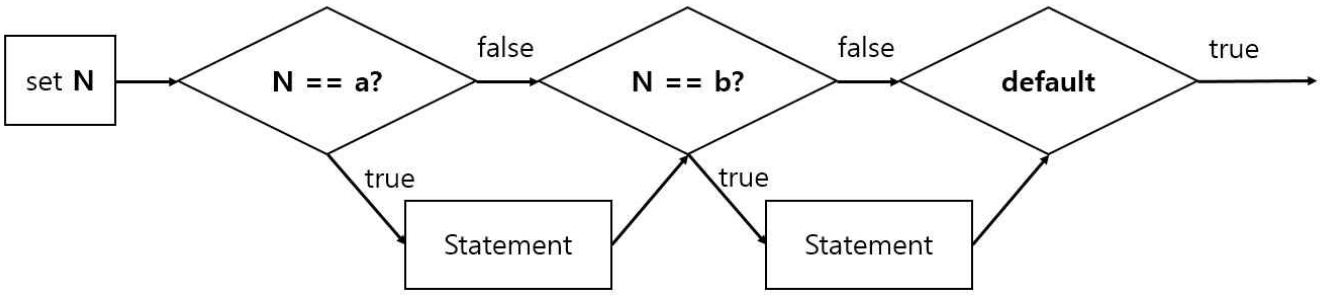


▲ 그림 5,6. for문 구현 소스코드 및 flowchart

3-2) switch case default

switch문은 if-else문을 응용해서 만든다. 우선 변수 하나를 switch문의 조건 검색용으로 쓰기 위해 적재를 한다. 그리고 pushFlow함수를 선언해서 FlowTable을 추가해주고, case문마다 “eq” 키워드를 활용해서 맞는지 확인하고 맞다면, 해당 라벨 내부의 소스 코드를 수행하는 중간언어를 생성해주고, 다음 라벨로 넘어가서 다시 case문을 검사하는 구조를 가지게하면 된다. default는 if-else문의 else와 같이 “ujp”라는 키워드를 활

용하여, 무조건 거쳐가게 사용하면 된다. 그리고 switch문이 끝나면 popFlow함수 선언, 그리고 탈출할 라벨을 파일에 적어주고 끝내면 된다.



▲ 그림 7. switch문 flowchart

```
case SWITCH_ST:
{
        char label1[LABEL_SIZE], label2[LABEL_SIZE], label3[LABEL_SIZE];

        genLabel(label2);
        for(p = ptr->son->brother; p; p = p->brother) {
                switch(p->token.number) {
                        case CASE_LST:
                                genLabel(label1);
                                pushFlow(flowTable, SWITCH_QUAL, label1, label2);
                                for(q = p->son; q; q = q->brother) {
                                        switch(q->token.number) {
                                                case CASE_ST:
                                                        processCondition(table, ptr->son->son->son);
                                                        emit1("ldc", atoi(q->son->token.value));
                                                        emit0("eq");
                                                        emitJump("tjp",label1);
                                                break;
                                                case DEFAULT_ST:
                                                        emitJump("ujp",label1);
                                                break;
                                        }
                                }
                                genLabel(label3);
                                emitJump("ujp",label3);
                        break;
                        case CASE_EXP:
                                emitLabel(label1);
                                for(q = p->son; q; q=q->brother)
                                        processStatement(table,q);
                                popFlow(flowTable);
                                emitLabel(label3);
                        break;
                }
        }
        emitLabel(label2);
}
break;
```

▲ 그림 8. switch문 구현 소스코드

3-3) break continue

break와 continue는 흐름을 끊거나 돌려버리는 역할을 하는 키워드다. 이는 오히려 크게 어렵진 않다. break문의 경우는 topFlow함수를 이용해서 상위 라벨의 정보를 얻어온 후에 상위 라벨의 정보가 담긴 endLabel로 ujp를 하면 된다.

continue의 경우는 반대로 findFlow함수를 이용해서 현재 라벨 정보가 담긴 startLabel을 얻어온다. 그리고 startLabel로 ujp를 하게 되면, 밑의 선언문들을 무시하고 루프를 돌게 된다. 이러한 경우는 에러 처리를 하기

위해서 startLabel이 0일 경우는 어떠한 FlowTable의 라벨도 지정되지 않았다는 뜻이고, 이는 다른 말로는 loop문을 도는 것이 아니기 때문에 오류로 처리했다. 구현 소스코드는 아래와 같다.

```
case BREAK_ST:
        topFlow(flowTable, &startLabel, &endLabel);
        if(startLabel == 0) {
                printf("break error!");
                return;
        }
        emitJump("ujp",endLabel);
        break;
```

```
case CONTINUE_ST:
        findFlow(flowTable,LOOP_QUAL, &startLabel, &endLabel);
        if(startLabel == 0) {
                printf("continue Error!");
                return;
        }
        emitJump("ujp",startLabel);
        break;
```

▲ 그림 9,10. break 및 continue문 구현 소스코드

3. 실행 환경 및 방법

실행 환경은 아래와 같다.

OS	Ubuntu 16.04 LTS
IDE	Visual Studio Code v1.22.2
Compiler	gcc, g++ (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0

▲ 표. 과제 실행 환경

실제 실행방법은 위와 같으며, 리눅스 환경에서 개발 및 테스트를 진행하였기에, 리눅스 기준으로 설명을 하도록 하겠다. 우선 위의 사항들을 모두 확인해보기 위해서 먼저 설치를 해야할 것이 있는데, 그것은 바로 flex와 bison이다. 만약 설치가 되지 않았을 경우 아래의 명령어를 터미널에 입력해주도록 하자.

```
KeonHeeLee@ubuntu:~$ sudo apt-get update
KeonHeeLee@ubuntu:~$ sudo apt-get install flex
KeonHeeLee@ubuntu:~$ sudo apt-get install bison
```

▲ 표. 실행 전 설치 해야 될 apt : flex, bison

그리고 실제로 과제는 깃허브(Github)에 올려서 원격 저장소로 활용하고 있다. 개요에서 설명했듯이 링크를 통해 다운로드를 받아도 되고, 그것이 아니라면 아래의 명령어를 입력해주도록 하자.

```
KeonHeeLee@ubuntu:~$ sudo apt-get install git (만약 설치가 안됐을 경우 사용하면 됨)
KeonHeeLee@ubuntu:~$ git clone https://github.com/KeonHeeLee/MiniC-Compiler
KeonHeeLee@ubuntu:~$ cd MiniC-Compiler
KeonHeeLee@ubuntu:~$ sudo chmod +0777 setup.sh
KeonHeeLee@ubuntu:~$ ./setup.sh
```

▲ 표17. 깃허브를 통해 다운로드 및 컴파일하는 방법

참고로 깃허브로 다운로드를 받았을 경우 Repository의 구성 요소는 아래와 같다.

```
MiniC-Compiler-master
    ├ image
    ├ src
    ├ test
    ├ .gitignore
    ├ README.md
    └ setup.sh
```

▲ 표. Github repository 구성요소

test 디렉토리에 테스트코드 및 실행결과들을 모두 저장되어 있다. (5개의 예제가 있다.) 그리고 여기서 **src/icg** 라는 디렉토리에 들어가면 원하는 소스코드가 있다. 각 파일들을 설명하면 아래와 같다.

```
src/icg
    ├ Makefile      /* 해당 소스코드들을 컴파일 시키기 위한 Makefile */
    ├ icg.c         /* 중간언어 생성기 코드 (c 파일) */
    ├ icg.h         /* 중간언어 생성기 코드 (헤더 파일) */
    ├ parser.h      /* AST 관련 헤더 파일 */
    ├ parser.c      /* AST 관련 c 파일 */
    ├ minic.y       /* AST 생성 및 프로그램의 메인 역할을 하는 소스 코드 */
    ├ scanner.l     /* LexicalAnalyzer 역할을 해주는 flex 파일 */
    ├ table.h       /* SymbolTable 혹은 FlowTable 관련 헤더 파일*/
    └ table.c       /* SymbolTable 혹은 FlowTable 관련 c 파일*/
```

▲ 표. src/icg 디렉토리 내부의 파일 설명

위의 과정들을 거쳤으면 소스 파일들이 있을 것이고, shell script(setup.sh) 파일을 통해서 중간언어 생성기와 ucode 인터프리터 소스코드를 모두 컴파일을 시켰기 때문에, 그냥 실행만 하면 된다. 실행 방법은 아래와 같다.

```
keonheelee@leekh:~/school/compiler$ git clone https://github.com/KeonHeeLee/MiniC-Compiler
'MiniC-Compiler'에 복제합니다...
remote: Counting objects: 285, done.
remote: Compressing objects: 100% (60/60), done.
remote: Total 285 (delta 22), reused 15 (delta 2), pack-reused 221
오브젝트를 받는 중: 100% (285/285), 363.36 KiB | 16.00 KiB/s, 완료.
델타를 알아내는 중: 100% (127/127), 완료.
연결을 확인하는 중입니다... 완료.
keonheelee@leekh:~/school/compiler$ cd MiniC-Compiler
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ls
image  README.md  setup.sh  src  test
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ sudo chmod +0777 setup.sh
[sudo] password for keonheelee:
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ls -l
합계 20
drwxrwxr-x 2 keonheelee keonheelee 4096  6월  8 18:10 image
-rw-rw-r-- 1 keonheelee keonheelee 1060  6월  8 18:10 README.md
-rwxrwxrwx 1 keonheelee keonheelee 1347  6월  8 18:10 setup.sh
drwxrwxr-x 6 keonheelee keonheelee 4096  6월  8 18:10 src
drwxrwxr-x 2 keonheelee keonheelee 4096  6월  8 18:10 test
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ./setup.sh
Made by KeonHeeLee] (Thursday, 7 June 2018)
- Github : https://github.com/KeonHeeLee/study-compiler
- E-mail : beta1360@naver.com

flex -t scanner.l > lex.yy.c
bison -d minic.y
minic.y: warning: 1 shift/reduce conflict [-Wconflicts-sr]
minic.y: warning: 1 reduce/reduce conflict [-Wconflicts-rr]
gcc -o minic lex.yy.c minic.tab.c parser.c icg.c table.c -ll -ly -w
rm -f *.o minic lex.yy.c *.tab.* *.ast *.uco

== Installation is complete. ==
== Usage is as follows. =======
 - Usage(make '.uco' file):: ./minic <miniC file>
 - Usage(execute U-code interpreter):: ./ucodei <uco file>
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ls
image  minic  README.md  setup.sh  src  test  ucodei
```

▲ 그림. 위의 실행방법에 명시해둔 명령어를 실행한 결과

KeonHeeLee@ubuntu:~$ ./minic <Mini-C file(.mc)>
KeonHeeLee@ubuntu:~$ ./ucodei <ICG file(.uco)>

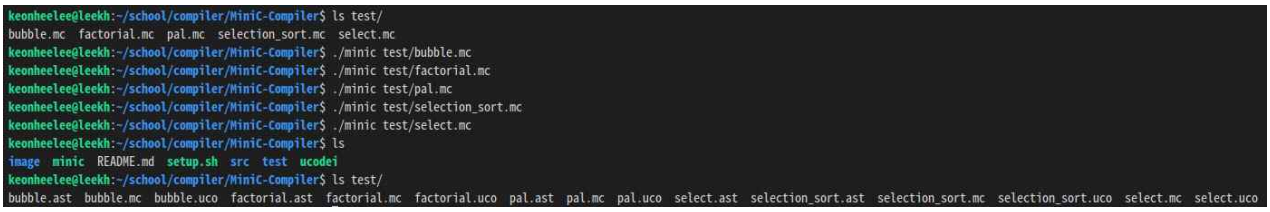
▲ 표. 중간 언어 생성 및 U-code 인터프리터 실행 방법

위와 같이 명령어를 실행하게 되면, 중간 언어 생성까지 minic라는 실행파일로 수행하게 된다. 그 후에 ucodei라는 실행파일을 통해서 ucode 인터프리터 과정을 거치면 된다.

4. 결과

Mini-C 소스 코드(.mc) 5개와 이 소스 코드를 이용해서 실제로 AST 테이블 생성(.ast), 중간 언어 생성(.uco) 및 U-Code 수행 결과(.lst)까지 첨부파일로 올려 두겠다.(총 20개의 파일 - 5개의 테스트 코드x4)

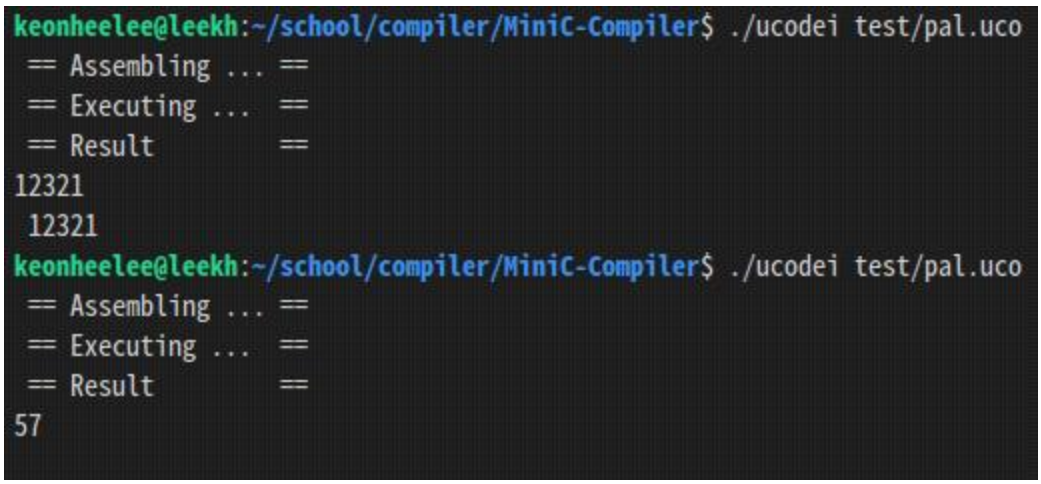
1) minic 실행파일 실행 결과



▲ 그림. 각 mc 파일을 실행한 결과

정상적으로 uco파일과 ast파일이 생성된 것을 확인할 수 있으며, 해당 내용은 과제3(파서 작성)의 내용과 동일하다. 여기서는 중간언어를 기록한 uco파일을 ucodei 실행 파일로 돌려서 정상적으로 결과가 나오면 제대로 되었다는 것을 알 수 있을 것이다.

2) Palindrome (pal.mc)



▲ 그림. pal.uco 실행 결과

pal.mc 파일은 팰린드롬이면 그 수를 그대로 출력하고, 아닐 경우에는 그냥 프로그램이 종료되는 구조를 가지고 있다. 실제로 위와 같이 12321은 팰린드롬 수이므로, 12321을 그대로 출력하고, 57은 좌우 반전하면 같은 수가 아니기 때문에, 아무것도 출력안하고 프로그램이 종료된다.

3) Bubble Sort (bubble.mc)

▲ 그림. bubble.uco 실행 결과

bubble.mc 파일은 0이 입력될 때까지 수를 배열에 계속 저장하고, 0이 입력되면 0을 제외한 모든 수를 Bubble-Sort를 하여 정렬 결과를 출력해주는 프로그램을 나타내는 소스코드다. 위와 같이 5 4 3 2 1 0을 입

력했을 때, 0을 제외한 나머지 수가 순서대로 정렬됨을 알 수 있다.

4) Factorial (factorial.mc)

```
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ./ucodei test/factorial.uco
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
5
 5 120
```

▲ 그림. factorial.uco 실행 결과

factorial.mc는 수를 입력하면 입력한 수의 factorial수를 구해주는 프로그램을 나타내는 소스코드다. 위와 같이 5를 입력하면 5*4*3*2*1 = 120이므로 정상적으로 실행됨을 알 수 있다.

5) Select number (select.mc)

```
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ./ucodei test/select.uco
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
3
 6
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ./ucodei test/select.uco
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
2
 0
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ./ucodei test/select.uco
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
1
 2
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ./ucodei test/select.uco
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
4
 10
```

▲ 그림. select.uco 실행 결과

select.mc는 1을 입력받으면 1+1을, 2를 입력받으면 2-2를, 3을 입력받으면 3*2, 그 외의 수는 10을 출력하게하는 프로그램을 나태내는 소스코드다. 이는 추가기능으로 구현한 switch문을 이용하기 위한 테스트 케이스며, 정상적으로 실행이 됨을 알 수 있다. 또한 select.uco 파일은 아래와 같다.

select.uco

```
main    proc    2    2    2
        sym     2    1    1
        sym     2    2    1
        ldp
        lda     2    1
        call    read
        ldc     0
        str     2    2
        lod     2    1
        ldc     1
$$4     nop     -> case 검사
        lod     2    1
        ldc     3
        eq
        tjp     $$5 -> 3이면 case 3으로
        ujp     $$6 -> 아니면 다음라벨로
$$5     nop     -> case 3: 시작
        lod     2    1
        ldc     2
        mult
```

```
        eq
        tjp     $$1 -> 1이면 case 1으로
        ujp     $$2 -> 아니면 다음라벨로
$$1     nop     -> switch문 진입 및 case 1:
        lod     2       1
        ldc     1
        add
        str     2       2
        ujp     $$0 -> break;
$$2     nop     -> case 검사
        lod     2       1
        ldc     2
        eq
        tjp     $$3 -> 2이면 case 2로
        ujp     $$4 -> 아니면 다음라벨로
$$3     nop     -> case 2: 시작
        lod     2       1
        ldc     2
        sub
        str     2       2
        ujp     $$0 -> break;
        str     2       2
        ujp     $$0 -> break;
$$6     nop     -> default문 진입
        ujp     $$7 -> 다음라벨로
        ujp     $$8
$$7     nop     -> default문 실행
        ldc     10
        str     2       2
        ujp     $$0 -> break;
$$8     nop
$$0     nop
        ldp
        lod     2       2
        call    write
        ret
        end
        bgn     0
        ldp
        call    main
        end
```

▲ 표. select.uco 파일 및 설명(볼드체 + 파란색 주석)

switch문 같은 경우는 라벨을 계속 옮겨가면서 case문에서 검사 후, 아니면 다른 라벨로 jump하는 방법으로 구현이 되었고, 이러한 방법이 정상적으로 구현되었음을 알 수 있다. 또한 break도 $$0라벨로 jump(ujp $$0)를 함으로서, switch문을 탈출하는 구조가 정상적으로 구현된 것을 볼 수 있다.

6) Selection Sort (selection_sort.mc)

```
keonheelee@leekh:~/school/compiler/MiniC-Compiler$ ./ucodei test/selection_sort.uco
 == Assembling ... ==
 == Executing ...  ==
 == Result         ==
 1 2 3 4 5 6 7
```

▲ 그림. selection_sort.uco 실행 결과

selection_sort.mc는 7부터 1까지 배열(내림차순)로 저장되어 있는데, 이를 선택정렬 알고리즘을 이용해서 위와 같이 정렬(오름차순)을 하는 것을 나타내는 소스코드다. 이는 추가기능으로 구현한 for문 관련 소스코드를 보여주기 위한 예제이며, uco파일은 아래와 같다.

selection_sort.uco

```
            sym     1       1       0
main        proc    11      2       2
            sym     2       1       1
            sym     2       2       1
            sym     2       3       1
            sym     2       4       1
            sym     2       5       7
//--- for(k = 0; k < MAX; ++k) ------
            ldc     0               -> for문 시작
            str     2       3       -> k = 0
$$0         nop
            lod     2       3
            ldc     7
            lt                      -> k < MAX
            fjp     $$1             -> 아니면 $$1로
            lod     2       3
            lda     2       5
            add
            ldc     7
            lod     2       3
            sub
            sti
            lod     2       3
            inc
            str     2       3       -> ++k
            ujp     $$0             -> 다시 조건문으로
//-----------end for-------------
//--- for(i = 0; i < MAX - 1; ++i) -----
$$1         nop                     -> for문 시작
            ldc     0
            str     2       1
$$2         nop
            lod     2       1
            ldc     7
            ldc     1
            sub
            lt                      -> i < MAX - 1
            fjp     $$3             -> 아니면 $$3으로
//--- for(j = i + 1; j < MAX; ++j) -----
            lod     2       1       -> for문 시작
            ldc     1
            add
            str     2       2       -> j = i+1
$$4         nop
            lod     2       2
            ldc     7
            lt                      -> j < MAX
            fjp     $$5             -> 아니면 $$5으로
            lod     2       1
            lda     2       5
            add
            ldi
            lod     2       2
            lda     2       5
            add
            ldi
            gt      -> if(array[i] > array[j])
            fjp     $$6             -> if not jump $$6
            lod     2       1
            lda     2       5
            add
```

```
        ldi
        str     2       4
        lod     2       1
        lda     2       5
        add
        lod     2       2
        lda     2       5
        add
        ldi
        sti
        lod     2       2
        lda     2       5
        add
        lod     2       4
        sti
$$6     nop
        lod     2       2
        inc
        str     2       2 -> ++j
        ujp     $$4       -> for문 반복
//-----------end for-------------
$$5     nop
        lod     2       1
        inc
        str     2       1 -> ++i
        ujp     $$2       -> for문 반복
```

```
//-----------end for-------------
//--- for(k = 0; k < MAX; ++k) ------
$$3     nop
        ldc     0         -> for문 시작
        str     2       3 -> k = 0
$$0     nop
        lod     2       3
        ldc     7
        lt                -> k < MAX
        fjp     $$8       -> 아니면 $$8로
        ldp
        lod     2       3
        lda     2       5
        add
        ldi
        call    write
        lod     2       3
        inc
        str     2       3 -> ++k
        ujp     $$7       -> for문 반복
//-----------end for-------------
$$8     nop
        ret
        end
        bgn     0
        ldp
        call    main
        end
```

▲ 표. select.uco 파일 및 설명(볼드체 + 파란색 주석)

여기서 눈여겨 볼 점이라면 for문의 구성인데, for문은 우선 for문 바깥 라벨에서 반복문에 쓰일 변수를 초기화시킨다. 그 후에 for문 라벨에 진입한다. 여기서 if문의 u-code 구조와 비슷하게 True면 for문 내부의 소스코드 실행 후 변수를 증가, 다시 검사하는 라벨로 돌아가게끔 설계했다. 아니면 for문 바깥으로 빠져나가는 구조로 되어있다. 즉, for문의 ucode는 아래와 같이 나오게 되어있다. (예시로 main부터 $$0까지를 들었다.)

```
        ldc     0
        str     2       3
$$0     nop
        lod     2       3
        ldc     7
        lt
        fjp     $$1
        lod     2       3
        inc
        str     2       3
        ujp     $$0
```

▲ 표. for(k = 0; k < MAX; ++k)의 ucode 예시

5. 구현 시 고려 사항 및 discussion

- 규모가 다소 큰 편이므로, 작은 모듈부터 조금씩 구현을 하여야 한다.

우선 책에 있는 소스코드만 받아 적어도 상당히 큰 규모다. 하지만, 이것만으론 전부다 구현하기엔 무리가 있을 수도 있으므로 필자의 경우는 라벨관리, 흐름 제어문 관련해서도 구조체를 추가적으로 구현했다. 이것 관련해서는 1단계 기능 완성 전에는 모듈 개별로 테스트를 진행했으며, 이것이 정상적으로 된다면 다른 소스코드와 이어붙이는 방법으로 구현을 하였다.

그 결과 1단계까지의 구현은 하루 정도로 끝이 났고, 2단계 구현부터 for문, switch와 case, default문, break와 continue 순서로 구현하였다. 그 결과 2단계 소스코드의 경우는 각각 하루정도 조금씩 만들어서 3일 정도로 여유롭게 구현을 완료하였다. 이와 같이 조금씩 그리고 철저하게 구현을 하는 것이 중요하며, 이렇게 함으로써, 나중에 크게 생길 버그라던지 전체적인 구조를 뜯어 고쳐야하는 불상사를 없앨 수 있을 것이다.

- 이번에도 역시 Makefile 때문에 애를 많이 먹었다.

저번 과제 같은 경우는 차라리 parser관련 함수 선언이나 변수들을 각각 util.h와 parser.h에 전부 다 저장하는 방법으로 만들었기 때문에 일시적으론 문제가 없었다. 하지만, 이번에는 parser.h와 icg.h도 연결해야 하기 때문에 Makefile 만드는데 문제가 생겼다. 이에 대해 .o 파일을 생성하는 방법을 생각하고 구현을 하였지만, 결국은 .c파일을 의존성으로 링크해서 Makefile을 작성하는 것이 좀 더 편하게 처리가 되었다.

- 컴파일 과정을 간소화시키기 위해서 한 것 = setup.sh 작성

이번에는 중간 언어 생성하는 프로그램을 컴파일 하는 것과 Ucode 인터프리터 소스코드를 컴파일 하는 과정 2가지를 거쳐야했기 때문에, 이는 오히려 불편하다고 느낄 수도 있을 것이다. 이러한 것을 최소화시키기 위해서 Shell Script를 통해서 make파일을 실행시키고, 현재 작업 디렉토리로 실행파일을 옮긴 후에, 컴파일 도중에 생긴 소스코드는 삭제하는 방법으로 구현을 하였다.

그 결과 번거롭게 컴파일을 하기 위해서 디렉토리를 여러 번 이동해야한다는 불편함과, 여러 gcc와 g++각각 고려하면서 컴파일을 해야한다는 귀찮음이 해소되었다. 단지 컴파일을 하기 위해선 2가지 명령어만 입력하면 끝난다. 사용하기 상당히 편해졌다.

```
KeonHeeLee@ubuntu:~$ sudo chmod +0777 setup.sh
KeonHeeLee@ubuntu:~$ ./setup.sh
```

▲ 표. 디렉토리 이동 및 컴파일까지 해주는 setup.sh 실행

- 테스트 케이스는 다양하게

한 가지 테스트 케이스로만 테스트를 진행하면 다른데서 버그가 터질지 알 수가 없을 것이다. 이러한 경우를 위해서 테스트 케이스를 최대한 다양하게 놓는 것이 중요한데, 이렇게 함으로써 얻는 장점은 아래와 같다.

다양한 버그들로부터 대처가 가능해진다. 테스트 케이스는 우선 프로그램이 잘 돌아가는지를 테스트하기 위한 수단인 것도 사실이다. 하지만 무엇보다 중요한 건 버그가 터질만한 상황들을 만들어서 테스트를 진행하면 "이 버그는 이렇게 고쳐야겠다."라는 식의 대처가 가능해진다. 그리고 다양하게 테스트 케이스를 놓기 때문에 다양한 버그들을 접할 수도 있고, 이들을 전부 대처를 함으로써 프로그램의 안정성은 높아질 것이다.

이러한 이유 때문에 그저 많은 테스트 코드보다 의미 있고 다양한 예제들을 보여줄 수 있는 테스트 코드를 선정, 그리고 그것 위주로 테스트를 진행하였다. (실제로 이 테스트 코드가 위에 제시한 5가지 + 첨부파일)

7. 과제 수행 일지

날짜	과제 수행 내용
2018-06-02	- 중간 언어 생성관련 개념 공부
2018-06-03	- 중간 언어 생성관련 개념 공부 - 1단계 구현 완료
2018-06-04	- 1단계 구현 테스트 - 2단계 구현 (for문 구현)
2018-06-05	- for문 테스트 (selection_sort.mc로 테스트) - 2단계 추가 구현 (switch와 case 구현)
2018-06-06	- switch와 case 테스트 (select.mc 일부 코드로 테스트 진행) - 2단계 추가 구현 (break와 continue 구현)
2018-06-07	- break와 continue 테스트 - 전체 테스트 및 shell 파일 작성, Github 관련 디렉토리 구조 정리
2018-06-08	- 보고서 작성

부록. 참고자료 및 첨부파일 목록

① 참고자료

[1] 오세만 ,「컴파일러 입문」, 정익사 p451~491 - 중간 코드 생성
[2] 오세만 ,「컴파일러 입문」, 정익사 p359~365 - 구문 분석기의 작성
[3] 오세만 ,「컴파일러 입문」, 정익사 p526~542 - 심벌 테이블
[4] 구글 검색, "Makefile를 작성할 때 알면 좋은 것들",
https://wiki.kldp.org/KoreanDoc/html/GNU-Make/GNU-Make-4.html
[5] 오세만 ,「컴파일러 입문」, 정익사 p612~641 - Mini C

② 첨부파일

=> 20130870_이건희(과제4).zip

- 20130870_이건희(과제4).hwp -> 현재 보고서
- 20130870_이건희(과제4).pdf -> 현재 보고서
- main -> 중간 언어 소스 코드
- image -> 스냅샷 이미지파일
- test -> .mc .ast .uco .lst 파일 (pal, bubble, select, selection, sort, factorial)
- MiniC-Compiler-master.zip -> Github Repository 다운로드 한 파일