Programming Project #3

- Bottom Parsing을 이용한 Mini-C 인터프리터 개발 -

- 소프트웨어학과 201520882학번 천윤서

1. 서론

- 과제 소개

본 과제는 Minic-C라는 특수하게 정의된 언어의 간단한 인터프리터를 개발하는 과제이다. 어휘 분석을 통해 토큰을 받아서 bison에서 문법 분석을 하며 트리를 생성하고, 최종적으로 트리를 통해서 프로그램을 실행한다.

- 구현된 부분/구현되지 않은 부분

구분	항목	구현 여부
필수 요소	변수 및 상수	구현
	수식과 연산자	구현
	수식 문장	구현
	조건문과 출력문	구현
	block 문	구현
필수 조건	Error 메세지 출력	구현
	Error Recovery	구현
선택	함수(선택)	미구현
	재귀 함수(선택)	미구현

필수 요구 사항은 전부 구현하였으나, 선택 사항인 함수 관련 부분은 구현하지 않았다. 정수와 실수의 혼합식은 실수로 변경되어 계산되어진다.

2. 문제 분석

- grammar rule 작성.

파트	문법	지원 연산자
문장 처리	stmts -> stmts stmt | ε	
	stmt -> assign ; | block | whileStmt | ifStmt | printStmt | ; | error ;	
	whileStmt -> while (assign) stmt	
	ifStmt -> if (assign) stmt else stmt	
	printStmt -> print assign ;	
	block -> { stmts }	
수식 연산자 (할당)	assign -> id = assign | equrel	
수식 연산자 (비교)	equrel -> equrel < cmprel | cmprel	<, <=, >, >=
	cmprel -> cmprel == expr | expr	!=, ==
수식 연산자 (가감승제)	expr -> expr + term | term	+, -
	term -> term * factor | factor	*, /
factor	factor -> (assign) | double | integer | id	

stmts는 stmt로 쪼개지며, stmt는 assign; block, while, if, print 등이 될 수도 있다. error는 yyac의 error recovery를 이용하기 위해서 넣은 것인데, 에러가 발견되고 ';'를 만날 때 까지의 문장을 없는 것처럼 취급하여 끝까지 컴파일 하게 돕는다.

수식 연산자는 2차 과제와 거의 비슷한 문법을 사용하지만 비교 연산자들이 추가되었다. 비교 연산자의 우선순위를 주기 위해서 equrel(equal relop) cmprel(compare relop) 심볼을 추가하였다. 2차 과제와 달리 unary는 조건에 따로 명시되지 않아서 제거하였다.

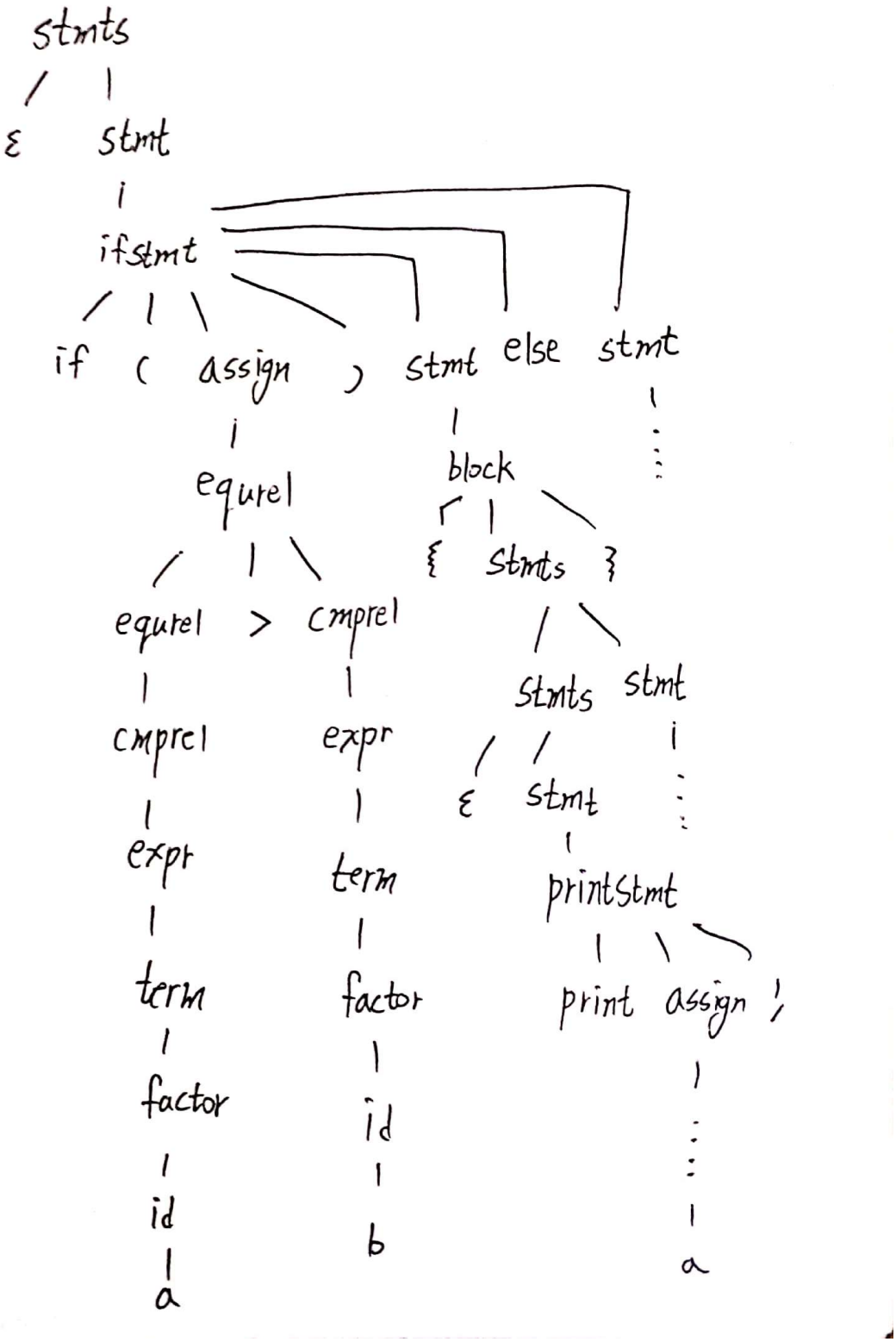
실제 bision에 입력할 때는 지원 연산자들을 전부 대입해서 | 로 묶고 작성해서 넣어야 한다. 예를 들어, expr 관련 식은 expr -> expr + term | expr – term | term 이 될 것이다.

간단한 식으로 parse tree의 예제를 들자면 아래와 같다.

```
if(a>b) {
        print a;
```

a = a +1;

} else print b;



- grammar rule의 LALR(1) parsing 여부에 관한 분석.

```
Grammar

    0 $accept: start $end

    1 start: stmts

    2 stmts: stmts stmt
    3      | %empty

    4 stmt: assign ';'
    5     | block
    6     | whileStmt
    7     | ifStmt
    8     | printStmt
    9     | error ';'

   10 whileStmt: WHILE '(' assign ')' stmt

   11 ifStmt: IF '(' assign ')' stmt ELSE stmt
```

yacc을 통해 본 결과, 어떤 conflict도 발견되지 않았고, 실제 예제를 집어넣었을 때에도 정상적으로 동작하였다.

3. 설계

- 주요 자료구조.

주요 자료구조들은 struct.h에 정의되어 있다.

1) symbolTable

변수	특징
char id[17]	아이디의 이름을 정의한다.
uval val	계산에 필요한 double과 int를 담는 union 자료구조이다.
int isInteger	해당 노드가 가진 값이 double인지 integer인지 정의한다.

2) syntaxTree

문법이 분석되면서 만들어지는 syntax tree의 자료구조이다.

변수	특징
char symbol	해당 노드가 어떤 노드인지 알려준다.(while, +, leaf 등)
syntaxTree **child	자식 노드들의 배열이다. 동적으로 크기가 변한다.
int childNum	자식이 몇 개인지 알려준다. 자식을 순환하는데 사용한다.
int childTableSize	동적으로 테이블 크기를 증가시키는데 사용되는 변수.
uval val	계산에 필요한 값을 저장한다. union이다.
int isInteger	해당 노드가 가진 값이 double인지 integer인지 정의한다.

함수	특징
syntaxTree *makeNode(char symbol)	심볼에 해당하는 새로운 노드를 만들고 초기화한다. 기본적으로 두 자식을 초기화한다.
syntaxTree *makeLeaf(char *type, uval val)	리프 노드를 만들고, 적절한 값을 저장한다.
void addChild(syntaxTree *parent, syntaxTree* child)	자식을 추가한다. 자식의 포인터 배열이 모자라면 2배로 증가시킨다.
void freeTree(syntaxTree *node)	루트 노드를 받아서 0부터 childNum까지의 자식을 순회하며 할당된 메모리를 해제한다.

- 프로그램의 모듈 및 구조.

1) source.l

1차 과제에서 제출했던 모듈을 재사용하였기에, 불필요한 부분에 대해 복잡하게 언급하지 않겠다.

토큰 종류	동작
한글자(+, -, *, ;, ...)	yytext[0]를 리턴한다.
ID	심볼 테이블에 등록한 뒤, ID 토큰을 리턴한다.
INTEGER, DOUBLE	각각 최대자리수로 자르는 등 처리를 한 뒤 yylval을 이용해서 값을 전달하고, 토큰을 리턴한다.
공백, 주석	기본적으로 무시한다. Newline의 경우는 몇 번 나왔는지 세서 에러의 위치를 알리는데 사용.
미정의	isLexicalError를 1로 설정하고, 에러문을 출력한다.

변수 및 함수	동작
void initTable()	심볼 테이블을 초기화한다.
int symbolIndex	심볼 테이블의 등록 마지막 위치를 가리킨다.
int lineCounter	현재 몇 번째 라인을 처리하고 있는지 알려준다.
int isLexicalError	Lexical Error 여부를 알려준다.

2) source.y

이번 과제의 핵심 부분이다. 주요 사용 변수들은 아래와 같다.

변수	특징
int isSyntaxError, syntaxErrorLine	문법 오류와 관련된 부분.
symbolTable *st	심볼 테이블이다. id 를 등록할 때 동적으로 크기가 변한다.
syntaxTree *prgm	syntax Tree 의 root 역할을 한다. 심볼 s 로 초기화된다.(stmts)

- syntax Tree 생성

SyntaxTree가 가지는 노드의 종류는 아래와 같다. 트리 구조에서 문장 리스트의 자식은 무한정 늘어날 수 있다. while문과 연산자 트리들은 자식을 둘을 가지며, 수식 문장과 출력문은 하나의 자식, if문은 3개의 자식을 지닌다.

심볼	부모 노드	행동
a	수식 문장	첫 번째 자식으로 연산자를 지니고 있다.
s	문장 리스트	n개의 문장에 해당하는 자식들을 가진다.
+,-,*,/	가감승제 연산자	자신이 연산할 두 트리 자식을 지닌다.
g(>), G(>=) l(<), L(<=) e(==), n(!=)	비교 연산자	자신이 연산할 두 트리 자식을 지닌다.
=	대입 연산자	ID를 왼쪽 자식으로 지니며, 오른쪽 자식은 트리이다.
w	조건분기문 (while)	왼쪽에 조건문을 나타내는 자식을, 오른쪽에는 실제 반복 순회할 자식을 지닌다.
i	조건분기문 (if)	첫 번째 자식은 조건문을 나타내며, 두번째와 세번째는 각각 참 거짓 분기의 순회할 자식이다.
p	출력문 (print)	하나의 출력할 자식 노드를 지니고 있다.
#	상수 노드	리프 노드이다. 생성과 동시에 숫자값을 가지고 있다.
@	아이디 노드	리프 노드이다. 생성과 동시에 id index를 가진다.

아래는 문법을 파싱 할 때 syntax tree 가 생성되는 과정이다. 실제 코드의 논리를 기술하였다. 표가 지나치게 길어지는 것을 방지하기 위해서 여러 연산자에 대해 동일한 논리를 가지는 식은 대표 연산자를 하나 지정하여 설명하였다.

문법	동작
start -> stmts	루트 노드인 prgrm이 stmts의 트리를 포인팅한다.
stmts -> stmts stmt	stmts의 문장 리스트 노드에 stmt의 노드를 자식으로

	추가한다.
stmts -> ε	문장 리스트 노드를 생성하여 stmts에게 넘긴다.
stmt -> assign ';'	수식 문장 노드를 생성하여 assign이 생성한 트리를 자식으로 하고 stmt에 넘긴다.
stmt -> ';'	더미 문장 리스트 노드를 생성하고 stmt에게 넘긴다.
stmt -> error ';'	더미 문장 리스트 노드를 생성하고 stmt에게 넘긴다.
whileStmt -> WHILE '(' assign ')' stmt	while 노드를 생성하여 whileStmt에게 넘긴다. 첫번째 자식은 assign의 트리, 두 번째 자식은 stmt의 트리가 된다.
ifStmt -> IF '(' assign ')' stmt ELSE stmt	if 노드를 생성한다. 첫 번째 자식은 assign의 트리, 두 번째와 세 번째 자식은 각각 $5, $7의 stmt가 가지는 트리가 된다.
printStmt -> PRINT assign ';'	assign의 트리를 자식으로 가지는 print 노드를 생성해서 printStmt에게 넘겨준다.
block -> '{' stmts '}'	stmts가 생성한 트리를 block에게 넘겨준다.
assign -> ID '=' assign	ID를 왼쪽 자식, 오른쪽의 assign을 오른쪽 자식으로 지니는 = 노드를 생성하여 $$에게 넘겨준다.
assign -> equrel **equrel -> cmprel** **cmprel ->expr** **term -> factor**	$1의 트리를 $$에게 넘겨준다.
equrel -> equrel GE cmprel **cmprel -> cmprel EQ expr** **expr -> expr '+' term** **term -> term '*' factor**	$1과 $3를 왼쪽과 오른쪽 자식으로 하는 $2의 노드를 생성해서 $$에게 넘겨준다.
factor -> '(' assign ')'	assign의 트리를 factor에게 넘겨준다.
factor -> INTEGER	$1로부터 값을 받아서 INTEGER노드를 만들고 이를 factor에게 넘긴다.

이를 표만 보고 동작을 파악하기는 쉽지 않는데, 실제 간단한 예를 통해 생성되는 트리를 보면 아래와 같다.

a = 3;

b = 1 + 3;

if(a > b) {

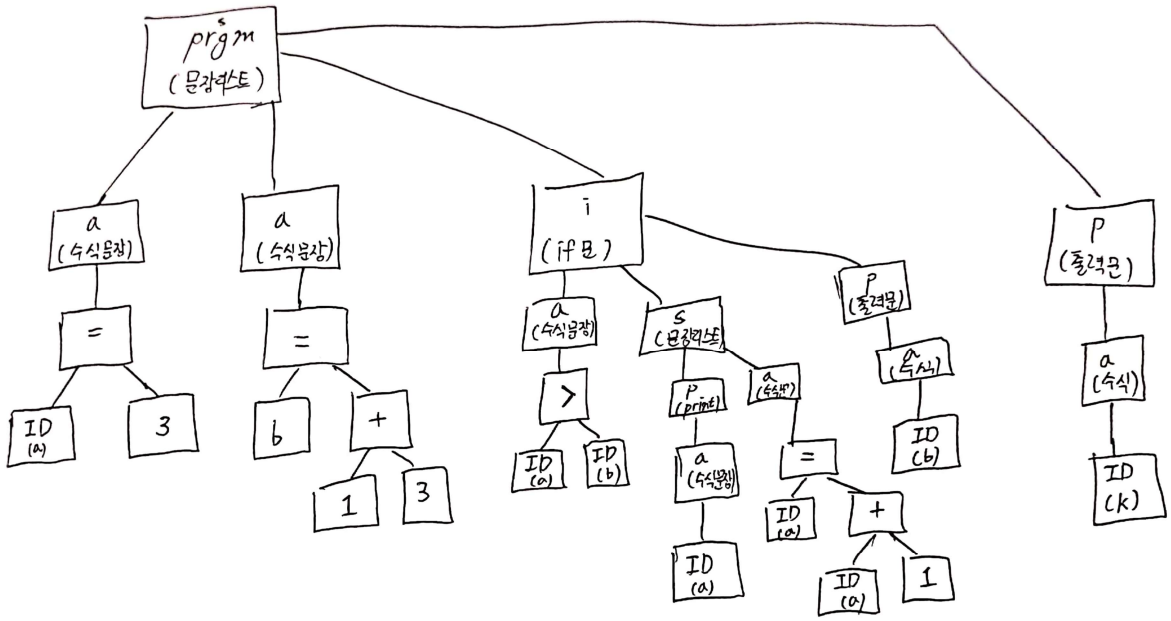
print a;

a = a + 1;

}

else print b;

print k;



- SyntaxTree 실행.

참고로 Error Recovery 가 동작하여 컴파일은 끝까지 이루어지면서 모든 문법 오류를 찾고 SyntaxTree 가 Error 가 발생한 부분은 더미 노드로 채워지면서 생성은 되지만 만들어진 SyntaxTree 를 실행하는 과정은 isSyntaxError 나 isLexicalError 가 1 로 설정되어있으면 하지 않는다. 이 더미 노드가 채워진 트리를 실행해봤는데, 앞 줄에 있는 에러로 인해 무시된 줄이 뒤로 갈수록 흐름을 크게 바꿔서 무의미한 결과가 출력됐다. 따라서 일반적인 컴파일러처럼 컴파일만 하고, 실행은 하지 않는 게 더 합당하다고 생각했다.

SyntaxTree 를 순회하면서 실제 실행하는 과정에 도와주는 함수는 아래와 같다. 해당 함수들은 중심 함수인 recursiveTreeProcessing(syntaxTree *node)라는 함수 안에서 자주 쓰이는 기능들이다.

함수	동작
uval getValueOfId(int index, int *isInteger)	index에 해당하는 symboltable의 id가 가진 값과, 자료형 여부를 isInteger로 알려준다.
copyNodeForCalculate (syntaxTree *dest, syntaxTree *source)	노드가 가지는 계산에 사용하는 값을 복사한다. while의 경우 노드를 여러 번 순회하는데, 원본 노드의 값을 변경하면 결과가 다르게 나올 수 있기 때문이다.
void processChildNodeForCalculate (syntaxTree *calNode1, syntaxTree calNode2)	계산을 하기 위해서 입력된 node가 id인 경우 symbol table에서 값을 얻어와서 상수 노드로 바꾸고, 어느 한쪽이 isInteger가 0으로 실수 노드라면, 다른 한쪽도 실수 노드로 바꿔서 실수끼리의 연산으로 변환한다.
void processExpression(char symbol, syntaxTree *node)	수식 연산자들을 처리하는 루틴이다. 수식 연산을 처리하는 과정이 거의 동일하여 원본함수에서 분리했다.

실제 recursiveTreeProcessing함수는 루트 노드부터 시작해서 재귀적으로 child 노드를 탐색하면서 symbol에 따라 서로 다른 동작을 수행하고, 부모에게 필요한 값을 전달한다.

수행하는 동작들은 아래 표와 같다.

심볼	부모 노드	행동
a	수식 문장	자식 노드를 재귀적으로 순회한 뒤, 자식 노드의 값을 얻어온다.
s	문장 리스트	왼쪽 자식부터 차례로 재귀적으로 순회한다.
+,-,*,/	가감승제 연산자	양쪽 자식 노드를 재귀적으로 처리한 뒤, 값을 부모 노드에 해당하는 연산자로 계산하여 부모 노드로 올린다.
g(>), G(>=) l(<), L(<=) e(==), n(!=)	비교 연산자	양쪽 자식을 재귀적으로 처리한 뒤, 그 값을 비교해서 부모 노드의 값으로 참이면 1을, 거짓이면 0을 올린다.
=	대입 연산자	양쪽 자식 노드를 순회한 뒤, 왼쪽 자식 노드에 해당 ID의 테이블에 오른쪽 자식의 값을 대입하고, 오른쪽 자식의 값을 부모 노드에 올린다.
w	조건분기문 (while)	왼쪽 자식을 한 번 순회한다. 두 번째로, 오른쪽 자식 노드를 순회하고, 다시 왼쪽 자식 노드를 순회한다. 두 번째 동작을 자식 노드가 1(참)인 동안 계속 반복한다.
i	조건분기문 (if)	첫 번째 자식 노드를 순회한다. 0이 아니라면 왼쪽 자식을 순회하고, 그렇지 않으면 오른쪽 자식을 순회한다.
p	출력문 (print)	자식 노드를 순회한 뒤 그 값을 받아서 출력한다.
#	상수 노드	리프 노드이다. syntaxTree 생성 시점에서 이미 처리 되어 있다.
@	아이디 노드	

4. 수행 결과

- 첫 번째 수행 결과.

<테스트 코드>

```
a = 10;
b = 20;
if(a>b)
    max = a;
else
    max = b;
print max;


sum = 0;
sum = sum + i;
print 10;
print a*b;
print a=b;

d = 103;

count = 0;
while(count < 5)
{
    count = count + 1;
    if(count > 3) print count;
    else print d;
}
```

<실행 결과>: 값이 정상적으로 출력되었다.

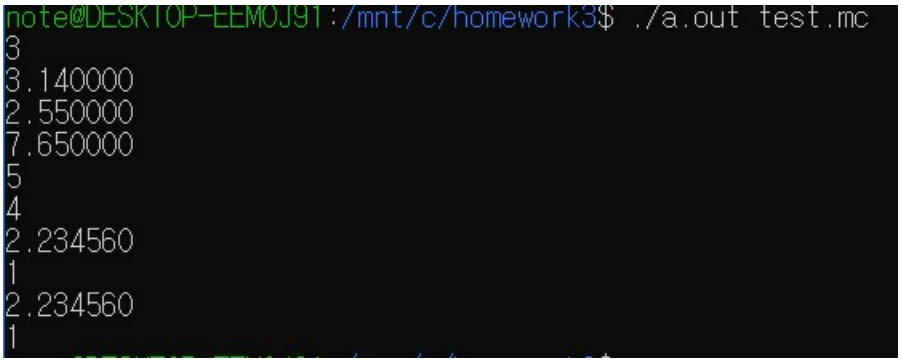
```
chun@ubuntu:~/workspace/compiler/homework3$ ./a.out sample.mc
20
10
200
20
103
103
103
4
5
chun@ubuntu:~/workspace/compiler/homework3$
```

- 두 번째 수행 결과.

<테스트 코드>

```
a = 3;
print a;
b = 0.14 + a;
print b;
c = 5.1/2;
print c;
d = a*c;
print d;

count = 5;
tval = 0;

while(count > 0) {
        if(count >3) print count;
        else if(count == 2) {
                if(tval == 0) {
                        tval = tval + 1;
                        print tval;
                }
                        else ;
} else print 1.23456+1;
        count = count - 1;
}


min = 0;
atmp = 1;
btmp = 2;
ctmp =3;

if(atmp <= btmp) {
        if(atmp >= ctmp) min = ctmp;
        else min = atmp;
} else {
        if(btmp >= ctmp) min = ctmp;
        else min = btmp;
}

print min;
```

<실행 결과>: 모든 값이 정상으로 출력되었다.



- syntax error 가 있는 경우.

<테스트 코드>

```
1 @
2 #
3 b = 2;
4 whil ( b == 3 ) b = b + 1;
5
6
7 a ===== b + 1;
8
9
10 if( a > b ) {
11
12     (( c = 3);
13      b = b + 1;
14
15 }
16 esle {
17
```

<실행 결과>

```
chun@ubuntu:~/workspace/compiler/homework3$ ./a.out eror.mc
[Lexical Error]: line 1
[Lexical Error]: line 2
[Syntax Error]: line 4
[Syntax Error]: line 7
[Syntax Error]: line 12
[Syntax Error]: line 16
```

lexical Error와 Syntax Error에 대해서 Error Recovery가 동작되기에, Lexical Error는 에러가 난 토큰을 없는 것처럼 무시하고, Syntax Error는 에러가 난 line(line seperator ';')을 지워서 없는 것처럼 취급하여 중단하지 않고 끝까지 컴파일을 수행한다.

1, 2번은 지원하지 않는 문자에 대해서 lexical error가 났고, 4라인은 while을 whil로 적었고, 7라인은 a "==" "==" "=" b + 1로 인식이 되어서 에러가 났을 것이다. 12라인은 닫힌 괄호가 오지 않아서 에러가 났고, 16번째 라인은 else를 esle로 잘못 쓰면서 에러가 났다.