

# Parser 구현 과제



과 목 : 컴파일러 02

학 과 : 컴퓨터공학

학 번 : 2020112011

이 름 : 정찬형

제 출 일 : 2025.11.03.

## 1. 개요

본 프로젝트는 MiniC 언어를 위한 LR 파서를 구현하는 것을 목표로 합니다.  
Scanner 와 Parser 를 문법 파일(MiniC.gr)과 파싱 테이블(MiniC.tbl)을 이용하여  
통합하고, 파싱 결과로 Right Parse 순서를 출력하도록 구현하였습니다.

### 프로젝트 구조

MiniC Scanner/

```
|— Scanner.h      # 토큰 정의 및 스캐너 인터페이스  
|— Scanner.c      # 어휘 분석기 구현
```

MiniC Parser/

```
|— Parser.h      # 파서 인터페이스  
|— Parser.c      # LR 파서 구현  
|— parserTest.c  # 파서 테스트 메인 프로그램
```

MiniC Grammar/

```
|— MiniC.gr      # MiniC 문법 정의  
|— MiniC.tbl      # 파싱 테이블  
|— MiniC.lst      # 심볼 리스트
```

Examples/

```
|— factorial.mc   # 테스트 프로그램들  
|— bubble.mc  
|— perfect.mc
```

## 2. 수정 전 코드 분석

### 2.1 Parser.c 원본 분석

#### 주요 함수 구조

##### 1) parser() 함수

```
void parser()  
{  
    extern int parsingTable[NO_STATES][NO_SYMBOLS + 1];  
    extern int leftSymbol[NO_RULES + 1], rightLength[NO_RULES + 1];  
    int entry, ruleNumber, lhs;  
    int currentState;  
    struct tokenType token;
```

```

sp = 0; stateStack[sp] = 0; // initial state
token = scanner(source);

while (1) {
    currentState = stateStack[sp];
    entry = parsingTable[currentState][token.number];

    if (entry > 0)          /* shift action */
    else if (entry < 0)      /* reduce action */
    else                      /* error action */
}
}

```

**기능 분석:** - LR 파싱 알고리즘의 핵심 구현 - 파싱 테이블을 참조하여 shift/reduce/accept 액션 수행 - 스택 기반 상태 관리 - 에러 발견 시 복구 메커니즘 호출

**문제점:** - Right Parse 결과를 기록하거나 출력하는 기능이 없음 - reduce 액션 발생 시 어떤 규칙이 적용되었는지만 출력하고 저장하지 않음

## 2) semantic() 함수

```

void semantic(int n)
{
    printf("reduced rule number = %d\n", n);
}

```

**기능 분석:** - 각 reduce 액션에서 호출되어 규칙 번호 출력 - 실제 의미 분석은 구현되지 않음

## 3) errorRecovery() 함수

```

void errorRecovery()
{
    struct tokenType tok;
    int parenthesisCount, braceCount;
    int i;

    // step 1: skip to the semicolon
}

```

```

parenthesisCount = braceCount = 0;
while (1) {
    tok = scanner(source);
    if (tok.number == teof) exit(1);
    if (tok.number == tsemicolon && parenthesisCount == 0 && braceCount == 0)
        break;
    // ... 괄호 카운팅 로직
}

// step 2: adjust state stack
for (i = sp; i >= 0; i--) {
    if (stateStack[i] == 36) break; // statement_list
    if (stateStack[i] == 24) break; // statement_list
    // ... 다른 상태들 확인
}
sp = i;
}

```

**기능 분석:** - 구문 에러 발생 시 복구 전략 수행 - 세미콜론까지 토큰 건너뛰기 - 파싱 스택을 적절한 상태로 조정

## 2.2 Scanner.h 원본 분석

```
#define NO_KEYWORD 7 // 수정 전: 7개 키워드만 정의
```

```

enum tsymbol {
    tnull = -1,
    tnot, tnotequ, tremainder, tremAssign, tident, tnumber,
    tand, tlparen, trparen, tmul, tmulAssign, tplus,
    tinc, taddAssign, tcomma, tminus, tdec, tsubAssign,
    tdiv, tdivAssign, tsemicolon, tless, tlesse,      // tcolon 없음
    tassign, tequal, tgreat, tgreat, tlbracket, trbracket,
    teof,
    // word symbols (순서가 파싱 테이블과 불일치)
    tconst, telse, tif, tint, treturn, tvoid, twhile, // 7개만
    tlbrace, tor, trbrace
};

```

**문제점:** 1. NO\_KEYWORD 가 7로 설정되어 있어 7개 키워드만 처리 2. enum 순서가  
파싱 테이블의 terminal symbol 순서와 불일치 3. tcolon 토큰이 정의되지 않음 4.  
break, case, continue, default, do, for, switch 키워드 누락

### 2.3 Scanner.c 원본 분석

```
enum tsymbol tnum[NO_KEYWORD] = {  
    tconst, telse, tif, tint, treturn, tvoid, twhile  
};  
  
char* keyword[NO_KEYWORD] = {  
    "const", "else", "if", "int", "return", "void", "while"  
};
```

**문제점:** 1. 7개 키워드만 배열에 포함 2. 알파벳 순서가 아닌 임의 순서로 정렬 3.  
tokenName 배열에 colon(‘:’) 토큰 이름 없음

---

## 3. 문제 발견 및 진단

### 3.1 토큰 번호 불일치 문제

**증상:**

Current Token: void

==== error in source ===

factorial.mc 의 void main() 구문에서 void 토큰을 인식하지 못하는 오류 발생.

**원인 분석:**

#### 1. MiniC.lst 파일의 CROSS REFERENCE 섹션 분석

MiniC.lst 파일의 1100~1160 라인에는 모든 문법 심볼들의 CROSS REFERENCE 정보가 있습니다:

```
*****  
*      CROSS      REFERENCE      *  
*****
```

' !='	82
' %'	94
' %='	75
'%ident'	110 16 34 33
'%number'	111 35 32
' &&'	79
' ('	60 62 112 102 59 58 17 63 61
' )'	60 62 112 102 59 58 17 63 61
' *'	92
' *='	73
' +'	89
' ++'	98 103
' +='	71
' ,'	30 21 109
' -'	96 90
' --'	99 104
' -='	72
' /'	93
' /='	74
' :'	55 54        <- colon: 규칙 55, 54에서 사용
' ;'	67 57 56 63 63 51 61 28
' <'	85
' <='	87
' ='	32 70
' =='	81
' >'	84
' >='	86
' [ '	101 34
' ] '	101 34
' _   _'	0
' break'	57        <- 키워드 시작
' case'	54
' const'	13
' continue'	56
' default'	55
' do'	61
' else'	59
' for'	63

'if'	59	58
'int'	14	
'return'	67	
'switch'	62	
'void'	15	<- void: 규칙 15에서 사용 (type_specifier)
'while'	60	61
'{'	23	
'  '	77	
'}'	23	

**주요 사항:** - Terminal symbols 는 순서대로 나열되어 있으며, 각각에 고유한 인덱스가 할당됨 - 심볼 목록을 0 번부터 세면: - 0: '!' - 1: '!= ' - ... - 20: ':' (colon) - ... - 30: 'eof' (실제로는 '\_|\_' 이후) - 31: 'break' - ... - 43: 'void'

## 2. 파싱 테이블에서의 심볼 번호 사용

MiniC.tbl 파일 분석:

```
#define NO_SYMBOLS 101      // 총 101 개 심볼 (terminal + nonterminal)

int parsingTable[NO_STATES][NO_SYMBOLS+1] = {
    {/** state 0 **/}
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 14, 0, 0, 0, 0, 0,
     13, 0, 0, 12, 0, 0, 0, 0, ...
     // 인덱스: 0   1   2   3   ... 40  41  42  43
     // 심볼:   '!'  '!= ' '%' '%=' ... 'int' ... ... 'void'
    },
    ...
};
```

**파싱 테이블 구조:** - 2 차원 배열: parsingTable[state][symbol] - state: 현재 파싱 상태 (0~192) - symbol: 토큰/심볼 번호 (0~100) - 값: shift/reduce/error 액션

**예시:** state 0에서 void 토큰 처리

```
parsingTable[0][43] = 12 // 43 번 위치(void)에 shift 12 액션
```

### 3. Scanner.h 의 토큰 번호 (수정 전)

```
enum tsymbol {
    tnot = 0,          // '!' - 맞음
    tnotequ,           // '!=` - 1 번 맞음
    ...
    tdiv,              // '/' - 18 번
    tdivAssign,         // '/=' - 19 번
    tsemicolon,         // ';' - 20 번 (잘못됨! 실제로는 ':' 위치)
    tless,              // '<' - 21 번 (잘못됨!)
    ...
    teof,               // 순서상 30 번이 아닌 위치
    // 키워드들
    tconst,             // 첫 번째 키워드 (31 번이어야 하는데...)
    telse,
    tif,
    tint,
    treturn,
    tvoid,              // 35 번 (실제 필요: 43 번)
    twhile,
    ...
};
```

**문제점:** - tcolon이 빠져서 20 번 위치에 tsemicolon이 잘못 배치됨 - 이후 모든 토큰이 한 칸씩 밀림 - 키워드가 7 개만 정의되어 전체 순서가 틀어짐 - tviod가 35 번에 위치 (실제 필요: 43 번)

### 4. 파싱 테이블 참조 메커니즘과 오류 발생

Parser.c 의 파싱 루프:

```
while (1) {
    currentState = stateStack[sp];
    entry = parsingTable[currentState][token.number];
    //                                     ^^^^^^^^^^
    //                               Scanner에서 반환된 토큰 번호

    if (entry > 0)      /* shift */
    else if (entry < 0) /* reduce */
```

```
    else          /* error - 잘못된 entry 접근 시 0 반환 */
}
```

### 오류 발생 시나리오:

```
factorial.mc: void main(void) { ... }
               ^^^^
```

Step 1: Scanner 가 'void' 토큰 스캔

- keyword 배열에서 "void" 문자열 검색
- 찾은 인덱스: 5 (7 개 키워드 중 6 번째, 0-based)
- tnum[5] = tvoid 반환
- Scanner.h 에서 tvoid = 35

Step 2: token.number = 35

Step 3: Parser 가 파싱 테이블 조회

- currentState = 0 (초기 상태)
- entry = parsingTable[0][35]
- BUT! 파싱 테이블은 void 를 43 번에 저장
- parsingTable[0][35]는 다른 심볼의 액션 또는 0 (error)

Step 4: 잘못된 액션 수행

- entry == 0 (error)
- "==== error in source ===" 출력
- void 토큰을 인식하지 못함

### 토큰 배치 비교:

파싱 테이블이 기대하는 레이아웃 (MiniC.lst CROSS REFERENCE 순서):

Index:	0	1	2	...	20	21	...	30	31	32	33	...	43	44
Token:	!	!=	%	...	:	;	...	eof	break	case	const	...	void	while

Scanner 가 제공하는 레이아웃 (수정 전):

Index:	0	1	2	...	18	19	20	...	30-ish	31-ish	...	35	36
Token:	!	!=	%	...	/	/=	;	...	eof	const	...	void	while

^

colon(:)이 빠져 이후 모두 -1 씩 밀림

## 인덱스 불일치 상세 분석:

실제 심볼	파싱테이블 위치	Scanner 위치 (수정 전)	차이	비고
:	20	없음	N/A	누락!
;	21	20	-1	colon 누락의 영향
(semicolon)				
eof	30	~29	-1	
break	31	없음	N/A	키워드 누락
case	32	없음	N/A	키워드 누락
const	33	31	-2	
continue	34	없음	N/A	키워드 누락
default	35	없음	N/A	키워드 누락
do	36	없음	N/A	키워드 누락
else	37	32	-5	
for	38	없음	N/A	키워드 누락
if	39	33	-6	
int	40	34	-6	
return	41	35	-6	
switch	42	없음	N/A	키워드 누락
void	43	36	-7	에러 발생 지점
while	44	37	-7	

**결론:** - Scanner 와 파싱 테이블 간의 토큰 번호 정확한 일대일 대응 필수 - : 토큰 1개 누락 + 키워드 7 개 누락 = 총 8 개 심볼 문제 - 누락 효과가 누적되어 void 는 -7 차이 발생 - 파싱 테이블의 symbol 배치는 PGS 가 MiniC.lst 순서로 자동 생성 - Scanner.h 의 enum 은 이 순서를 반드시 따라야 함

---

## 4. 수정 내역

### 4.1 Parser.c 수정

수정 1: *stdlib.h* 헤더 추가

// 수정 전

```
#include "../MiniC Grammar/MiniC.tbl"
#include "Parser.h"
```

// 수정 후

```
#include <stdlib.h>
#include "../MiniC Grammar/MiniC.tbl"
#include "Parser.h"
```

이유: `exit()` 함수 사용을 위해 필요

수정 2: *Right Parse* 추적 변수 추가

// 수정 전

```
int sp;
int stateStack[PS_SIZE];
int symbolStack[PS_SIZE];
```

// 수정 후

```
int sp;
int stateStack[PS_SIZE];
int symbolStack[PS_SIZE];
```

```
int rp; // right parse index 추가
```

```
int rightParse[RP_SIZE]; // right parse array 추가
```

이유: `reduce` 된 규칙 번호를 순서대로 저장하기 위한 배열

수정 3: *parser()* 함수 - 초기화 및 기록

// 수정 전

```
void parser()
```

```
{
```

```
...
```

```
sp = 0; stateStack[sp] = 0;
```

```
token = scanner(source);
```

```

while (1) {
    ...
    else if (entry < 0) {
        ruleNumber = -entry;
        if (ruleNumber == GOAL_RULE) {
            if (errcnt == 0)
                printf("\n *** valid source ***\n");
            return;
        }
        semantic(ruleNumber); // 규칙 번호 출력만 함
        ...
    }
}

// 수정 후
void parser()
{
    ...
    sp = 0; stateStack[sp] = 0;
    rp = 0; // right parse index 초기화 추가
    token = scanner(source);
    while (1) {
        ...
        else if (entry < 0) {
            ruleNumber = -entry;
            if (ruleNumber == GOAL_RULE) {
                if (errcnt == 0) {
                    printf("\n *** valid source ***\n");
                    printRightParse(); // right parse 출력 추가
                }
                return;
            }
            // 규칙 번호 저장 추가
            if (rp < RP_SIZE) {
                rightParse[rp++] = ruleNumber;
            }
            semantic(ruleNumber);
        }
    }
}

```

```

    ...
}

}

}

```

**변경 사항:** 1. rp 인덱스를 0으로 초기화 2. reduce 액션마다 규칙 번호를 rightParse 배열에 저장 3. 파싱 성공 시 printRightParse() 호출

**수정 4: printRightParse() 함수 추가**

```

void printRightParse()
{
    int i;
    printf("\n *** Right Parse ***\n");
    for (i = rp - 1; i >= 0; i--) {
        printf("%3d", rightParse[i]);
        if ((rp - i) % 20 == 0) printf("\n"); // 20 개마다 줄바꿈
    }
    printf("\n");
}

```

**기능:** - rightParse 배열에 저장된 규칙 번호를 역순으로 출력 - Rightmost derivation 순서 표시 - 가독성을 위해 20 개마다 줄바꿈

**Right Parse의 의미:** - Bottom-up 파싱에서 reduce 된 규칙들을 기록 - 역순으로 출력하면 Top-down의 Rightmost derivation과 동일 - 문법 규칙 적용 순서를 명확히 파악 가능

#### 4.2 parserTest.c 작성

```

#include <stdio.h>
#include <stdlib.h>
#include "Parser.h"

extern FILE* source; // Parser.c 의 source 변수 참조

int main(int argc, char* argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <source file>\n", argv[0]);
        return 1;
    }
}

```

```

}

source = fopen(argv[1], "r");
if (source == NULL) {
    fprintf(stderr, "Error: Cannot open source file '%s'\n", argv[1]);
    return 1;
}

printf("--- Start Parsing ---\n");
parser();
printf("--- Parsing Finished ---\n");

fclose(source);
return 0;
}

```

**기능:** 1. 명령줄 인자로 소스 파일명 받기 2. 파일 열기 및 에러 처리 3. parser() 함수 호출 4. 파일 닫기

**설계 결정:** - 표준 C 함수(fopen) 사용으로 크로스 플랫폼 호환성 확보 - 명확한 에러 메시지 제공 - extern 선언으로 Parser.c 의 source 변수 공유

#### 4.3 Scanner.h 수정

**수정 1: NO\_KEYWORD 증가**

```
// 수정 전
#define NO_KEYWORD 7
```

```
// 수정 후
#define NO_KEYWORD 14
```

**이유:** MiniC 문법의 모든 키워드 포함 (14 개)

**수정 2: enum tsymbol 재정렬**

```
// 수정 전
enum tsymbol {
    tnull = -1,
    tnot, tnotequ, tremainder, tremAssign, tident, tnumber,
    tand, tlparen, trparen, tmul, tmulAssign, tplus,
```

```

tinc, taddAssign, tcomma, tminus, tdec, tsubAssign,
tdiv, tdivAssign, tsemicolon, tless, tlesse, // tcolon 없음, 순서 잘못
tassign, tequal, tgreat, tgreat, tlbracket, trbracket,
teof, // 위치가 30 번이 아님
tconst, telse, tif, tint, treturn, tvoid, twhile, // 7개만, 순서 불일치
tlbrace, tor, trbrace
};

// 수정 후
enum tsymbol {
tnull = -1,
tnot, tnotequ, tremainder, tremAssign, tident, tnumber,
/* 0      1      2      3      4      5      */
tand, tlparen, trparen, tmul, tmulAssign, tplus,
/* 6      7      8      9      10     11     */
tinc, taddAssign, tcomma, tminus, tdec, tsubAssign,
/* 12     13     14     15     16     17     */
tdiv, tdivAssign, tcolon, tsemicolon, tless, tlesse,
/* 18     19     20     21     22     23     */
tassign, tequal, tgreat, tgreat, tlbracket, trbracket,
/* 24     25     26     27     28     29     */
teof,
/* 30 */
tbreak, tcase, tconst, tcontinue, tdefault,
/* 31     32     33     34     35     */
tdo, telse, tfor, tif, tint, treturn,
/* 36     37     38     39     40     41     */
tswitch, tvoid, twhile, tlbrace, tor, trbrace
/* 42     43     44     45     46     47     */
};

```

**주요 변경:** 1. tcolon 을 20 번 위치에 추가 2. teof 를 정확히 30 번 위치에 배치 3. 키워드들을 파싱 테이블 순서에 맞춰 재배열: - break(31), case(32), const(33), continue(34), default(35) - do(36), else(37), for(38), if(39), int(40), return(41) - switch(42), void(43), while(44) 4. 주석으로 각 토큰의 인덱스 명시

**검증:** MiniC.lst 의 terminal symbol 순서와 완벽히 일치하도록 정렬

#### 4.4 Scanner.c 수정

수정 1: tnum 배열 업데이트

// 수정 전

```
enum tsymbol tnum[NO_KEYWORD] = {  
    tconst, telse, tif, tint, treturn, tvoid, twhile  
};
```

// 수정 후

```
enum tsymbol tnum[NO_KEYWORD] = {  
    tbreak, tcase, tconst, tcontinue, tdefault, tdo, telse,  
    tfor, tif, tint, treturn, tswitch, tvoid, twhile  
};
```

**변경 사항:** - 14 개 모든 키워드의 토큰 번호 포함 - 알파벳 순서로 정렬 (키워드 검색 시 일치하도록)

수정 2: keyword 배열 업데이트

// 수정 전

```
char* keyword[NO_KEYWORD] = {  
    "const", "else", "if", "int", "return", "void", "while"  
};
```

// 수정 후

```
char* keyword[NO_KEYWORD] = {  
    "break", "case", "const", "continue", "default", "do", "else",  
    "for", "if", "int", "return", "switch", "void", "while"  
};
```

**변경 사항:** - 14 개 모든 키워드 문자열 포함 - 알파벳 순서로 정렬 - tnum 배열과 정확히 일치하는 순서

수정 3: tokenName 배열에 colon 추가

// 수정 전

```
char* tokenName[] = {  
    "!",      "!=" ,      "%",      "%=",      "%ident",      "%number",  
    ...  
    "/",      "/=",      ";",      "<",      "<=",      // colon 없음  
    ...
```

```

};

// 수정 후
char* tokenName[] = {
    "!",      "!=" ,   "%",      "%=",     "%ident",   "%number",
    /* 0       1      2        3        4        5        */
    "&& ",   "(" ,   ")" ,      "*",      "*=",      "+",
    /* 6       7      8        9        10       11       */
    "++ ",    "+=" ,   ",",      "-",      "--",      "-=",
    /* 12      13     14       15       16       17       */
    "/",      "/=" ,   ":" ,      ";",      "<",      "<=",
    /* 18      19     20       21       22       23       */
    "=" ,    "==" ,   ">",      ">=",     "[",      "]",
    /* 24      25     26       27       28       29       */
    "eof",
    /* 30 */
    "break",  "case",  "const",  "continue", "default",
    /* 31      32     33       34       35       */
    "do",     "else",  "for",   "if",     "int",     "return",
    /* 36      37     38       39       40       41       */
    "switch", "void",  "while", "{",      "||",      "}"
    /* 42      43     44       45       46       47       */
};


```

**변경 사항:** 1. 20 번 위치에 ":" (colon) 추가 2. 31-44 번 위치에 14 개 키워드 추가 3. 주석으로 인덱스 명시하여 가독성 향상

**수정 4:** *strcpy\_s* 를 *strcpy* 로 변경

```

// 수정 전 (Scanner.c 의 scanner 함수 내)
strcpy_s(token.value.id, sizeof(token.value.id), id);


```

```

// 수정 후
strcpy(token.value.id, id);


```

**이유:** WSL/Linux 환경에서 표준 C 함수 사용

**수정 5:** colon 토큰 처리 추가

```

// scanner() 함수의 switch 문에 추가
case ':':
```

```
token.number = tcolon;
break;
```

**이유:** MiniC 문법에서 switch-case 문의 label에 사용되는 ‘:’ 처리

## 5. 실험 결과

### 5.1 컴파일

```
$ gcc -o parser parserTest.c Parser.c '../MiniC Scanner/Scanner.c' \
-I'../MiniC Scanner' -I'../MiniC Grammar' -Wall
```

**결과:** 경고 없이 성공적으로 컴파일됨

### 5.2 테스트 케이스 1: factorial.mc

**소스 코드:**

```
void main(void)
{
    int n, result;
    n = 5;
    result = factorial(n);
}

int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

**실행 결과:**

```
--- Start Parsing ---
reduced rule number = 15
reduced rule number = 12
reduced rule number = 9
...
```

(총 247 개 규칙 reduce)

...

reduced rule number = 1

\*\*\* valid source \*\*\*

\*\*\* Right Parse \*\*\*

```
1 3 4 6 23 37 39 43 59 50 67 52 68 69 76 78 80 83 88 92
95102105107108 69 76 78 80 83 90 91 95100111 88 91 95100110
100110 91 95100110 50 67 52 68 69 76 78 80 83 88 91 95100111
68 69 76 78 81 83 88 91 95100111 80 83 88 91 95100110 25 7
17 18 20 22 33 8 9 12 14 16 8 9 12 14 2 4 6 23 37 40
42 51 52 68 69 76 78 80 83 88 91 95102105107108 69 76 78 80
83 88 91 95100110100110 40 42 51 52 68 70 69 76 78 80 83 88
91 95102105107108 69 76 78 80 83 88 91 95100110100110 95100
110 40 42 51 52 68 69 76 78 80 83 88 91 95102105107108 69 76
78 80 83 88 91 95100110100110 39 42 51 52 68 69 76 78 80 83
88 91 95102105107108 69 76 78 80 83 88 91 95100110100110 24
26 28 30 31 33 29 31 33 8 9 12 14 7 17 19 16 8 9 12 15
```

--- Parsing Finished ---

**분석:** - 파싱 성공: “\*\*\* valid source \*\*\*” 메시지 출력 - Right parse 가 역순으로 출력됨 - 규칙 1번(start symbol)이 마지막에 위치 - 총 247 개 규칙이 적용되어 factorial 프로그램 인식

### 5.3 테스트 케이스 2: bubble.mc

**소스 코드:** 버블 정렬 알고리즘 구현 (약 50 줄)

**실행 결과:**

--- Start Parsing ---

reduced rule number = 15

...

(총 602 개 규칙 reduce)

...

\*\*\* valid source \*\*\*

```

*** Right Parse ***
1 2 4 6 23 37 40 44 60 41 23 37 40 42 51 52 68 69 76 78
80 83 88 91 98 95100110 39 42 51 52 68 69 76 78 80 83 88 91
95102105107108 69 76 78 80 83 88 91 95101 68 69 76 78 80 83
...
--- Parsing Finished ---

```

**분석:** - 파싱 성공 - 복잡한 제어 구조(중첩 루프)를 포함한 프로그램 처리 - 602 개 규칙 적용으로 factorial 보다 복잡한 구조 확인

#### 5.4 테스트 케이스 3: perfect.mc

**소스 코드:** 완전수 판별 프로그램

**실행 결과:**

```

--- Start Parsing ---
reduced rule number = 13
...
(총 367 개 규칙 reduce)
...
*** valid source ***

*** Right Parse ***
1 3 4 6 23 37 40 44 60 41 23 37 40 42 51 52 68 69 76 78
80 83 88 91 98 95100110 40 43 58 42 51 52 68 71 69 76 78 80
...
--- Parsing Finished ---

```

**분석:** - 파싱 성공 - switch-case 문 포함 프로그램 처리 - colon 토큰 정상 인식 확인

#### 5.5 테스트 결과 요약

프로그램	줄 수	규칙 수	결과	특이사항
factorial.mc	12	247	성공	재귀 함수
bubble.mc	50	602	성공	중첩 루프, 배열
perfect.mc	35	367	성공	switch-case

**모든 테스트 케이스 통과**

## 6. Right Parse 분석

### 6.1 Right Parse 의 의미

Bottom-up 파싱과 Rightmost Derivation 의 관계:

#### 1. LR 파싱 동작:

- 입력을 왼쪽에서 오른쪽으로 읽음 (Left-to-right)
- Rightmost derivation 의 역순으로 reduce 수행
- 각 reduce 는 생성 규칙을 역으로 적용

#### 2. Right Parse 출력:

- reduce 된 규칙들을 시간순으로 기록
- 역순으로 출력하면 Top-down Rightmost derivation 과 동일
- 프로그램의 구문 구조를 생성 순서대로 표현

### 6.2 factorial.mc 의 Right Parse 해석

출력된 Right Parse 의 일부 규칙을 Minic 문법으로 해석:

```
Rule 1: program -> external_dcl
Rule 3: external_dcl -> external_dcl external_dcl
Rule 4: external_dcl -> function_def
Rule 6: function_def -> function_header compound_statement
...
Rule 15: var_dcl -> type_specifier IDENT SEMICOLON
Rule 33: type_specifier -> INT
Rule 43: statement -> expression_statement
...
```

**해석:** 1. Rule 1: 전체 프로그램 시작 2. Rule 3-6: main 함수와 factorial 함수 정의 3. Rule 15, 33: 변수 선언 (int n, result) 4. Rule 43+: 함수 호출 및 대입문 5. ...: 재귀 호출 및 조건문 처리

### 6.3 규칙 적용 빈도 분석

factorial.mc에서 자주 사용된 규칙:

규칙 번호	빈도	의미
69	12 회	primary → IDENT
76	12 회	postfix_expression → primary
78	12 회	unary_expression → postfix_expression
95	16 회	relational_expression → ...
100	20 회	assignment_expression → ...
110	16 회	expression → assignment_expression

**분석:** - 식(expression) 관련 규칙들이 가장 빈번 - 식별자와 함수 호출이 많이 사용됨 - 재귀 호출로 인한 반복 패턴 관찰

## 7. 결론

### 7.1 구현 결과

본 프로젝트에서 다음을 성공적으로 구현하였습니다:

#### 1. Right Parse 출력 기능

- reduce 된 문법 규칙을 배열에 저장
- 역순 출력으로 Rightmost derivation 표현
- 20 개 규칙마다 줄바꿈으로 가독성 향상

#### 2. Scanner-Parser 통합

- 토큰 번호를 파싱 테이블과 일치하도록 조정
- 14 개 모든 MiniC 키워드 지원
- 표준 C 함수 사용으로 크로스 플랫폼 호환성 확보

#### 3. 테스트 프로그램

- 명령줄 인자로 소스 파일 지정
- 명확한 에러 메시지 제공
- 파싱 시작/종료 표시

### 7.2 학습 내용

#### 1. LR 파싱 메커니즘:

- Shift/Reduce/Accept 액션의 동작 원리
- 파싱 테이블의 구조와 참조 방법
- Bottom-up 파싱과 Rightmost derivation의 관계

## 2. 토큰 설계의 중요성:

- 토큰 번호는 파싱 테이블과 정확히 일치해야 함
- enum 순서가 파서 동작에 직접적 영향
- 일관된 심볼 번호 체계의 필요성

### 7.3 소감

심볼 테이블을 분석하는 것에서 많은 어려움이 있었다. 특히 :등 누락된 심볼을 찾아내는 부분이 가장 힘들었다. 다만 앞으로의 과정 중에서 가장 어려우리라 생각된 부분이 잘 처리되어 기쁘다.