

# Compiler Term Project

## 1. Scanner Report

2014004211

컴퓨터소프트웨어학부


표영권

## 과제 요구사항

1. C-Minus 언어의 Scanner를 C implementation으로 구현
2. C- Minus 언어의 Scanner를 lex file로 구현
3. 모두 컴파일 및 실행에 이상이 없어야 함
4. C- Minus 언어의 Lexical convention:

### Lexical Convention of C-Minus

- **Keyword**  
else if int return void while (lower case)
- **Symbol**  
+ - \* / < <= > >= == != = ; ,  
( ) [ ] { } /\* \*/
- **Token**  
*ID = letter letter\**  
*NUM = digit digit \**  
*letter = a | ... | z | A | ... | Z*  
*digit = 0 | 1 | ... | 9*

Hanyang University  
Division of Computer Science & Engineering

## 실행 방법

- Build : `$make`
- C implementation : `$ ./cminus_cimpl [테스트할 파일명]`
- Flex : `$ ./cminus_flex [테스트할 파일명]`
- Remove output files : `$make clean`

## 구현 과정

작업 환경은 Ubuntu Linux 20.x 버전에서 진행하였다. Program code set은 K. Louden Tiny C code set을 다운받아, 이번 과제의 C-Minus 언어에서 쓰이지 않는 token 및 state들을 제거하고 시작했다. 이는 global.h, scan.c, parse.c에 적용하였으며 이 세 가지가 lexical analysis를 실행하는 source files라고 볼 수 있다. 그리고 새롭게 curly brace, bracket(called brace in this subject code), greater or equal, not equal, less or equal, semi colon, comma의 token들과 이에 해당되는 각 state들, 그리고 comment에 대한 state를 추가한다. 또한 결과를 echo하는 print implementation 부분에서도 'Tiny C' 대신 'C minus'로 출력하도록 수정하였으며 이는 Makefile 또한 적용하였다.

## 가. cminus\_cimpl (C code로 구현)

Single letter token들의 경우 conflict가 없으므로 기존에 주어진 transition 방식과 일치하지만, 그들 중 LE, GE 등과 같이 다음 symbol(token)을 점검해야 하는 경우가 있으므로 새로운 state를 추가하여 다음 look-ahead symbol을 보고 state transition을 작성하였다.

```
break; // state state ended.
case INEQ:
    state = DONE;
    if (c != '=')
    {
        /* backup in the input */
        ungetNextChar();
        save = FALSE;
        currentToken = ASSIGN;
    }
    else currentToken = EQ;
    break;
case INNE:
```

위 그림에서 INEQ는 state이며, token '=' 1개를 처음으로 입력받았을 때 transition되는 state이다. 그 다음 look-ahead symbol이 '='이 아닌 다른 문자일 경우 현재의 token을 assignment로 하고, '='일 경우에는 비교 연산을 수행하고 있음을 표기한다. 마찬가지로 2개의 symbol을 인식하여 1개의 token으로 처리하는 다른 state 및 token들도 이와 같이 구현하였다.

그러나 comment의 경우 시작부분인 '/'을 입력받고 끝나는 것이 아니라 끝나는 부분인 '\*/'또한 입력받아야 해당 comment state가 종료됨을 알 수 있고, 첫 token을 scan하는 과정에서 '/'에 의해 INOVER state가 될 수도 있다. 이 경우 GE, LE, EQ와 마찬가지로 '/'를 입력받은 뒤 '\*'가 나오지 않으면 나눗셈 state임을 확정하도록 하여, '/'를 입력받으면 INCOMMENT\_state로, 그 후 '\*'를 입력받으면 INCOMMENT state, 그 외의 다른 symbol이 입력되면 INOVER로 transition하도록 구성하였다.

```
case INCOMMENT:
    save = FALSE;
    if (c == EOF)
    { state = DONE;
      currentToken = ERROR; // need to close
    }
    else if (c == '/*')
    {
        c = getNextChar();
        if(c == '/') state = START;
    }
    break;
case INCOMMENT_: // now one '/' character read.
    save = FALSE;
    if (c == '/*') state = INCOMMENT;
    else // input whitespace, id, num, etc.
    {
        ungetNextChar();
        save = TRUE;
        c = '/';
        state = INOVER;
        currentToken = OVER;
    }
    break;
case INOVER:
    state = DONE;
    save = FALSE;
    if (c == ' ' || c == '\t') currentToken = OVER;
    else if (!(isdigit(c) || isalpha(c) || c == '(')) // not in FIRST(factor)
    { currentToken = ERROR; }
    ungetNextChar();
    break;
```

INOVER state에서는 나눗셈 연산 기호 '/' 이후에 올 수 있는 terminal symbols(equals FOLLOW('/'))가 아닐 경우 이 Language에 나타나지 않는 경우이므로 currentToken을 ERROR가 되게 하였다.

INCOMMENT state는 실제로 이 state에 있는 동안 나타나는 모든 symbol들은 계속 이 state 내에 있게 되도록 작성하였다. 만약 아직 INCOMMENT state가 끝나지 않은 상태에서 EOF를 만났을 경우에는 ERROR로 처리하도록 하였으며 '\*'를 입력받은 뒤 '/'가 곧바로 나타나지 않는 한 모든 symbol에 대해 계속 INCOMMENT state에 머물도록 작성하였다. 또한 COMMENT는 실제적인 기능을 수행하지 않기 때문에 해당 state가 종료되어도 DONE state가 아닌 START state로 transition 한다.

#### 나. cminus\_flex (flex.l file로 구현)

Flex는 regular expression을 기준으로 token stream에서 tokenize하므로, C code로 구현했을 때 보다 conflict를 따지지 않아도 된다. Regular expression은 longest match first이므로 "="과 "=="를 그대로 rule action을 기입하여도 별도로 정상적으로 동작하기 때문이다. 그러나 문제는 comment에 있다. Comment의 경우 nested되면 안 된다는 조건이 있기 때문에 longest match first인 regular expression을 이용하기 위해서는 "/\*"가 입력된 후 가장 먼저 "\*/"를 만나기까지의 모든 string을 accept하도록 comment rule을 작성하여야 한다.

이번 과제에서는 이를 해결하기 위해 regular expression의 [^ ] rule을 이용하였다. "/\*"를 입력받은 이후 최초로 "\*/"를 입력받지 않는 모든 string을 comment 내의 string으로 처리한 후 마지막으로 "\*/"를 받기까지의 string을 comment로 규정하도록 regular expression을 작성하였다.

```
15
16 digit      [0-9]
17 number     {digit}+
18 letter     [a-zA-Z]
19 identifier {letter}+
20 newline    \n
21 whitespace [ \t]+
22 comment    \/\/*(^[*\/](\/\*)|.){whitespace}|{newline})*\*\/
23
```

현재 comment의 regular expression은 nested issue가 있을 수 있겠으나, 해당 과제에서는 그러한 string은 Language에 없다는 확인을 받고 consideration에서 제외하였다.

## 실행 결과

<pre> 1 /* A program to perform Euclid's 2 Algorithm to computer gcd */ 3 4 int gcd (int u, int v) 5 { 6     if (v == 0) return u; 7     else return gcd(v,u-u/v*v); 8     /* u-u/v*v == u mod v */ 9 } 10 11 void main(void) 12 { 13     int x; int y; 14     x = input(); y = input(); 15     output(gcd(x,y)); 16 } </pre> <p>test.tn 1,1 모두</p>	<pre> 1 2 C-MINUS COMPILATION: test.tn 3 1: /* A program to perform Euclid's 4 2: Algorithm to computer gcd */ 5 3: 6 4: int gcd (int u, int v) 7 4: reserved word: int 8 4: ID, name= gcd 9 4: { 10 4: reserved word: int 11 4: ID, name= u 12 4: , 13 4: reserved word: int 14 4: ID, name= v 15 4: ) 16 5: { 17 5: { 18 6: if (v == 0) return u; 19 6: reserved word: if 20 6: { 21 6: ID, name= v 22 6: == 23 6: NUM, val= 0 24 6: } 25 6: reserved word: return 26 6: ID, name= u 27 6: ; 28 7: else return gcd(v,u-u/v*v); 29 7: reserved word: else 30 7: reserved word: return 31 7: ID, name= gcd 32 7: { 33 7: ID, name= v 34 7: , 35 7: ID, name= u 36 7: - 37 7: ID, name= u 38 7: / 39 7: ID, name= v 40 7: * 41 7: ID, name= v 42 7: ) 43 7: ; 44 8: /* u-u/v*v == u mod v */ 45 9: } 46 9: } 47 10: 48 11: void main(void) 49 11: reserved word: void 50 11: ID, name= main 51 11: { 52 11: reserved word: void 53 11: ) 54 12: { </pre> <p>1,0-1 꼭대기</p>	<pre> 1 2 C-MINUS COMPILATION: test.tn 3 3: reserved word: int 4 3: ID, name= gcd 5 3: { 6 3: reserved word: int 7 3: ID, name= u 8 3: , 9 3: reserved word: int 10 3: ID, name= v 11 3: ) 12 4: { 13 5: reserved word: if 14 5: { 15 5: ID, name= v 16 5: == 17 5: NUM, val= 0 18 5: } 19 5: reserved word: return 20 5: ID, name= u 21 5: ; 22 6: reserved word: else 23 6: reserved word: return 24 6: ID, name= gcd 25 6: { 26 6: ID, name= v 27 6: , 28 6: ID, name= u 29 6: - 30 6: ID, name= u 31 6: / 32 6: ID, name= v 33 6: * 34 6: ID, name= v 35 6: ) 36 6: ; 37 8: } 38 10: reserved word: void 39 10: ID, name= main 40 10: { 41 10: reserved word: void 42 10: ) 43 11: { 44 12: reserved word: int 45 12: ID, name= x 46 12: ; 47 12: reserved word: int 48 12: ID, name= y 49 12: ; 50 13: ID, name= x 51 13: = 52 13: ID, name= input 53 13: { 54 13: { </pre> <p>1,0-1</p>
---	--	---

좌측은 예제 파일, 중간은 C implementation code, 우측은 flex code로 구현한 실행 파일의 결과를 보여준다. 가장 위쪽을 보면 cimpl result의 경우 comment가 classification 없이 그대로 출력되는 모습이, flex result의 경우에는 이 문장이 인식되지 않도록 처리가 된 모습을 볼 수 있다. 그 외의 id, reserved word 등도 잘 accept 되는 것을 볼 수 있다.

## 구현 중 나타난 issue와 solution

C implementation의 경우, comment를 처리하기 위해 DFA를 작성하는 것이 관건이었다. '/' symbol을 받았을 때 의미상 INOVER state check를 먼저 해야 할지, INCOMMENT\_ state check를 먼저 해야 할지부터 고민이 되었으나, INCOMMENT\_를 먼저 할 경우 INOVER와 INCOMMENT가 병렬적으로 transition determinance를 진행하지만 INOVER를 먼저 할 경우 INOVER state -> INCOMMENT\_ state -> COMMENT state의 과정을 거치게 되어 그 depth가 늘어나 비효율적이라고 판단하였다. 따라서 INCOMMENT\_ state로 먼저 transition한 후, 다음 look-ahead symbol이 '\*' 인지 아닌지만을 두고 INOVER, INCOMMENT state로 나뉘어 transition하게끔 작성하였다.

효율적인 DFA를 작성하는 일뿐만 아니라, C 언어의 기술적인 issue도 있었다. reservedLookup 함수에서 reserved token을 가장 먼저 수정하는 바람에 NULL 값이 나타나는 경우도 있었는데, 이는 make를 하는 도중 core dump error가 나타나 디버깅을 통해 알아내었다. 따라서 다음과 같이 NULL이 아닐 경우에 대한 condition을 추가해주었다.

```

/* lookup an identifier to see if it is a reserved word */
/* uses linear search */
static TokenType reservedLookup (char * s)
{ int i;
  for (i=0;i<MAXRESERVED;i++)
  {
    //printf("i:%d, %s/%s\n", i, s, reservedWords[i].str);
    if (reservedWords[i].str != NULL && !strcmp(s,reservedWords[i].str))
    {
      return reservedWords[i].tok;
    }
  }
  return ID;
}

```

종료를 위해 symbol check를 하는 progress 또한 'save' variable, 'currentToken' variable의 flow를 이해해야 완벽하게 작성할 수 있었다. INCOMMENT\_ state에서는 기본적으로 currentToken을 OVER로 설정해준 뒤 '\*' symbol이 detect되었을 경우에만 INCOMMENT state로 진행되게끔 하였으며, 그렇지 않았을 경우에는 'save' variable의 값을 True로 해줌으로써 나눗셈 symbol을 읽었다는 의미를 명확히 하였다.

Flex implementation의 경우에는 comment에 대해 간단하게 regular expression을 작성하였다가, nested되지 않았음에도 가장 먼저 나타난 '/\*'와 가장 마지막에 나타난 '\*/'를 모두 comment로 인식하는 것을 보고 longest match first가 기본적으로 수행됨을 깨닫고 이에 대한 issue를 해결하기 위한 condition을 명확한 문장으로 정리하는 것으로부터 시작하였다.

"/\*를 입력받은 후, 가장 먼저 \*/를 만나는 string"

위 문장으로 출발하였다. 여기서 "가장 먼저 \*/를 만난다"는 의미를 재구성하면

"/\*를 입력받은 후, \*/가 아닌 모든 substring 후 \*/를 만나는 string"

으로 바꿀 수 있게 된다. 여기서 'a가 아닌 모든 substring'은 '[^a].\*'로 표현될 수 있기 때문에 이 issue는 이렇게 해결되었다.