Design and Development of Compiler for C- Language
(설계 프로젝트 수행 결과)

과목명: [CSE4120] 기초 컴파일러 구성
담당교수: 서강대학교 컴퓨터공학과 정 성 원
개발자: 20151623 한상구

개발기간: 2019. 3. 27 - 2019. 3. 27

각 단계별 결과 보 고 서

프로젝트 제목: Design and Development of Compiler for C-Language:
Phase 1: Design and Implementation of Lexical Analyzer
제출일: 2019. 3. 27.
개발자: 한상구

I. 개발 목표

Flex(or Lex)를 이용하여 C- language 에 대해 lexical analysis 를 수행할 수 있는 lexical analyzer 를 구현한다.

C- language 에 관한 세부 내용은 교재[1] Appendix A 를 참조하며, Makefile 을 통해 build 한 analyzer 는 주어진 *.tny 파일의 token table 을 적절하게 생성할 수 있어야 한다.

큰 구조는 교재에서 주어졌으며, 교재에서 제공하는 틀 위에 새로운 token 을 추가하고 comment error 를 잡아낼 수 있도록 설계한다.

II. 개발 범위 및 내용

가. 개발 범위

1. Flex 를 이용한 tokenizing, token 을 이용한 적절한 token table 출력
2. Comment handling 및 Comment related-error handling

나. 개발 내용

1. Lex 문법을 이용한 적절한 rule 및 aux routine 설정
2. Comment digest 도중 발생할 수 있는 error 에 관한 logic 구상 및 handling

III. 추진 일정 및 개발 방법

가. 추진 일정

~2019.03.27 lexical analyzer 설계 및 구현

나. 개발 방법

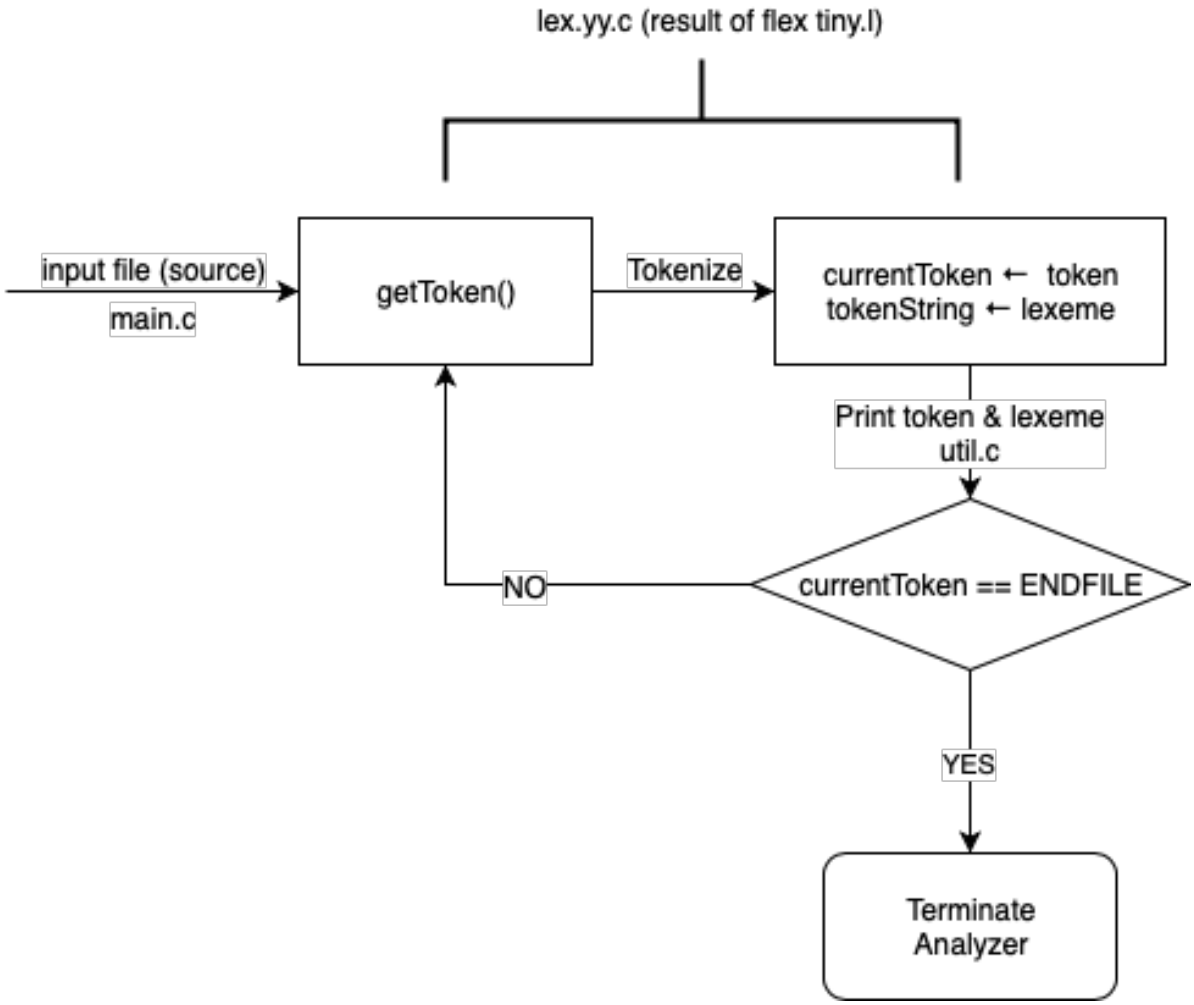
Flex 를 이용하여 tokenize (lex), 이후 주어진 token 을 이용하여 token table 출력 (C)

---

[1] Compiler Construction Principles and Practice, K. C. Louden

IV. 연구 결과

1. 합성 내용:



<그림 1 - C- lexical analyzer 의 간단한 flow chart>

tiny.l 의 aux subroutine section 에서 정의했던 getToken()을 중심으로 작동되는 analyzer 입니다. 크게 관찰하자면, 주어진 source 에서 EOF 를 읽어들일 때 까지 source 를 digest 하고, 그 과정을 token 단위로 진행시킬 수 있도록 작성되었습니다.

2. 분석 내용:

본 프로젝트에서 요구한 파일은 총 5 개입니다. 위 flow chart 와 연계하여 각 구성 요소를 설명하겠습니다.

- main.c

Source file 을 읽고, getToken()을 호출하며 digest 하고, analyze 가 끝나면 analyzer 를 종료시킵니다. 위 flow chart 의 시작과 끝이 본 코드에 구현되어 있습니다.

- globals.h

위 flow chart 에서 전체적으로 사용되는 (서로 다른 c 파일에서 사용되는) 변수 및 타입, 매크로를 가지고 있는 헤더파일 입니다. token type, source, output buffer 등이 선언되어 있습니다.

- util.c, util.h

위 flow chart 에서 token table 을 출력하는 (Print token & lexeme) 코드가 구현되어 있습니다.

- tiny.l

lex 문법을 이용해 tokenize rule, aux subroutine (getToken(), yyerror()) 을 구현한 코드입니다.

본 프로젝트에서 요구하는 rule 을 regular expression & exact matching 을 통해 선언하고, 이 rule 에 따라 주어진 source 를 getToken()을 통해 digest 하게 됩니다.

3. 제작 내용:

globals.h, main.c, util.h 의 경우 linux kernel coding style 에 따라 prettify 한 것, 본 프로젝트에 사용되지 않는 부분은 전부 덜어낸 것, 간단한 출력 함수의 사용 외에 로직상 변경사항이나 특이점이 없기 때문에 서술하지 않겠습니다.

Comment digest 가 주된 관건이었던 프로젝트였기 때문에, tiny.l 에서 어떻게 error handling 을 할 것인가에 대한 설계가 중요했습니다. 아직 lexical analysis 이기 때문에, syntax error 는 잡아낼 수 없는 환경이므로 comment digest 도중 error 가 발생할 수 있는 상황은 */가 읽히기 전에 EOF 가 읽히는 상황뿐입니다.

```
"/*"            {
                  register int c;
                  for (;;) {
                          while ((c = input()) != '*' && c != EOF)
                                  lineno += c == '\n';
                          if (c == '*') {
                                  while ((c = input()) == '*');
                                  lineno += c == '\n';
                                  if ( c == '/' )
                                          break;
                          }
                          if (c == EOF) {
                                  yyerror("Comment Error");
                                  return ERROR;
                          }
                  }
                }
```

<그림 2 - tiny.l 에서 comment digest 를 위해 선언한 rule>

/* token 이 매칭되면, */가 들어오기 전까지 모든 입력 문자를 무시합니다. line number 를 놓치지 않고 계산하기 위해 무시되는 문자가 \n 일 시 line number 를 증가시키는 것을 확인할 수 있으며, */ 이전에 EOF 가 들어오는 경우 ERROR token 을 반환하는 것을 확인할 수 있습니다.

```
void yyerror(char const *msg)
{
        strcpy(yytext, msg);
}
```

<그림 3 - tiny.l 에서 error message dumping 을 위해 선언한 aux subroutine>

yyerror 는 ERROR token 발생 시, lex.yy.c 이외의 파일에서 따로 error message handling routine 을 구현하지 않기 위해 작성한 aux subroutine 입니다. lex 에서 사용하는 변수인 yytext 에 error message 를 dump 함으로써 기존 다른 token 이 table 에 출력되는 양식을 그대로 사용할 수 있도록 했습니다. (tokenString 은 yytext 에서 값을 dump 해오는 것을 이용했습니다)

위 2 가지 구현 이후, 모든 토큰에 대해 동일한 로직을 통해 출력을 진행할 수 있었고, 다음과 같은 tokenName 이라는 enum-name table 을 이용해 간단하게 구현할 수 있었습니다

```
static char const *tokenName[] = {
        "EOF", "ERROR", "IF", "ELSE", "INT", "RETURN", "VOID",
        "WHILE", "ID", "NUM", "=", "==", "!=", "<", "<=", ">",
        ">=", "+", "-", "*", "/", "(", ")", "[", "]", "{", "}",
        ",", ";"
};
```

<그림 4 - util.c 에서 tokenType enum 과 매칭되는 이름을 담은 array>

```
void printToken(TokenType token, const char *tokenString)
{
        switch (token) {
        case IF:
        case ELSE:
        case INT:
        case RETURN:
        case VOID:
        case WHILE:
        case ASSIGN:
        case EQ:
        case NEQ:
        case LT:
        case LTE:
        case GT:
        case GTE:
        case LPAREN:
        case RPAREN:
        case LBRAC:
        case RBRAC:
        case LCURLY:
        case RCURLY:
        case COMMA:
        case SEMI:
        case PLUS:
        case MINUS:
        case TIMES:
        case OVER:
        case ENDFILE:
        case NUM:
        case ID:
        case ERROR:
                fprintf(listing, "\t\t%s\t\t%s\n",
                        tokenName[token], tokenString);
                break;
        default: /* should never happen */
                fprintf(listing, "Unknown token: %d\n", token);
        }
}
```

<그림 5 - yyerror()를 통한 인터페이스 통일, enum-name array 를 이용해 간단히 구현한 printToken>

모든 token 의 출력 인터페이스를 통일함에 따라 switch fallthorugh 를 통해 기존 로직보다 더욱 간단하게 token table 에서 token 정보를 출력할 수 있도록 구현할 수 있었습니다.

4. 시험 내용:

기본적인 binary search 를 C- token 으로만 구현한 파일 기반으로 두 개의 테스트 파일을 만들어 시험했습니다. 본 프로젝트에서 가장 중요한 부분은 Comment digest 이기 때문에, line number 가 잘 계산 되고 있는가, Comment 가 닫히지 않은 상태에서 EOF 가 들어오는가 (comment error)를 확인할 수 있도록 테스트 파일을 구성했습니다.

코드 중간중간 multi line comment, single line comment 를 삽입하여 line number 를 잘 쫓아가는 것을 확인할 수 있었고, /* ~ */ 사이의 글자를 digest 함에 있어 문제가 없음을 확인할 수 있었습니다.

라인 중간에 삽입 된 single line comment 또한 성공적으로 digest, 이후 token 을 정상적으로 digest 하는 것을 확인 할 수 있었습니다.

/* 로 multiline comment 가 열린 이후, */ 로 comment 가 닫히지 않은 채 EOF 를 읽을 경우, 성공적으로 ERROR 를 감지해내고 lex 내부 변수인 yytext 에 에러메시지를 담아 tokenString 에서 에러 메시지를 읽어 들일 수 있었습니다.

5. 평가 내용:

이번 프로젝트에서 요구하는 기능 모두를 구현하였습니다. 추가로, yyerror()를 통해 ERROR TOKEN 의 경우에도 token 출력 인터페이스를 통일함으로써 enum-name table 을 이용, 기존 간단하지만 반복되는 번잡한 코드에 비해 간결하고 가독성 좋은 코드를 작성할 수 있었습니다.

V. 기타

1. 자체 평가:

기본적으로 요구하는 프로그램의 동작 이외에도, 프로그래머로서 고려해야할 유지보수에 관해 고민하고 더 좋은 생산성을 낼 수 있도록 노력했습니다. 일관된 coding convention 을 적용하여 코드의 가독성을 높이고, 비슷한 기능을 하는 인터페이스에 관하여 통일할 수 있도록 설계하였습니다.

그 결과 가독성이 높은 코드와, 유지 보수를 위해 여러 파일의 코드를 점프하는 것이 아닌 해당 인터페이스에 관한 코드만 수정하면 완료할 수 있는 환경을 만들 수 있었습니다.

2. 기타 자유 서술:

lex 의 동작 방식 및 문법, 그리고 programming language 가 어떻게 analyze 되는지에 대한 전반적인 시야를 얻을 수 있었다.