Design and Development of Compiler for C- Language

(설계 프로젝트 수행 결과)

1. 각 단계별 결과 보고서

과목명 : [CSE4120] 기초 컴파일러 구성

담당교수 : 서강대학교 컴퓨터공학과 정성원

개발자 : 엄태경

개발기간 : 2019. 3. 19. ~ 2019. 3. 27

각 단계별 결과 보고서

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

I. 개발 목표

본 프로젝트는 C- 프로그래밍 언어를 처리할 수 있는 컴파일러 구현의 첫 단계로, 어휘분석기(Lexical Analyzer)를 개발하는 것을 목표로 한다. C- 언어는 수업 교재인 [1]의 Appendix A 에 정의된 언어로, 적은 수의 어휘와 문법만을 사용하도록 구성된 C 언어의 부분집합이다. 어휘분석(Lexical Analysis)은 문자열로 된 소스코드를 어휘(lexeme) 단위로 분석하여 토큰 열로 표현하는 작업이다.

II. 개발 범위 및 내용목표

가. 개발 범위

본 프로젝트에서는 [1]의 Appendix B 에 제시된 TINY 언어의 컴파일러 소스코드 중 어휘분석기에 필요한 부분만을 발췌하여 C- 언어에 맞게 수정한다. 해당하는 파일은 C 언어로 작성된 "main.c", "globals.h", "util.h", ""util.c", 그리고 Lex 문법에 따라 작성된 "tiny.l"이다. 여기에 더해 소스코드를 컴파일하고 실행파일을 생성하는 명령어를 처리하는 "Makefile"도 관련 소스코드의 수정에 따라 고쳐써야 한다.

나. 개발 내용

C- 언어는 29 개의 토큰 (키워드 6 개, 특수기호 21 개, 기타 2 개)으로 이루어진다. [1]에 명시된 규칙에 따라 각각의 토큰을 정규표현식으로 정리하여 LEX 파일로 구현하고, flex 툴을 이용해 이를 C 언어로 변환하여 나머지 C 언어 소스코드와 함께 컴파일해 어휘분석기 프로그램을 완성한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

3 월 19 일: 과제 명세서 수령
3 월 20 일: 과제 명세 상 요구사항 분석
3 월 21 일: C- 프로그래밍 언어 문법 분석 및 정규표현식 작성
3 월 22 일-24 일: C- 프로그래밍 언어 어휘분석기 작성
3 월 25 일: 테스트용 C- 프로그램 작성 및 어휘분석기 테스트
3 월 26-27 일: 보고서 작성

나. 개발 방법

1. 개발 환경

로컬 환경에서 개발과 테스트를 마친 뒤 학교에서 제공되는 cspro 서버에 파일을 업로드해 정상 작동을 다시 확인한다. 소프트웨어 버전에 다소 차이가 있으나, 각 프로그램의 업데이트 로그 등을 통해 버전에 따른 기능 차이가 본 프로젝트 수행에 영향을 주지 않음을 확인하고 진행한다.

로컬 환경: Ubuntu 18.0.4, gcc 7.3.0, flex 2.6.4
서버 환경: Ubuntu 16.0.4, gcc 5.4.0, flex 2.6.0

2. 개발 절차

(1) [1]에 주어진 TINY 컴파일러의 소스코드와 Lex 명세를 분석하여 작동 구조를 이해하고, 어휘분석기 작동에 필요한 부분과 TINY 프로그래밍 언어에 특화된 부분을 각각 파악한다.

(2) [1]에 명시된 C- 프로그래밍 언어의 어휘와 토큰을 파악하고 Lex 문법에 따른 정규표현식을 작성한다.
(3) TINY 컴파일러의 소스코드와 Lex 명세를 수정하여 C- 프로그래밍 언어의 어휘분석 기능을 수행할 수 있도록 수정한다.
(4) 수정한 결과물에서 어휘분석 기능과 무관한 부분을 배제하여 완성된 프로그램이 온전히 어휘분석기로만 작동하도록 수정한다.
(5) C- 언어 문법에 따라 테스트 프로그램을 작성하고 완성된 어휘분석기에 입력하여 결과를 확인한다. C- 언어 명세 상의 모든 토큰을 포함하는 테스트 프로그램을 적어도 하나 작성하여 이용한다.

IV. 연구 결과

1. 합성

프로그램을 구성하는 각 파일 간의 관계는 다음과 같다. "cmin.l"로 작성한 C- 언어의 어휘 및 토큰 명세를 Flex 툴로 처리하면 C 언어로 이를 결정적 유한 오토마타(Deterministic Finite Automata, DFA)로 구현해 처리할 수 있는 함수 "yylex()"가 "lex.yy.c"에 작성된다. 이 함수는 "cmin.l"에 작성된 토큰 스캔 함수 "getToken()"에 의해 호출되어 토큰을 처리하고 어휘분석을 실시한다. "getToken()"은 어휘분석의 결과를 "utils.h/c"가 제공하는 "printToken()" 함수를 이용해 표준출력에 표시하는 기능까지 수행한다.

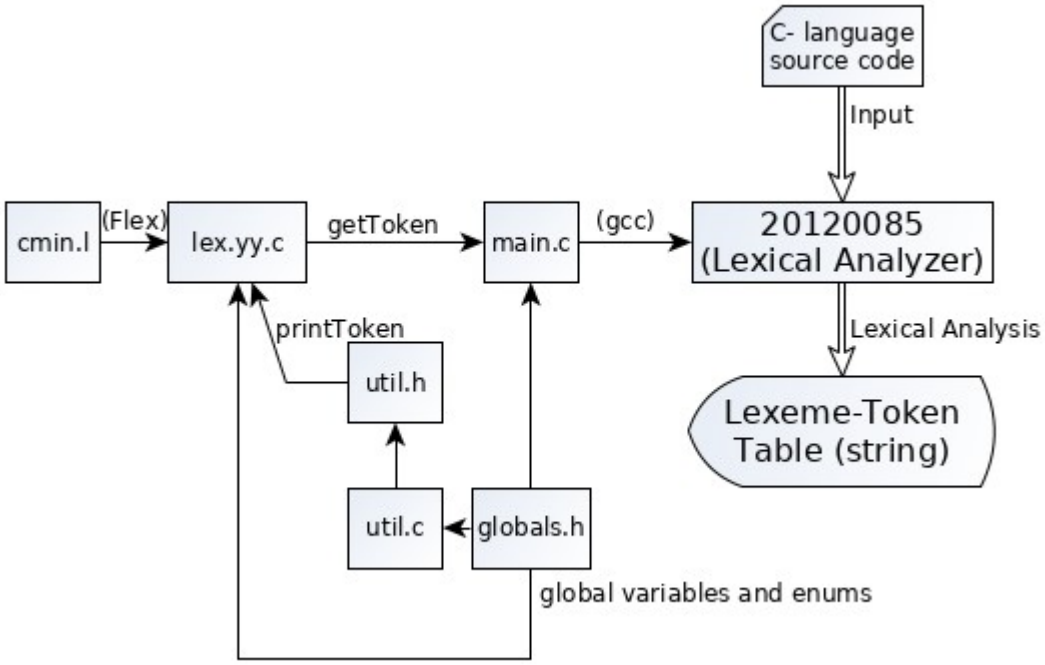

도표 1: 프로그램 구성 요소 관계도

과제 제출물은 이 중 "cmin.l", "main.c", "util.c", "util.h", "globals.h" 등 프로그램 소스코드 다섯 개와 이를 컴파일하는 규칙을 명시한 "Makefile" 등이다. 여기에 더해 두 개의 테스트 프로그램("test1.c", "test2.c")과 본 보고서 등 총 9 개 파일이 압축 파일 형태로 제출된다.

make 툴로 소스코드를 컴파일하여 얻은 실행파일 "20120085"는 C- 프로그래밍 언어의 소스코드에 대한 어휘분석기 프로그램이다. 이 프로그램은 C- 언어 소스코드를 텍스트 파일로 입력받아, 이에 대한 어휘분석 결과를 어휘-토큰 테이블(Lexeme-Token Table) 형태로 표준출력에 표시한다. 실행파일의 사용 방법은 다음과 같다. 어휘분석을 시행할 C- 언어 소스코드의 경로를 실행파일의 인자로 전달하면 된다.

```
$ ./20120085 sample/test1.c
```

2. 분석

C- 프로그래밍 언어 명세를 분석하여, 본 어휘분석기가 처리해야 할 토큰을 다음과 같은 정규표현식으로 정리한다. 정규표현식은 Lex 에서 인식하는 문법에 따른다.

* Previously defined names

Name	Regular expression	Name	Regular expression
digit	[0-9]	identifier	{letter}+
number	{digit}+	newline	\n
letter	[a-zA-Z]	whitespace	[\t]+

* Tokens

Token	Regular expression	Token	Regular expression
(Reserved words)		(Special Symbols)	
ELSE	"else"	>=	">="
IF	"if"	==	"=="
INT	"int"	!=	"!="
RETURN	"return"	=	"="
VOID	"void"	;	";"
WHILE	"while"	,	","
(Special symbols)		(	"("
+	"+"	)	")"
-	"-"	[	"["
*	"*"	]	"]"
/	"/"	{	"{"
<	"<"	}	"}"
<=	"<="	(Other tokens)	
>	">"	NUM	number
/*, */	(아래 본문에 설명)	ID	identifier

주석은 시작 조건(start condition)을 이용해 처리할 수 있다. 시작 조건은 Flex 툴의 공식 매뉴얼인 [2]에 제시된 문법으로, 해당 매뉴얼에서도 시작 조건을 사용하는 예시로 주석 처리를 들고 있다. 다음과 같은 방식으로 주석문을 처리할 수 있다.

주석을 시작하는 기호인 "/*"을 만나면 유한 오토마타가 "COMMENT" 상태에 진입한다. 이 상태에서는 오직 "*/"를 만날 때에만 원래 상태("INITIAL")로 돌아갈 수 있다. 그 사이에 식별되는 모든 비개행 문자는 무시되며, 개행문자가 식별되면 "yylineno"를 증가시킨다. "COMMENT" 상태에 있을 때 파일 끝(End Of File, EOF)에 이르면 에러로 인식하고 해당하는 토큰을 반환한다. 주석 기호는 C- 언어 명세 상 정의된 특수기호이기는 하지만, 토큰을 발생시키지 않으며 주석의 내용과 함께 그대로 소모되어 무시된다. 이는 다음과 같은 Lex 표현식으로 나타낼 수 있다.

```
%x COMMENT
%%
"/*"                    {BEGIN(COMMENT);}
<COMMENT>.              {/* eat up comment body */}
<COMMENT>{newline}      {/* yylineno automatically increments */}
<COMMENT><<EOF>>        {
                          BEGIN(INITIAL);
```

```
                          strncpy(yytext,"Comment Error", MAXTOKENLEN);
                          return ERROR; }
<COMMENT>"*/"           {BEGIN(INITIAL);}
%%
```

이 Lex 표현식은 다음과 같이 다양한 형태의 올바르거나 잘못된 주석문을 정상적으로 처리한다.

```
(1)
/*   comment   */ : no error
  "/*에서 "COMMENT" 상태로 진입한다.
  "  comment  "를 소모한다.
  "*/"에서 "INITIAL" 상태로 진입한다.
  정상적으로 주석문에 해당하는 문자열을 모두 소모한다.

(2)
/*   comment
     comment
                */ : no error
  "/*에서 "COMMENT" 상태로 진입한다.
  "  comment  \n    comment          "를 소모한다.
  "*/"에서 "INITIAL" 상태로 진입한다.
  정상적으로 주석문에 해당하는 문자열을 모두 소모한다.

(3)
/*   comment      : error
  "/*에서 "COMMENT" 상태로 진입한다.
  "  comment  "를 소모한다.
  "*/" 기호를 만나지 못해 "COMMENT" 상태에 머무른다.
  이 상태로 뒤이어 나오는 모든 문자열을 소모하다가 마침내 파일끝(EOF)에 이르면 에러가 발생한다.

(4)
/*   comment   /* : error
  "/*에서 "COMMENT" 상태로 진입한다.
  "  comment  /*"를 소모한다.
  "*/" 기호를 만나지 못해 "COMMENT" 상태에 머무른다.
  이 상태로 뒤이어 나오는 모든 문자열을 소모하다가 마침내 파일끝(EOF)에 이르면 에러가 발생한다.
```

3. 제작

* cmin.l

각 어휘를 인식하기 위한 정규표현식이 Lex 문법에 따라 작성되었다. Flex 가 기본으로 생성하는 함수 중 사용하지 않을 것을 배제하기 위해 "%option noyywrap noinput nounput"을 설정한다. 뼈대로 삼은 TINY 컴파일러는 소스코드의 줄 번호를 "lineno" 변수로 직접 관리하였으나, 이를 사용자 코드로 직접 작성하기보다는 Flex 내장 기능에 맡기는 것이 안전하다고 판단하여 "%option yylineno"를 설정하였다.

숫자(digit), 수(number), 글자(letter), 식별자(identifier), 개행문자(newline), 공백문자(whitespace) 등의 정의는 TINY 언어와 C- 언어가 동일하므로 기존의 정규표현식을 그대로 이용한다.

C- 언어의 6 개 예약어와 주석 기호를 제외한 19 개 특수 기호, 그리고 정규표현식으로 정의된 수와 식별자는 식별되는 대로 바로 토큰으로 인식하면 된다. 개행문자와 공백문자는 식별되는 대로 아무 동작 없이 다음 문자로 넘어갈 수 있다. 앞서 "yylineno" 옵션을 설정하였으므로 개행문자가 식별되면 자동으로 소스코드의 줄 번호가 증가한다.

주석은 앞서 "분석"에서 제안한 대로, 시작 조건을 이용해 주석 기호 사이의 주석문을 올바르게 처리한다.

"cmin.l"에서는 "yylex()" 함수를 호출해 앞서 정의한대로 토큰을 식별한 뒤 "printToken()"을 호출해 그 결과를 표준출력으로 표시하는 함수인 "getToken()" 또한 정의돼있다. 과제 명세에 지시된 형식대로 출력문을 수정하기만 하면 이 함수는 TINY 컴파일러에서 사용하던 코드를 그대로 사용할 수 있다.

* main.c

이 소스코드는 원래 TINY 컴파일러의 주된 뼈대가 되는 부분으로, 어휘분석 뿐만 아니라 그 이후의 단계까지 진행할 수 있는 코드가 작성되어있다. 따라서 본 프로젝트의 범위를 초과하는 부분을 코드에서 배제하기 위해, 전처리문으로 "NO_PARSE, NO_ANALYZE, NO_CODE"를 설정해 오직 어휘분석만 시행하도록 한다. 이 단계에서는 "scan.h"를 사용하지 않기 때문에 해당 include 문은 삭제하며, 출력문의 형식을 과제 명세의 지시대로 수정한다.

* util.h/c

본 프로젝트의 범위에서는 "util.h/c"의 함수 중 "printToken()"만 사용한다. 이 함수는 식별된 토큰과 어휘를 화면에 출력하는 함수이다. "cmin.l"에서 정의한 각 토큰에 대해 적절한 출력문을 작성한다. 특기할 점은 모든 예약어는 소문자로 인식된 어휘 문자열을 대문자로 변환하면 곧 토큰 문자열이며, 주석 기호를 제외한 모든 특수기호는 어휘 문자열과 토큰 문자열이 동일하다는 점이다. 이를 이용하면 다음과 같은 방식으로 6 개 예약어를 한 번에 처리할 수 있고, 19 개 특수기호도 한 번에 처리가 된다.

```
case ELSE: case IF: (중략) case WHILE: (6 개 예약어)
    식별된 어휘 문자열을 대문자로 변환하여 출력; (토큰에 해당)
    식별된 어휘 문자열을 그대로 출력;          (어휘에 해당)
    break;
case PLUS: case MINUS: (중략) case RBRACE: (19 개 특수기호):
    식별된 어휘 문자열을 그대로 출력; (토큰에 해당)
    식별된 어휘 문자열을 그대로 출력; (어휘에 해당)
    break;
```

수와 식별자는 토큰이 [1]에 각각 "NUM"과 "ID"로 정의되어 있으므로, 해당 토큰이 식별되면 각 문자열을 토큰으로 출력하고, 식별된 어휘 문자열을 그대로 어휘로 출력하면 된다.

* globals.h

프로그램 전반에 사용되는 전역 변수, 전처리문, 열거형 등이 정의된다. "MAXRESERVED"는 C- 언어 상 예약어의 수인 6 으로 설정하며, "enum TokenType"은 파일끝 토큰("ENDFILE")과 에러 토큰("ERROR")에 더해 "cmin.l"에서 열거한 토큰을 그대로 나열한다.

여기에 더해, 본래 "scan.h"에 정의되었던 일부 전처리문("MAXTOKENLEN"), 전역변수("tokenString"), 함수형("getToken") 등을 본 소스코드로 옮긴다. 이는 본 과제의 명세 상 "scan.h"를 제출하지 않는 상황에서, "cmin.l"과 "util.c" 등 일부 소스코드가 "scan.h"에 정의된 내용을 사용하기 때문이다. 해당하는 부분을 "globals.h"로 옮기면 링크 에러 없이 정상적으로 컴파일을 할 수 있다.

* Makefile

본 프로젝트에서 컴파일 되어야 할 파일은 다음과 같다. 우선 flex 툴을 이용해 "cmin.l"이 컴파일 되어 C 언어 소스코드인 "lex.yy.c"가 생성되어야 하고, gcc 컴파일러로 "main.c", "util.c", "lex.yy.c"가 컴파일 및 링크되어 실행파일 "20120085"가 생성되어야 한다. gcc 로 컴파일을 할 때에는 각종 버그와 에러를 미연에 방지하기 위해 "-Wall" 옵션을 적용하고, 최선의 성능을 위해 "-O3" 옵션을 적용한다.

정리하면 "make" 커맨드를 실행했을 때 수행되어야 하는 쉘 커맨드는 다음과 같다. 이를 수행할 수 있도록 "Makefile"을 작성한다.

```
flex cmin.l
gcc -Wall -O3   -c -o main.o main.c
gcc -Wall -O3   -c -o util.o util.c
gcc -Wall -O3   -c -o lex.yy.o lex.yy.c
gcc -o 20120085 main.o util.o lex.yy.o
```

4. 시험

우선 과제 명세와 함께 수령한 토큰 테이블 출력 예시 상 테스트 프로그램에 대해 어휘분석을 실시했다. 다음과 같이 이상 없이 어휘분석이 실행되어, 예시로 주어진 결과와 동일한 출력을 확인했다. 본 프로그램은 "for"문과 증감연산자 "++" 등을 사용하므로 C- 프로그래밍 언어의 올바른 소스코드는 아니지만, 어휘분석 단계에서는 이를 파악할 수 없다.

```
$ ./20120085 test1.c
      line number token       lexeme
-----------------------------------------------------
      2           INT         int
      2           ID          arr
      2           [           [
      2           NUM         11111
      2           ]           ]
      2           ;           ;
(후략)
```

다음으로 주석 처리를 확인하기 위해 토큰 테이블 출력 예시로 주어진 테스트 프로그램에 대해 어휘분석을 실시했다. 주석문 내에 작성된 문자열은 모두 정상적으로 무시되어 어휘분석이 이루어지지 않는다. 주석이 열린 채로 파일끝을 만나 에러를 확인하는 것까지 정상적으로 이루어진다.

```
$ ./20120085 test2.c
      line number token       lexeme
-----------------------------------------------------
      11          ERROR       Comment Error
      11          EOF
```

마지막으로 모든 토큰이 정상적으로 인식되는 것을 확인하기 위해, 29 개 토큰을 모두 이용하는 프로그램을 작성하여 테스트했다. 유의미한 기능을 수행하는 프로그램은 아니며 가능한 모든 토큰을 적어도 한 차례씩 사용하는 것을 목적으로 작성된 소스코드이다. 어휘분석 결과 주석문을 올바르게 처리하고 주석 기호를 제외한 27 종류의 토큰이 정상적으로 출력되는 것을 확인하였다.

```
int mod(int x, int y) {
    /* return x mod y */
    return x-x/y*y;
}
void main(void) {
    int i, arr[100];
    i = 0;
    while(i < 10) {
        if(mod(i,2) == 1) {
            output(i);
        }
        else {
            output(0);
        }
        i = i + 1;
    }
    i = 0;
    while(i <= 99) {
        if(mod(i,5) > 3) {
            arr[i] = i;
        }
        if(mod(i,5) >= 4) {
            output(i);
        }
    }
}
```

5. 평가

본 프로젝트는 [1]에 명세된 C- 프로그래밍 언어의 컴파일러를 제작하는 첫 단계로 어휘분석기를 구현하였다. C- 언어의 어휘 규칙을 정규표현식으로 나타내어 어휘 토큰을 처리할 수 있는 결정적 유한 오토마타를 구성하였고, 이를 이용해 소스코드 문자열을 처리하여 분석된 결과를 표준출력에 표시하는 프로그램을 완성하였다.

다양한 테스트 프로그램을 통해 프로그램에 요구되는 기능인 어휘분석을 성공적으로 수행함을 확인하였다. 이 과정에서 어휘분석 단계에서 처리할 수 있는 오류인 "주석문 내 파일 끝(End Of File Inside Comment)" 오류를 처리하였다.

기존에 완성된 TINY 프로그래밍 언어의 컴파일러 소스코드를 재사용하면서 불필요한 작업을 최소화하는 동시에, 기존 TINY 언어에 특화되어 C- 언어와 무관한 부분을 모두 제거하여 안정성을 획득하였다.

V. 참고 문헌

[1] Kenneth C. Louden. 1997. *Compiler Construction: Principles and Practice*. PWS Publishing Co. Boston, MA, USA. pp. 491-544.

[2] The Flex Project. 2012. “10 Start Conditions”. *Lexical Analysis With Flex, for Flex 2.6.2.* Retrieved March 26, 2019 from https://westes.github.io/flex/manual/Start-Conditions.html#Start-Conditions