Design and Development of Compiler for C- Language

2. Design and Implementation of LALR Parser

과목명 : [CSE4120] 기초 컴파일러 구성

담당교수 : 서강대학교 컴퓨터공학과 정성원

개발자 : 엄태경

개발기간 : 2019. 4. 16. ~ 2019. 5. 3.

현 개발 단계 결과 보고서

프로젝트 제목: Design and Development of Compiler for C-Language:
Phase 2: Design and Implementation of LALR Parser
제출일: 2019. 5. 3.
개발자: 엄태경

I. 개발 목표

본 프로젝트는 C- 프로그래밍 언어를 처리할 수 있는 컴파일러 구현의 두번째 단계로, LALR 문법분석기 (LALR parser)를 개발해 문법분석(syntactic analysis) 기능을 구현하는 것을 목표로 한다. 이 단계까지 구현된 컴파일러에 소스코드를 입력하면 추상문법트리(Abstract Syntax Tree)가 구성된다.

II. 개발 범위 및 내용목표

가. 개발 범위

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

나. 개발 내용

C- 언어의 문법은 29 개의 BNF 규칙으로 이루어진다. [1]에 명시된 각 규칙 분석하여 추상문법트리를 구성하는데 필요한 요소를 파악하고, C 언어와 Yacc 문법에 따라 Yacc 파일을 구현한다. GNU/bison 툴을 이용해 이를 C 언어로 변환하여 나머지 C 언어 소스코드와 함께 컴파일해 LALR 문법분석기 기능을 완성한다.

III. 추진 일정 및 개발 방법

가. 추진 일정

4 월 16 일: 과제 명세서 수령 및 요구사항 분석
4 월 17-18 일: C- 프로그래밍 언어 문법 규칙 분석
4 월 19 일-22 일: C- 프로그래밍 언어 문법분석기 작성
4 월 23-30 일: 테스트용 C- 프로그램 작성 및 문법분석기 테스트, 디버깅
5 월 1-3 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, bison 3.0.4
서버 환경: Ubuntu 16.0.4, gcc 5.4.0, flex 2.6.0, bison 3.0.4

2. 개발 절차

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

(2) [1]에 명시된 C- 프로그래밍 언어의 문법규칙을 파악하고 Yacc 문법에 따라 추상문법트리를 구성할 수 있도록 규칙을 정리한다.
(3) 1 단계에서 구현한 C- 프로그래밍 언어 어휘분석기에 더하여, TINY 컴파일러의 소스코드와 Yacc 파일을 수정하고 추가해 C- 프로그래밍 언어의 문법분석 기능을 구현한다.
(4) C- 프로그래밍 언어 문법에 따라 테스트 프로그램을 작성하고 완성된 문법분석기에 입력하여 결과를 확인한다. 추상문법트리가 올바르게 구성되어 적절하게 출력되는 것을 확인한다.

IV. 연구 결과

1. 합성

프로그램을 구성하는 각 파일 간의 관계는 다음과 같다. "cmin.y"와 이를 Flex 툴로 처리하여 생성한 "lex.yy.c"는 프로젝트 1 단계에서 구현한 내용으로, 여기서 설명은 생략한다.
"cm.y"로 작성한 C- 프로그래밍 언어의 문법 규칙을 GNU bison 툴로 처리하면 C 언어로 이를 결정적 유한 오토마타(Deterministic Finite Automata, DFA)로 구현해 처리할 수 있는 함수 "yyparse()"가 "y.tab.c/h"에 작성된다. 이 함수는 "cm.y"에 작성된 토큰 스캔 함수 "parse()"에 의해 호출되어 문법분석을 실시하고 그 결과로 추상문법트리를 생성한다. 이렇게 생성된 추상문법트리는 "util.c/h"에 구현된 "printTree()"에 입력되어 표준출력으로 출력된다.

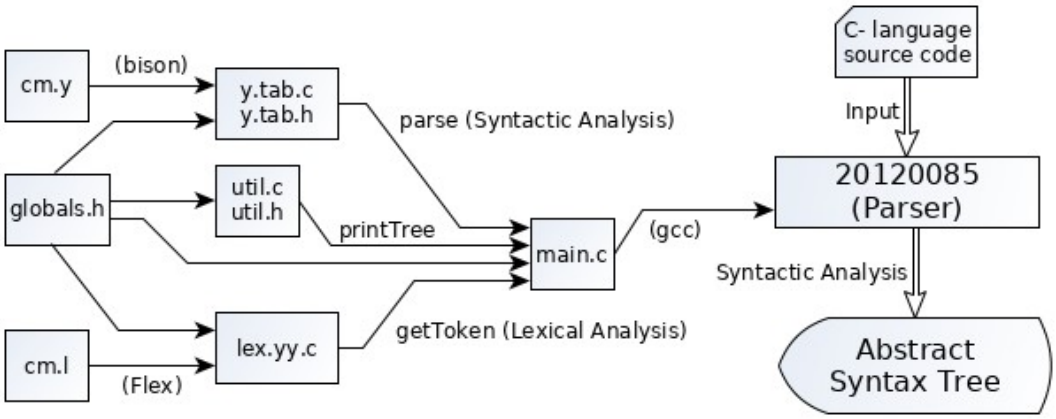

도표 1: 프로그램 구성 요소 관계도

과제 제출물은 이 중 "cm.l" "cm.y", "main.c", "util.c", "util.h", "globals.h" 등 프로그램 소스코드 일곱 개와 이를 컴파일하는 규칙을 명시한 "Makefile" 등이다.

make 툴로 소스코드를 컴파일하여 얻은 실행파일 "20120085"는 C- 프로그래밍 언어의 소스코드에 대한 문법분석기 프로그램이다. 이 프로그램은 C- 언어 소스코드를 텍스트 파일로 입력받아, 이에 대한 문법분석 결과를 추상문법트리(Abstract Syntax Tree) 형태로 표준출력에 표시한다. 실행파일의 사용 방법은 다음과 같다. 어휘분석을 시행할 C- 프로그래밍 언어 소스코드의 경로를 실행파일의 인자로 전달하면 된다.

```
$ ./20120085 test.c
```

2. 분석 및 제작

*** 충돌 해소: 조건문 현수 문제에서의 이동-환원 충돌**

C- 프로그래밍 언어 규칙을 분석한 결과, BNF 로 주어진 문법은 이미 각 연산자의 우선순위 및 계산 방향을 포함하고 있어 따로 우선순위를 명시할 필요는 없다. 하지만 규칙 15 에 정의된 조건문 문법에는 전형적인 현수 문제("dangling else")가 있으며, 이런 상황에서 "else"는 항상 가장 가까운 "if"와 연결되어야 한다는 충돌 처리 규칙은 문법 외로 주어진다. 이 문제를 해결하기 위하여, 다음과 같이 처리하였다.

```
…
%right NOELSE ELSE
%%
…
```

```
selection_stmt      : IF LPAREN expression RPAREN statement %prec NOELSE
                        { … }
                    | IF LPAREN expression RPAREN statement ELSE statement
                        { … };

(주: IF, LPAREN, RPAREN, ELSE는 각각 문자열 "if", "(", ")", "else"에 해당하는 토큰.)
```

이것은 [2]에서 설명된 문맥의존 우선순위(Context-Dependent Precedence)를 이용한 것이다. 더미 터미널 기호 "NOELSE"와 실제 조건문에 사용되는 터미널 "ELSE"의 방향성을 오른쪽에서 왼쪽으로 지정함으로써, 반드시 두번째 규칙(이동)이 첫번째 규칙(환원)보다 우선하도록 강제한다. 이렇게 하면 "if-else" 형태의 조건문에서 "else"가 항상 바로 앞의 "if"와 연결되도록 문법분석이 이루어진다.

추상문법트리에서 표현해야 할 노드의 종류를 다음과 같이 정리한다.

```
* Node Types
```

대분류	소분류	설명
StmtK	CompoundK	복합문(여러 문의 나열)
	SelectionK	조건
	IterationK	반복
	ReturnK	반환
ExpK	AssignK	대입
	OpK	이진연산
	ConstK	상수
	VarK	배열 아닌 변수 이름
	ArrK	배열 이름
	CallK	호출하는 함수 이름
DeclK	VarDeclK	배열 아닌 변수 선언
	ArrDeclK	배열 선언
	FunDeclK	함수 선언
TypeKind	TypeGeneralK	타입 지정자
ParamKind	VarParamK	배열 아닌 변수인 파라미터
	ArrParamK	배열인 파라미터
	VoidParamK	함수에 파라미터가 없는 경우

이를 이용하면 다음과 같이 노드를 구조체로 표현할 수 있다.

```
* Node Structure
typedef struct treeNode
{
   struct treeNode *child[MAXCHILDREN];
   struct treeNode *sibling;
   int lineno;
   NodeKind nodekind;
   union {
      StmtKind stmt;
      ExpKind exp;
```

```
      DeclKind decl;
      TypeKind type;
      ParamKind param;
   } kind;
   union {
      TokenType op;        /* for operator */
      int val;             /* for constant */
      char *name;          /* for variable */
   } attr;
   ExpType type; /* for type checking of exps */
} TreeNode;
```

Yacc 파일에 구현된 각 문법 규칙은 주로 새로운 노드를 생성하고 속성을 지정하거나, 어떤 노드를 다른 노드의 형제로 연결한다. 새로운 노드를 생성하는 규칙은 다음과 같다. 각 노드를 생성하는 작업은 "util.c"에 정의된 "newStmtNode, newExpNode, newDeclNode, newTypeNode, newParamNode" 등 각 함수에서 메모리를 할당하고 각 속성을 초기화한다.

* Node Declaration

규칙	노드 생성 (Sematic Action)
number : NUM	상수 노드(ConstK)
var_declaration : type_specifier identifier SEMI	변수 선언 노드(VarDeclK)
var_declaration : type_specifier identifier LBRACKET number RBRACKET SEMI	배열 선언 노드(ArrDeclK)
type_specifier : INT | VOID	타입 노드(TypeGeneralK)
fun_declaration : type_specifier identifier LPAREN params RPAREN compound_stmt	함수 선언 노드(FunDeclK)
params: VOID	빈 파라미터 노드(VoidParamK)
param: type_specifier identifier	변수 파라미터 노드(VarParamK)
param: type_specifier identifier LBRACKET RBRACKET	배열 파라미터 노드(ArrParamK)
compound_stmt : LBRACE local_declarations statement_list RBRACE	복합문 노드 (CompoundK)
selection_stmt : IF LPAREN expression RPAREN statement %prec NOELSE | IF LPAREN expression RPAREN statement ELSE statement	조건문 노드 (SelectionK)
iteration_stmt : WHILE LPAREN expression RPAREN statement	반복문 노드(IterationK)
return_stmt : RETURN SEMI | RETURN expression SEMI	반환문 노드(ReturnK)
expression : var ASSIGN expression	대입문 노드(AssignK)
var : identifier	변수 노드(VarK)
var : identifier LBRACKET expression RBRACKET	배열 노드(ArrK)
relop: LTE | LT | GT | GTE | EQ | NEQ	이진연산 노드(OpK)
addop: PLUS | MINUS	
mulop: TIMES | OVER	
call: identifier LPAREN args RPAREN	호출 노드(CallK)

각 노드가 가질 수 있는 자녀 노드는 다음과 같다. 각 노드는 명시적인 자녀를 최대 3 개 가질 수 있다. "(들)"은 해당 자녀가 형제관계를 통해 길게 이어질 수 있음을 의미한다.

또, 앞서 제시한 노드 구조체의 구조에 따르면 각 노드는 {연산자 종류, 상수값, 이름(identifier)} 중 하나의 값을 속성으로 가질 수 있고, 수식은 타입 속성을 가진다. 노드 종류마다 가질 수 있는 속성을 함께 정리한다.

```
* Children Nodes and Attributes
```

노드 종류	자녀 1	자녀 2	자녀 3	속성
CompoundK	변수선언문(들)	명령문(들)		
SelectionK	조건	조건이 성립할 때 실행할 명령문(들)	조건이 성립하지 않을 때 실행할 명령문(들)	
IterationK	조건	조건이 성립할 때 반복할 명령문(들)		
ReturnK	반환할 값			
AssignK	대입될 변수	대입할 값		
OpK	피연산자 1	피연산자 2		연산자 종류(attr.op)
ConstK	–			값(attr.var)
VarK	–			이름(attr.name)
ArrK	배열 인덱스			이름(attr.name)
CallK	호출 인자(들)			이름(attr.name)
VarDeclK	타입			이름(attr.name)
ArrDeclK	타입	크기		이름(attr.name)
FunDeclK	타입	파라미터(들)	명령문(들)	이름(attr.name)
TypeGeneralK	–			타입
VarParamK	타입			이름(attr.name)
ArrParamK	타입			이름(attr.name)
VoidParamK	–			

구성된 트리를 출력하는 것은 "util.c"에 정의된 "printTree" 함수이다. 이 함수는 트리 상의 노드 하나를 가리키는 포인터를 인자로 받아 다음과 같은 절차로 재귀적으로 추상문법트리를 출력한다.

1) 들여쓰기 단계를 높여 출력되는 노드의 높이가 한 단계 낮아지게 한다.
2) 노드의 대분류와 소분류에 따라서, 노드의 종류와 속성을 적절히 출력한다.
3) 모든 자녀 노드를 하나씩 재귀적으로 출력한다.
4) 형제 노드로 이동한 뒤 2)로 이동한다. 형제노드가 더 이상 없으면 5)로 이동한다.
5) 들여쓰기 단계를 낮추어 출력되는 노드의 높이가 원상복구되게 한다.

즉, "printTree" 함수는 다음과 같은 구조로 이루어져있다. 각 대분류와 소분류의 설명은 앞서 설명한 바를 그대로 간단하게 출력하며, 위 표에 표시한대로 "속성"이 있는 노드는 속성을 함께 출력한다. 특기할 것은 연산자 노드(OpK)의 속성인 연산자 종류(attr.op)를 출력할 때에는, 저장된 나열형(enum) 값을 그대로 출력하는 것이 아니라 해당하는 실제 연산자 어휘를 출력한다. 저장된 나열형을 실제 연산자 어휘로 변환하는 작업은 함수 "getOp"에서 수행한다.

```
  INDENT; /* 들여쓰기 단계 증가 */
  int sibling = 0; /* 형제 노드의 수를 세는 변수 */
  while (tree != NULL)
  {
    if(!sibling && tree->sibling != NULL) {
      printSpaces();
      fprintf(listing, "(\n");
      INDENT;
    } /* 형제노드의 나열이 시작되기 전에 괄호 '('를 출력하고 들여쓰기 증가 */
```

```
    printSpaces();  /* 적절히 들여쓰기를 하여 노드의 높이를 표현 */
    if (tree->nodekind == (대분류))
    {
      switch (tree->kind.(대분류))
      {
      case (소분류):
        fprintf(listing, "소분류의 내용과 속성을 적절히 출력\n");
        break;
      /* 모든 소분류에 대해 case 문으로 구현 */
      default:
        fprintf(listing, "알 수 없는 소분류; 오류 출력\n");
        break;
      }
    }
    /* 모든 대분류에 대해 else if 문으로 구현 */
    else
      fprintf(listing, "알 수 없는 대분류; 오류 출력\n");
    for (i = 0; i < MAXCHILDREN; i++)
      printTree(tree→child[i]); /* 모든 자녀 노드 출력 */
    tree = tree→sibling; /* 형제 노드로 이동 */
    if(sibling && tree == NULL){
      UNINDENT;
      printSpaces();
      fprintf(listing, ")\n");
    } /* 형제노드의 나열이 끝나면 들여쓰기 단계를 낮추고 괄호 ')' 출력 */
    if(tree!=NULL) {
      sibling++;
    } /* 형제노드의 수를 셈 */
  }
  UNINDENT; /* 들여쓰기 원상복구 */
```

각 노드 대분류 및 소분류 별 구체적인 구현은 특별한 것 없이 반복적이고 일관적인 구조이므로 생략한다.

4. 시험

우선 과제 명세와 함께 예시로 주어진 gcd.c 소스코드를 입력하여 다음과 같이 추상문법트리가 적절히 구성되고 출력되는 것을 확인한다. 괄호와 함께 함수 선언문, 함수 파라미터, 함수 호출 인자, 변수 선언문, 복합문 등 형제 노드의 나열이 적절하게 표시됨을 알 수 있다.

```
$ ./20120085 gcd.c
Syntax tree:
  (
    Function Declaration: gcd
      Type: int
      (
        Parameter (variable): u
          Type: int
        Parameter (variable): v
          Type: int
      )
      Compound Statement
        Selection Statement
          Op: ==
            Variable: v
            Const: 0
          Return Statement
            Variable: u
          Return Statement
            Calling: gcd
```

```
            (
              Variable: v
              Op: -
                Variable: u
                Op: *
                  Op: /
                    Variable: u
                    Variable: v
                  Variable: v
            )
   Function Declaration: main
     Type: void
     Parameter: void
     Compound Statement
       (
         Variable Declaration: x
           Type: int
         Variable Declaration: y
           Type: int
       )
       (
         Assign Expression
           Variable: x
           Calling: input
         Assign Expression
           Variable: y
           Calling: input
         Calling: output
           Calling: gcd
             (
               Variable: x
               Variable: y
             )
       )
 )
```

다음으로 문법 오류가 있는 소스코드를 입력하여 오류가 발생하는 줄 번호를 올바르게 출력함을 확인한다.

```
$ ./20120085 test2.c
Syntax error at line 3: syntax error
Error found while parsing token:                ID              num
```

5. 평가

본 프로젝트는 [1]에 명세된 C- 프로그래밍 언어의 컴파일러를 제작하는 두번째 단계로 문법분석기를 구현하였다. C- 언어의 문법 규칙을 적절히 Yacc 문법에 따라 정리하여 추상문법트리를 적절하게 구성하였다. 이를 이용해 소스코드를 분석하여 생성한 추상문법트리를 표준출력에 표시하는 프로그램을 완성하였다. 다양한 테스트 프로그램을 통해 프로그램에 요구되는 기능인 문법분석을 성공적으로 수행함을 확인하였다.

V. 참고 문헌

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

[2] Charles Donnelly and Richard Stallman. 2019. *Bison 3.3.* "5.4 Context-Dependent Precedence". Retrieved May 3, 2019 from https://www.gnu.org/software/bison/manual/html_node/Contextual-Precedence.html