Design and Development of Compiler for C- Language

Syntax Parser

과목명: 기초 컴파일러 구성 : [CSE4120]

담당교수: 서강대학교 컴퓨터공학과 정성원

개발자: 20151575 윤제형

개발기간: 2019. 3. 24 - 2019. 3. 26

프로젝트 제목: Design and Development of Compiler for C-Language:
Phase 1: Design and Implementation of LALR Parser
제출일: 2019.5.2
개발자: 윤제형 (20151575)

1. 개발 목표

- C- language를 이해하는 lexical analyzer를 전 프로젝트에서 개발했었다. 이를 기반으로 생성된 Token Stream 을 parsing 하기 위하여 Context Free Grammar (CFG) 를 이해하고 Abstract Syntax Tree (AST) 를 생성하는 LALR(1) Parser 를 개발한다. 하지만 이는 LR(1) 을 축약 한 것이므로 Reduce/Reduce conflict가 날 수 있음에 유의한다.

2. 개발 범위 및 내용

a. 개발 범위

i. 책에서 제시되는 C-의 Grammar rule를 코드로 구현 (cm.y)

ii. Grammar rule의 statement 에 대한 semantic action들을 제작 (cm.y)

iii. Grammar rule에 대한 자료구조를 정의 하고 이를 handling 하는 method들과 이를 print 하는 함수를 제작

b. 개발 내용

- c- 의 문법을 설명하고 있는 규칙인 grammar rule를 작성 하기 위해 BNF 문법을 따라 cm.y 의 기본 골자 구조를 생성한다. grammar rule를 분석 한 뒤 이를 tree 로 만들 때 필요 한 node 들의 type을 정의하고 자료구조를 생성한다. 이후 tree 를 생성하는 semantic action들을 생성하고 이를 print하는 함수를 제작한다.
- 책의 tiny parser에서는 id, type 등이 출현 할 때 이를 저장하여 놓고 semantic action에서 이를 활용한다. 하지만 이는 compound stmt 나 다른 nested된 stmt 들에 의해 덮어 씌어 질 수 있으므로 이를 해결하기 위해 간단한 stack을 구현한다.

3. 추진 일정 및 개발 방법

a. 추진 일정

- 2019.4.28 ~ 2019.5.02 : 프로그램 분석 및 제작
- 2019.5.02 ~ 2019.5.03 : 프로그램 보고서 작성

b. 개발 방법

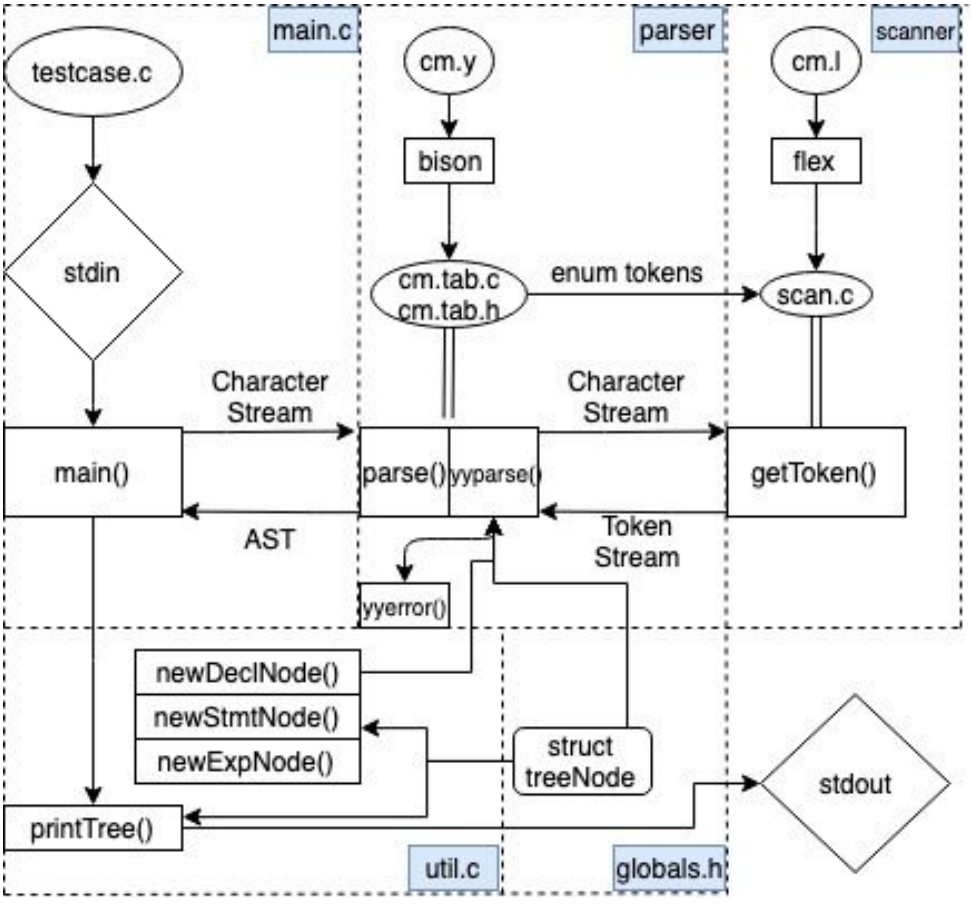
- 제시된 tiny.y를 참고하여 semantic action을 생성한다.
- C- 문법은 두가지로 나누어진다고 생각한다. 하나는 선언문(declare)이고 다른 하나는 지시문(statement) 이다.
- 선언 문은 ,함수 선언 문 ,매개변수 선언 문, 변수 선언 문, 배열 선언 문 으로 나눠져 있다.

- 지시문은 표현(expression) 문, 결합({}) 문, 분기(if) 문, 반복(while) 문, 반환 {return} 문, 호출(Call) 문 으로 이루어져 있다. 이중에 표현 문은 선언문에 포함되지만 따로 추출하여 새로 선언한다.
- 표현문은 실행자, 상수, 변수 로 이루어져 있다.
- 선언문에 관여 하는 type은 void, int, array로 이루어져 있다.
- 위와 같은 kind들을 고려하여 자료구조를 생성하고 이에 따라 tree를 print하는 함수를 구현한다. 이를 구현 할 때 공백을 통해 tree의 계층을 표현한다.

4. 연구 결과

a. 합성 내용

소프트 웨어 구성도



Software의 흐름: 이번 프로젝트의 목적은 C- 의 Grammar를 bison문법에 맞게 제작, 이용하여 LALR Parser를 생성하고 lexical analyzer를 통해 생성한 Token Stream을 적절한 Abstract Syntax Tree로 바꾸어 출력 하는 것이다. 이를 토대로 parser 와 scanner를 bison이나 flex를 통해 생성한다. main에서 parser를 호출하면 parser는 scanner와 소통하여 주어진 input을 grammar대로 parsing해 나간다. 이 과정에서 util.c와 globals.h 에 존재하는 tree node와 이와 관련된 method를 활용하고 error가 발생하면 이는 해당 syntax로는 parsing할 수 없으므로 yyerror 호출하여 syntax error를 발생 시킨다. 이후 parsing된 결과를 printTree로 보내여 알맞게 출력 시킨다.

Bison: Grammar rule에 해당하는 LALR Parser를 제작하여 준다. -d를 통해 헤더파일을 생성할 수 있으며 -dv 옵션을 통해 헤더 파일도 생성하고 statement 들과 item들을 적은 테이블을 출력 시킬 수 있다. (bison -dv cm.y)

사용 방법: ./20151575 testcase1.c

b. 분석 내용

- **Tree Node Data Structure:** tiny에서 제시된 구조를 많이 차용하여 제작하였다. nested된 node들을 저장할 child와 같은 계층의 node를 저장할 sibling을 선언하였고 3가지 node의 분류를 구별 할 flag와 분류된 node의 종류를 표현할 flag도 선언하였다. 각 node의 종류에 맞게 사용될 data를 선언 하였고 이는 name, array size, operator, value, type 등을 포함 하였다.

- **Semantic Action:** nested된 node는 child 에 저장 하였고 자신의 계층과 같은 node들은 sibiling에 저장하였다. 각 정보들을 알맞게 저장하였는데 type이나 id name등은 overwrite되기 때문에 stack 을 구현하여 해당 token이 나온 경우 바로 이 정보를 push 하고 추후 reduce 시에 이를 pop하는 방법으로 안전하게 접근 하였다.

- **Stack:** operator, id name, type 등을 저장하기 위한 stack 을 구현하였다.

- **Conflict:** 현재 Grammar 의 LALR Parser 에서는 1개의 conflict만이 존재한다. 이는 select stmt 에서
  if (expression) statement .
  if (expression) statement . else statement
  의 state를 가진 item에서 lookahead가 else일때 이를 shift를 할 지 reduce를 할 지 conflict가 난다. 이를 해결하기 위하여 우선순위(Precedence) 를 활용하여 ‘) ‘보다 else의 우선순위를 높여 shift를 지시하여 이 충돌 문제를 해결 하였다.

- **Print Tree:** 기본적으로 여백을 통해 자식 관계와 형제 관계를 구분한다. 선언문에서는 선언된 함수나 변수를 출력할때 자신의 이름과 구분을 출력하고 printType을 통해 indent하여 자신의 type을 출력한다. 매개 변수의 경우 없을 경우 void를 출력한다. 이후 자식 노드들의 출력으로 재귀 문을 호출한다.
  지시문 출력의 경우 자신의 특징을 출력하고 바로 자식들의 출력으로 재귀 문을 호출하는데 여기서 자식들은 조건문이나 매겨변수 Nested 된 명령어 들이다.
  표현문에서는 대부분 주어진 부분을 출력하고 자식을 재귀 호출로 출력하는데 ID의 경우 만약 자식이 존재한다면 이는 ID가 array인 것 이므로 이후 등장하는 expression은 index임을 알리기 위해 “index expression”을 출력한다.

c. 제작 내용

globals.h

```
typedef enum {DeclK,StmtK,ExpK} NodeKind;
typedef enum {FunK,VarK,VarArrK,ParaK} DeclKind;
typedef enum {CompndK,SelcK, IterK, RetK, CallK} StmtKind;
typedef enum {OpK,ConstK, IdK} ExpKind;
typedef enum {Void, Integer, Array} Type;
```

- 분석 내용대로 3가지 종류의 node가 존재하고 표현문은 지시문에 표현되지만 따로 빼서 3가지 종류로 선언 하였다.

```
#define MAXCHILDREN 4

typedef struct treeNode{
    struct treeNode * child[MAXCHILDREN];
    struct treeNode * sibling;
    int lineno;

    NodeKind nodekind;
    union {
        DeclKind decl;
        StmtKind stmt;
        ExpKind exp;
    }kind;
    union {
        struct {
            char *name;
            int arr_size;
        }decl;
        TokenType op;
        int val;
    }attr;
    Type type;
} TreeNode;
```

- tree node의 구현으로 분석 내용대로 node의 분류, 종류를 나타내는 flag(nodekind, kind) 들이 존재하고 이 분류에 따라 사용하는 attr 이 존재한다.

util.c

```
void printTree( TreeNode * tree ){
    int i;
    INDENT;
    while(tree != NULL) {
        printSpaces();
        if (tree->nodekind == DeclK){
            switch(tree -> kind.decl){
                case FunK:
                    fprintf(listing,"Function: %s\n",tree->attr.decl.name);
                    printType(tree);
                    break;
                case VarK:
                case VarArrK:
                    fprintf(listing,"ID: %s\n",tree->attr.decl.name);
                    printType(tree);
                    break;
```

- printTree의 구현 내용중 일부이다. 보이는 데로 분류와 종류를 나누어 알맞게 공백을 기준으로 출력하였다.

```
TreeNode * newDeclNode(DeclKind);
/* Function newStmtNode creates a
 * node for syntax tree constructi
 */
TreeNode * newStmtNode(StmtKind);

/* Function newExpNode createsa ne
 * node for syntax tree constructi
 */
TreeNode * newExpNode(ExpKind);
```

- node에 종류에 맞게 flag를 setting 하여 생성해주는 함수를 작성하였다.

cm.y

```
static void push_op(TokenType op){
    StackOp[++top] = op;
}
static TokenType pop_op(){
    if(top >= 0)
        return StackOp[top--];
    return -1;
}
```

- 사용된 stack method중 일부 로 operator를 저장하고 반환해준다.

```
#define SAVEID do{\
    push_ID(copyString(prev));\
    savedLineNo = lineno;\
}while(0)\
```

```
var-declaration         :   type-specifier ID { SAVEID; } SEMI
                            {
                                $$ = newDeclNode(VarK);
                                $$ -> attr.decl.name = pop_ID();
                                $$ -> lineno = savedLineNo;
                                $$ -> type = pop_type();
                            }
```

- semantic action중의 일부로 overwrite문제를 해결하기 위하여 push, pop을 이용하여 정보를 저장하는 모습이다.

```
%nonassoc RPAREN
%nonassoc ELSE
```

- 우선순위를 통해 conflict 를 해결한 모습이다.

d. 시험 내용

- 정상적인 input

```
cse20151575@cspro9:~/Compiler/proj2$ cat testcase/errorfree.c
void main ( void )
{
    int x;
    x = 2;
    if(x == 1){
        x = input();
    }
}
cse20151575@cspro9:~/Compiler/proj2$ ./20151575 testcase/errorfree.c

C- COMPlLATION: testcase/errorfree.c

Syntax tree:
  Function: main
    Type: Void
    Parameter: VOID
    Compound statement
      ID: x
        Type: Int
      Op: =
        Id: x
        const: 2
      If
        Op: ==
          Id: x
          const: 1
        Compound statement
          Op: =
            Id: x
            Call procedure: input
```

- 비정상적인 input

```
cse20151575@cspro9:~/Compiler/proj2$ cat testcase/error.c
void main ( void )
{
    int x;
    x = 2;
    if(x == 1){
        x = input();
}
cse20151575@cspro9:~/Compiler/proj2$ ./20151575 testcase/error.c

C- COMPlLATION: testcase/error.c
Syntax error at line 8: syntax error
Current token: EOF

Syntax tree:
```

두개의 test가 모두 무리 없이 돌아가는 모습입니다.

e. 평가 내용

- 장점은 stack을 활용하여 overwrite문제를 깔끔하게 해결 했다는 것이다. 이문제를 해결 하기 위해서는 $i 를 활용하는 방법도 있는데 이는 pointer에 상수값을 저장하는 것이기 때문에 위험하고 warning이 발생한다. 이 때문에 stack을 사용하여 이문제를 해결하였다.
- 단점은 stack 을 linked list 가 아닌 array를 통해 구현했기 때문에 stack overflow가능성이 있다는 것이다. 현재 256개의 크기를 지정해 놨는데 이는 웬만한 code에서는 문제가 되지 않지만 오류를 발생시킬 위험성을 내포하고 있다.