REPORT

Parser Implementation with YACC

전공 : 소프트웨어전공
학번 : 2013011167
이름 : 진 용 석

1. 파서 구현 방법 및 주요 소스코드 설명

이전 과제에서 구현한, flex를 활용하는 Scanner 를 기반으로, YACC 를 통해 Parse Tree 를 생성하도록 기능을 확장하였습니다.

현재 문법 상에서 기존 방식대로는 ID 또는 NUM 토큰의 원본 문자열을 가져오는 데에 어려움이 있었습니다. 따라서 문자열을 저장하는 Stack 을 구현하여, ID 또는 NUM 을 입력받으면 하나씩 Push 이를 저장할 때 Pop 하여 문자열을 불러왔습니다. Stack 의 top 은 전역 변수로 두어 어느 파일에서나 접근할 수 있도록 처리하였습니다.

```c
/**
 * push a token string to stack.
 */
StackNode *PushStack(StackNode *top, char *tokenString)
{
    StackNode *newNode;

    newNode = (StackNode *)malloc(sizeof(StackNode));
    strncpy(newNode->token, tokenString, MAXTOKENLEN + 1);
    newNode->next = top;

    return newNode;
}

/**
 * pop a token string from stack.
 * this element should be freed after use.
 */
StackNode *PopStack(StackNode **top)
{
    StackNode *retNode;

    /* error handling */
    if (*top == NULL)
    {
        printf("error occured : pop from empty stack\n");
        return NULL;
    }

    retNode = *top;          /* pop current top node from stack */
    *top = retNode->next;    /* set current top node to the next node */

    return retNode;
}
```

C-Minus 가 사용할 기본적인 Syntax 는 PDF 의 Grammar Rules 를 참조하여, 이를 cminus.y 파일에 구현하였습니다. YACC 의 문법에 따라 만들어졌으며, savedTree 변수에 최종적인 트리의 root node 를 저장하고, 상황에 따라 savedNum 내지는 savedName 변수에 데이터를 저장합니다. Token은 기존에 Parser 를 구현할 때와 동일합니다.

```
%token IF ELSE WHILE RETURN INT VOID
%token ID NUM
%token ASSIGN EQ NE LT LE GT GE PLUS MINUS TIMES OVER LPAREN RPAREN
%token LCURLY RCURLY LBRACE RBRACE SEMI COMMA
```

```
func_declaration    : type_specifier ID LPAREN params RPAREN compound_stmt
                      {
                          /* get string from stack. */
                          StackNode *tokenNode = PopStack(&top);
                          savedName = copyString(tokenNode->token);
                          free(tokenNode);

                          /* create new node. */
                          $$ = newDeclareNode(IdDec);

                          $$->attr.name = copyString(savedName);
                          free(savedName);

                          $$->child[0] = $4;
                          $$->child[1] = $6;

                          if ($$->child[0] == NULL)
                              $$->child[0] = newEmptyNode();
                      }
                    ;
```

```
var_declaration     : type_specifier ID SEMI
                      {
                          /* get string from stack. */
                          StackNode *tokenNode = PopStack(&top);
                          savedName = copyString(tokenNode->token);
                          free(tokenNode);

                          /* create new node. */
                          $$ = newDeclareNode(IdDec);

                          $$->attr.name = copyString(savedName);
                          free(savedName);
                      }
                    | type_specifier ID LBRACE NUM RBRACE SEMI
                      {
                          /* get size from stack. */
                          StackNode *tokenNode = PopStack(&top);
                          savedNum = atoi(tokenNode->token);
                          free(tokenNode);

                          /* get string from stack. */
                          tokenNode = PopStack(&top);
                          savedName = copyString(tokenNode->token);
                          free(tokenNode);

                          /* create new node. */
                          $$ = newDeclareNode(IdDec);

                          $$->attr.name = copyString(savedName);
                          free(savedName);

                          $$->child[0] = newDeclareNode(SizeDec);
                          $$->child[0]->attr.val = savedNum;
                      }
                    ;
```

변수 및 함수 선언 문법입니다. 이하 각종 문법들은 이러한 형식으로 구현되어, 최종적으로 yacc 에 의해 .c 파일과 .h 파일로 컴파일됩니다.

변수에서 허용하는 형태는 두 가지이며, 자료형은 int 와 void 가 있습니다.

int variable;

int array[5];

함수에서 혀옹하는 형태는 한 가지입니다. 반환 자료형은 마찬가지로 int 와 void 가 있으며, argument 는 void 내지는 빈 칸으로 처리할 수도 있습니다.

int main(int v, int g);

이하 각종 문법들은 과제 명세 PDF 에 첨부된 이 문법대로 구현하였습니다.

Statement 는 총 5개로 구분합니다. Compound-stmt 는 중괄호 {} 로 묶인 여러 개의 statement 를 나타냅니다. Selection-stmt 는 if 와 else 로 구성된 분기문에 해당하며, else 는 존재하지 않을 수 있습니다. Iteration-stmt 는 while 으로 구성된 반복문에 해당합니다. Return-stmt 는 return 이 포함된, 함수의 값을 반환하는 statement 에 해당합니다. 마지막으로 Compound-stmt 는 {}로 묶인 statement 의 집합을 가리킵니다.

연산 우선순위는 일반적인 수학적 연산 순서에 맞게 (*, /) -> (+,/) -> 비교 연산자 순으로 연산합니다. 단, ()로 묶인 expression 은 가장 먼저 처리합니다.

1. *program* → *declaration-list*
2. *declaration-list* → *declaration-list declaration* | *declaration*
3. *declaration* → *var-declaration* | *fun-declaration*
4. *var-declaration* → *type-specifier* **ID ;** | *type-specifier* **ID [NUM] ;**
5. *type-specifier* → **int** | **void**
6. *fun-declaration* → *type-specifier* **ID (** *params* **)** *compound-stmt*
7. *params* → *param-list* | **void**
8. *param-list* → *param-list* **,** *param* | *param*
9. *param* → *type-specifier* **ID** | *type-specifier* **ID [ ]**
10. *compound-stmt* → **{** *local-declarations statement-list* **}**
11. *local-declarations* → *local-declarations var-declarations* | *empty*
12. *statement-list* → *statement-list statement* | *empty*
13. *statement* → *expression-stmt* | *compound-stmt* | *selection-stmt* | *iteration-stmt* | *return-stmt*
14. *expression-stmt* → *expression* **;** | **;**
15. *selection-stmt* → **if (** *expression* **)** *statement* | **if (** *expression* **)** *statement* **else** *statement*
16. *iteration-stmt* → **while (** *expression* **)** *statement*
17. *return-stmt* → **return ;** | **return** *expression* **;**
18. *expression* → *var* **=** *expression* | *simple-expression*
19. *var* → **ID** | **ID [** *expression* **]**
20. *simple-expression* → *additive-expression relop additive-expression* | *additive-expression*
21. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. *additive-expression* → *additive-expression addop term* | *term*
23. *addop* → **+** | **-**
24. *term* → *term mulop factor* | *factor*
25. *mulop* → ***** | **/**
26. *factor* → **(** *expression* **)** | *var* | *call* | **NUM**
27. *call* → **ID (** *args* **)**
28. *args* → *arg-list* | *empty*
29. *arg-list* → *arg-list* **,** *expression* | *expression*

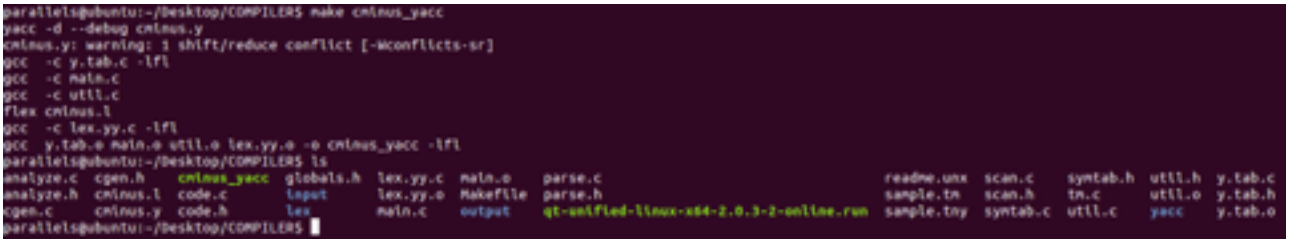
Parse Tree 생성 및 출력을 위한 각종 함수와 데이터를 추가하였습니다. 빈 노드를 나타내기 위한 newEmptyNode 함수, 변수 및 함수 선언을 위한 newDeclareNode 함수를 추가였습니다. 이를 통하여 해당 상황에 알맞는 적절한 Node 를 생성할 수 있도록 처리하였습니다. Declaration 단위의 type 을 정의하였지만, Parse Tree 생성 시에는 특별히 사용할 일이 없기에 type 처리는 일단 보류하고 이후에 필요하면 추가할 수 있도록 하였습니다.

마지막으로, Tree 출력을 위한 printTree 함수를 새로 정의한 데이터들에 맞게 개선하였습니다. 가독성을 위해 nth child 를 출력하도록 처리하고 개행하였습니다. nth child 없이 개행된 항목은 nth child node 의 sibling 으로 해석합니다.

2. 컴파일 방법 및 환경

이 프로젝트는 Ubuntu 14.04, 커널 버전 3.13.0-34-generic 에서 개발 및 테스트하였습니다.

기존 Scanner 를 구현하였던 Makefile 스크립트에 cminus_yacc 를 컴파일하기 위한 명령어를 추가하였습니다. 컴파일은 'make cminus_yacc' 명령어로 실행할 수 있으며, 각종 object file 들과 실행 파일인 cminus_yacc 파일이 생성됩니다.



make clean 명령어를 통해 중간 생성물과 최종 생성된 바이너리를 삭제할 수 있습니다.

3. 예시 및 결과 화면

테스트를 위한 test.cm 코드입니다. 기존의 Scanner 과제에서 제공된 테스트 코드와 거의 동일지만, array 선언 체크를 위해 일부 수정하였습니다.

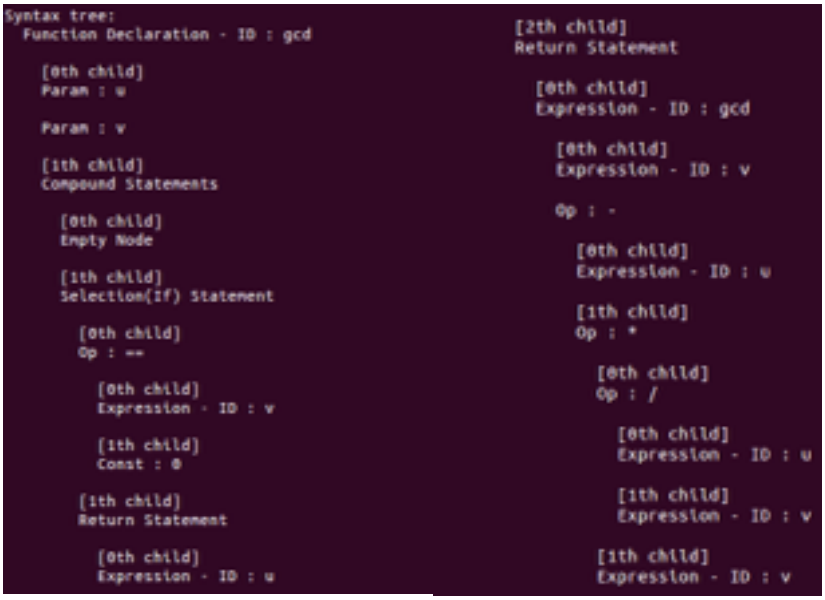
```
/* A Program to perform Euclid's
   Algorithm to computer gcd */

int gcd (int u, int v)
{
    if (v == 0) return u;
    else return gcd(v,u-u/v*v);
    /* u-u/v*v == u mod v */
}

void main(void)
{
    int x[3]; int y;
    x = input(); y = input();
    output(gcd(x,y));
}
```

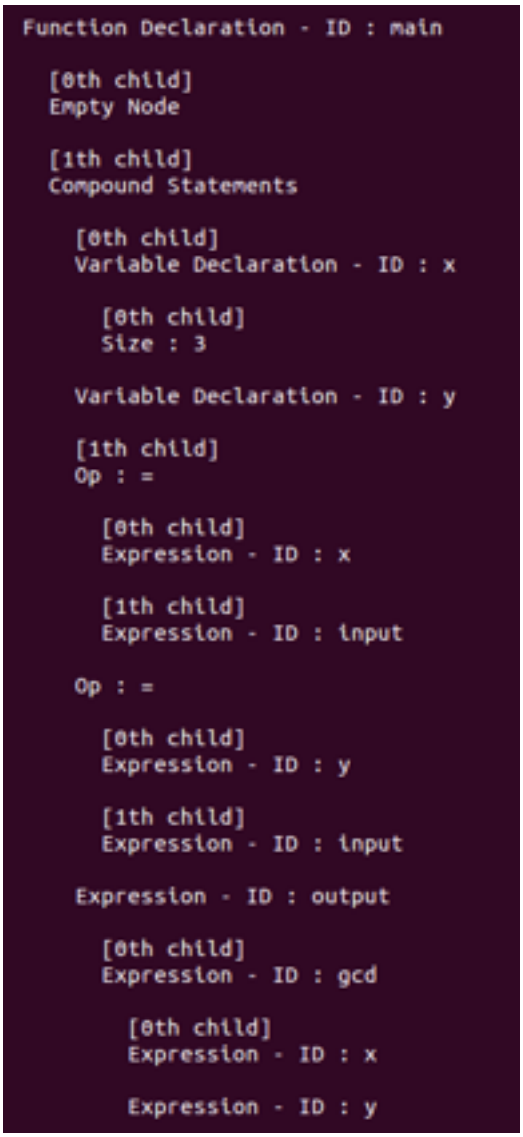
실행 방법은 다음과 같습니다.

$ ./tiny [test_file]
$ ./cminus_flex [test_file]



gcd 함수 parse 결과입니다. argument 이름과 지역 변수 선언 (Empty Node), 그리고 나머지 statement 구분이 잘 처리되었음을 확인할 수 있습니다.

특히, 2번째 return statement 에서는 뺄셈과 곱셈, 나눗셈이 복합적으로 처리하고 있습니다. 이에 대한 수식 트리도 적절하게 생성되었음을 확인할 수 있습니다.



main 함수 parse 결과입니다. argument 는 없어서 Empty Node 로 처리되었으며, Compound Statement 의 0번째 자식에는 변수 x와 y 의 선언이 처리되었습니다. x는 크기 3의 배열으로 선언되었습니다.

x = input(); , y = input(); statement 도 잘 처리되었고, output(gcd(x,y)) 에서도 gcd 의 argument 가 x, y 로 잘 처리되었음을 확인할 수 있습니다.