Compiler Design

Project #2. Parser

2018008904 이성진

본 프로젝트는 Tiny Compiler를 수정하여 C-minus를 컴파일할 수 있게 하는 parser를 구현하는 과제입니다.

1. Compilation environment and method

- 프로젝트를 개발하고 컴파일한 환경은 Intel CPU를 사용하는 Windows 10 Edu 64bit에서 WSL 2 버전으로 Ubuntu 20.04를 설치하여 사용했습니다. C compiler는 gcc를 사용했습니다.
- 프로젝트를 빌드하기 위해서는 make 명령을 활용하면 되며, 아래 Makefile의 내용을 바탕으로 세부 옵션을 설정할 수 있습니다.

- **make**: make cminus_paser와 동일합니다.
- **make cminus_parser**: Flex를 이용한 lex를 통해 만들어진 token stream을 Yacc/Bison을 이용하여 parsing하는 프로그램을 생성하도록 컴파일을 수행합니다. cminus_parser라는 이름의 바이너리 파일이 생성됩니다.
- 생성된 바이너리 파일은 cminus_parser [C-minus code file]과 같이 command argument를 통한 입력 파일을 지정해 주면 해당 파일을 scan하여 token들을 추출한 결과를 출력해줍니다.

2. Brief explanations about how to implement and how it operates

- 이 프로젝트는 C-minus로 작성된 프로그램의 lexical analysis를 위해 적절한 reserved keyword, symbol, 그리고 token 등을 정의한 다음 Yacc/Bision을 이용해 정의하는 grammar를 이용해 parsing을 진행하는 프로그램을 개발하는 것입니다.

2.1. BNF grammar for C-MINUS

program → declaration-list
declaration-list → declaration-list declaration | declaration
declaration → var-declaration | fun-declaration
var-declaration → type-specifier ID ; | type-specifier ID [NUM] ;
type-specifier → int | void
fun-declaration → type-specifier ID (params) compound-stmt
params → param-list | void
param-list → param-list , param | param
param → type-specifier ID | type-specifier ID [ ]
compound-stmt → { local-declarations statement-list }
local-declarations → local-declarations var-declarations | empty
statement-list → statement-list statement | empty
statement → expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt
expression-stmt → expression ; | ;
selection-stmt → if (expression) statement | if (expression) statement else statement
iteration-stmt → while (expression) statement
return-stmt → return ; | return expression ;

expression → var = expression | simple-expression
var → ID | ID [expression]
simple-expression → additive-expression relop additive-expression | additive-expression
relop → <= | < | > | >= | == | !=
additive-expression → additive-expression addop term | term
addop → + | -
term → term mulop factor | factor
mulop → * | /
factor → (expression) | var | call | NUM
call → ID (args)
args → arg-list | empty
arg-list → arg-list , expression | expression

2.2. Changing Tiny compiler definitions to C-minus definitions

main.c: NO_ANALYZE와 TraceParse을 TRUE로 설정하여 syntax analysis만 진행하며 parsing의 결과물인 AST만을 출력하도록 했습니다.

```
/* set NO_ANALYZE to TRUE to get a parser-only compiler */
#define NO_ANALYZE TRUE
```

```
int TraceParse = TRUE;
```

globals.h: yacc 폴더 내의 globals.h를 복사하여 수정하였는데, 기존 Tiny를 위해 정의된 NodeKind 및 각각의 Node의 종류 및 expression의 type을 판단하고 출력하기 위한 ExpType을 C-MINUS에 맞게 재정의해 주었습니다. 또한 새로 정의된 grammar에 알맞은 AST를 구성하기 위해 각각의 node를 구성하는 treeNode 구조체의 정의 역시 수정했습니다.

```
typedef enum {StmtK, ExpK, DeclK, ParamK} NodeKind;
typedef enum {CompK, IfK, IfElseK, WhileK, ReturnK} StmtKind;
typedef enum {OpK, AssignK, ConstK, VarArrK, CallK} ExpKind;
typedef enum {VarK, FuncK} DeclKind;
typedef enum {VoidK, NonVoidK} ParamKind;

/* ExpType is used for type checking */
typedef enum {Void, VoidArr, Integer, IntegerArr} ExpType;
```

```
typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union {  StmtKind stmt;
              ExpKind exp;
              DeclKind decl;
              ParamKind param;} kind;
     union { TokenType op;
             int val;
             char * name;
             ArrExp arrExp; } attr;
     ExpType type; /* for type checking of exps */
   } TreeNode;
```

util.c: globals.h와 마찬가지로 Tiny 기반의 기존 함수들을 새로 정의된 C-MINUS NodeKind와 treeNode에 알맞게 이들을 제대로 출력할 수 있도록 printTree를 수정하고 printType 함수를 정의했습니다.

```
/* printType print type specification */
void printType(ExpType type){
  switch(type){
    case Void:
      fprintf(listing, "void\n");
      break;
    case VoidArr:
      fprintf(listing, "void[]\n");
      break;
    case Integer:
      fprintf(listing, "int\n");
      break;
    case IntegerArr:
      fprintf(listing, "int[]\n");
      break;
    default:
      fprintf(listing, "Unknown type specification: %d\n", type);
```

2.3. BNF grammar Yacc implementation

- C-minus의 규칙에 맞게 AST를 구성하는 node 구조체의 형식과 출력 방식을 수정한 이후에는, 실제 AST를 생성하도록 Yacc 파일을 수정해주어야 합니다. 기존 Tiny를 구현한 grammar를 바탕으로, cminus.y 파일을 생성하고 이 파일에 C-MINUS token들을 정의한 다음 BNF grammar에 맞게 규칙을 정의했습니다.
- ID와 NUM의 경우는 각각 토큰에 해당하는 treeNode를 생성해 attr.name과 attr.val에 값을 적절히 저장하여 reduce 과정에서 사용될 수 있도록 했습니다.

```
%token IF ELSE WHILE RETURN INT VOID
%token ID NUM
%token ASSIGN EQ NE LT LE GT GE PLUS MINUS TIMES OVER
%token ERROR

%% /* Grammar for C-MINUS */

program     : declaration_list
              { savedTree = $1;}
            ;
declaration_list  : declaration_list declaration
                  {
                    YYSTYPE t = $1;
                    if (t != NULL)
                    {
                      while (t->sibling != NULL)
                          t = t->sibling;
                      t->sibling = $2;
                      $$ = $1;
                    }
                    else $$ = $2;
                  }
                  | declaration { $$ = $1; }
                  ;
declaration : var_declaration { $$ = $1; }
            | fun_declaration { $$ = $1; }
            ;
```

```
id          : ID
              {
                $$ = newExpNode(VarArrK);
                $$->attr.name = copyString(tokenString);
                //printf("%s\n", $$->attr.name);
              }
            ;
num         : NUM
              {
                $$ = newExpNode(ConstK);
                $$->attr.val = atoi(tokenString);
              }
            ;
```

- relop, mulop, addop 등의 operator는 굳이 새로운 노드를 저장할 필요성이 떨어져 각각의 token에 해당하는 tokenType 값을 (void *)로 캐스팅하여 전달하였습니다.

```
relop       : LE { $$ = (void *)LE; }
            | LT { $$ = (void *)LT; }
            | GT { $$ = (void *)GT; }
            | GE { $$ = (void *)GE; }
            | EQ { $$ = (void *)EQ; }
            | NE { $$ = (void *)NE; }
            ;
```

```
addop       : PLUS { $$ = (void *)PLUS; }
            | MINUS { $$ = (void *)MINUS; }
            ;
```

```
mulop       : TIMES { $$ = (void *)TIMES; }
            | OVER { $$ = (void *)OVER; }
            ;
```

- 기존에 정의된 grammar를 그대로 정의하게 되면 dangling else problem에 의해 shift/reduce conflict가 발생하기 때문에, if-else가 여러 번 반복되는 경우 가장 가까운 if와 else를 하나의 if-else statement로 묶어서 판단할 수 있도록 grammar를 수정하였습니다.

```
statement   : open_statement { $$ = $1; }
            | closed_statement { $$ = $1; }
            ;
open_statement  : IF LPAREN expression RPAREN statement
                  {…
                  }
                | IF LPAREN expression RPAREN closed_statement ELSE open_statement…
                | WHILE LPAREN expression RPAREN open_statement…
                ;
closed_statement  : simple_statement { $$ = $1; }
                  | IF LPAREN expression RPAREN closed_statement ELSE closed_statement…
                  | WHILE LPAREN expression RPAREN closed_statement…
                  ;
simple_statement  : expression_stmt { $$ = $1; }
                  | compound_stmt { $$ = $1; }
                  | return_stmt { $$ = $1; }
                  ;
```

3. Examples and corresponding result screenshots

make 명령을 이용해 생성된 cminus_parser를 test용 프로그램을 컴파일하기 위해 실행한 결과 주석에 해당되는 부분은 무시되고 나머지 코드에 대한 lexical analysis가 이루어진 뒤 결과물인 token stream을 정의된 grammar를 통해 syntax analysis가 진행된 결과 AST가 정상적으로 생성되어 출력된 것을 볼 수 있습니다.

```
/* 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; int y;
    x = input(); y = input();
    output(gcd(x,y));
}
```

test.1.txt

```
2_Parser$ ./cminus_parser ./test.1.txt > result.txt
```

```
C-MINUS COMPILATION: ./test.1.txt

Syntax tree:
  Function Declaration: name = gcd, return type = int
    Parameter: name = u, type = int
    Parameter: name = v, type = int
    Compound Statement:
      If-Else Statement:
        Op: ==
          Variable: name = v
          Const: 0
        Return Statement:
          Variable: name = u
        Return Statement:
          Call: function name = gcd
            Variable: name = v
            Op: -
              Variable: name = u
              Op: *
                Op: /
                  Variable: name = u
                  Variable: name = v
                Variable: name = v
  Function Declaration: name = main, return type = void
    Void Parameter
    Compound Statement:
      Variable Declaration: name = x, type = int
      Variable Declaration: name = y, type = int
      Assign:
        Variable: name = x
        Call: function name = input
      Assign:
        Variable: name = y
        Call: function name = input
      Call: function name = output
        Call: function name = gcd
          Variable: name = x
          Variable: name = y
```

parser result(AST)