Parser

2. Parser

Compilation environment and method

Ubuntu 22.04.01 LTS

gcc 11.3.0

/2_Parser/ 에서

make을 통해 cminus_parser 실행파일 생성

Implementation

globals.h

```
typedef enum {StmtK,ExpK,DeclK} NodeKind;
typedef enum {VarK,FuncK,ParaK} DeclKind;
typedef enum {IfK,If_ElseK,ReturnK,WhileK,CompK} StmtKind;
typedef enum {BiOpK,ConstK,CallK,AssignK,VarAccK} ExpKind;

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

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

cminus BNF grammar는 크게 statement, declaration, expression으로 나뉠 수 있다. 따라서 NodeKind는 StmtK, DeclK, ExpK로 나뉘게 열거형을 선언하고 각 StmtK, DeclK, ExpK는 세부 kind로 나뉘게 열거형을 선언한다.

util.c

```
TreeNode * newDeclNode(DeclKind kind)
{ TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
  int i;
  if (t==NULL)
    fprintf(listing,"Out of memory error at line %d\n",lineno);
  else {
    for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
    t->sibling = NULL;
    t->nodekind = DeclK;
    t->kind.decl = kind;
    t->lineno = lineno;
    t->type = Void;
  }
  return t;
}

void printType(ExpType type){
  switch (type)
  {
  case Integer:
    fprintf(listing,"int\n");
    break;
  case IntArr:
    fprintf(listing,"int[]\n");
    break;
  case Void:
    fprintf(listing,"void\n");
    break;
  case VoidArr:
    fprintf(listing,"void[]\n");
    break;
  default:
    break;
  }
}
```

globals.h에서 DeclKind를 추가했기 때문에 newDeclNode()을 만들어준다.

ExpType에 맞는 문자열을 출력해주기 위해 printType()을 만들어준다.

printTree()을 treeNode의 kind에 따라 요구된 문자열을 출력하도록 수정해준다.

cminus.y

1. identifier 처리

```
id          : ID
              {
```

```
                $$ = newDeclNode(VarK);
                savedName = copyString(tokenString);
                $$->attr.name = savedName;
              }
```

identifier가 들어왔을 때 상위 트리노드에서 identifier의 문자열을 사용하기 위해 임시 트리노드를 만들고, attr.name에 문자열을 저장해준다.

이후 상위 트리노드에 identifier의 문자열을 저장해주고 임시 트리노드를 메모리 해제해준다.

ex) 아래 코드의 5,6번째 줄. 상위 트리노드($$)에 이름을 저장해주고 임시 트리노드($1)을 메모리 해제해준다.

```
call        : id LPAREN args RPAREN
              {
                $$ = newExpNode(CallK);
                $$->child[0] = $3;
                $$->attr.name = $1->attr.name;
                free($1);
              }
```

2. int, void와 int[], void[]구분

   기존 코드

```
var_decl    : type_spec ID SEMI
              {
                $$ = $1;
                $$->attr.name = $2->attr.name;
                free($2);
              }
            | type_spec ID LBRACE num RBRACE SEMI
              {
                $$ = $1;
                $$->attr.name = $2->attr.name;
                free($2);
                $$->child[0] = $4;
              }
            ;
type_spec   : INT
              {
                $$ = newDeclNode(VarK);
                $$->type = Integer;
              }
            | VOID
              {
                $$ = newDeclNode(VarK);
                $$->type = Void;
              }
```

var_decl이 위의 규칙으로 파싱될 때에는 $$의 type가 Inteager 혹은 Void이어야한다. 반면 아래의 규칙으로 파싱될 때에는 $$의 type가 IntArr 혹은 VoidArr이어야한다. 하지만 type_spec에서는 트리노드에 어느 타입을 저장해야 할지 모른다.

문법을 아래와 같이 수정하여 이러한 문제를 해결했다.

```
var_decl    : INT id SEMI
              {
                $$ = newDeclNode(VarK);
                $$->type = Integer;
                $$->attr.name = $2->attr.name;
                free($2);
              }
            | VOID id SEMI
              {
                $$ = newDeclNode(VarK);
                $$->type = Void;
                $$->attr.name = $2->attr.name;
                free($2);
              }
            | INT id LBRACE num RBRACE SEMI
              {
                $$ = newDeclNode(VarK);
                $$->type = IntArr;
                $$->attr.name = $2->attr.name;
                free($2);
                $$->child[0] = $4;
              }
            | VOID id LBRACE num RBRACE SEMI
              {
                $$ = newDeclNode(VarK);
                $$->type = VoidArr;
                $$->attr.name = $2->attr.name;
                free($2);
                $$->child[0] = $4;
              }
            ;
```

수정된 문법에 의하면 다음 토큰이 무엇이 오는지에 따라 type을 다르게 저장해줄 수 있다.

3. if, if-else의 shift/reduce conflict

```
selec_stmt  : IF LPAREN exp RPAREN stmt ELSE stmt
              {
                $$ = newStmtNode(If_ElseK);
                $$->child[0] = $3;
                $$->child[1] = $5;
                $$->child[2] = $7;
              }
            | IF LPAREN exp RPAREN stmt
```

```
              {
                $$ = newStmtNode(IfK);
                $$->child[0] = $3;
                $$->child[1] = $5;
              }
            ;
```

```
selec_stmt: IF LPAREN exp RPAREN stmt • ELSE stmt
          | IF LPAREN exp RPAREN stmt •
```

selec_stmt의 FOLLOW에는 ELSE가 있기 때문에 위와 같은 상황에서 shift/reduce conflict가 발생한다.

본 Project에서는 가까운 IF와 ELSE가 묶여야 하기 때문에 위와 같은 상황에서 shift를 해야한다.

```
%left LPAREN RPAREN
%left ELSE
```

Definitions부분에 위와같이 ELSE의 우선순위를 더 높게 설정하여 shift/reduce conflict을 해결했다.

Sample

input

```
int check(int bag){
    int m;
    m = 0;
    return m >= M;
}

int main(int argc, void argv [])
{
    int testcase;
    int T;
        int left;
        int mid;
        left = 0;
        right = 10000;
    cintie(0);
    couttie(0);
        candyclear();

        while (left < right){
            mid = left + right + 1 / 2;
            if (check(mid)){
```

```
                left = mid;
            } else {
                right = mid;
            }
        }
    return 0;
}
```

output

```
C-MINUS COMPILATION: test.3.txt

Syntax tree:
  Function Declaration: name = check, return type = int
    Parameter: name = bag, type = int
    Compound Statement:
      Variable Declaration: name = m, type = int
      Assign:
        Variable: name = m
        Const: 0
      Return Statement:
        Op: >=
          Variable: name = m
          Variable: name = M
  Function Declaration: name = main, return type = int
    Parameter: name = argc, type = int
    Parameter: name = argv, type = void[]
    Compound Statement:
      Variable Declaration: name = testcase, type = int
      Variable Declaration: name = T, type = int
      Variable Declaration: name = left, type = int
      Variable Declaration: name = mid, type = int
      Assign:
        Variable: name = left
        Const: 0
      Assign:
        Variable: name = right
        Const: 10000
      Call: function name = cintie
        Const: 0
      Call: function name = couttie
        Const: 0
      Call: function name = candyclear
      While Statement:
        Op: <
          Variable: name = left
          Variable: name = right
        Compound Statement:
          Assign:
            Variable: name = mid
            Op: +
              Op: +
                Variable: name = left
                Variable: name = right
              Op: /
```

```
              Const: 1
              Const: 2
          If-Else Statement:
            Call: function name = check
              Variable: name = mid
            Compound Statement:
              Assign:
                Variable: name = left
                Variable: name = mid
            Compound Statement:
              Assign:
                Variable: name = right
                Variable: name = mid
      Return Statement:
        Const: 0
```