

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 해제해 준다.

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 Integer 혹은 Void이어야한다. 반면 아래의 규칙으로 파싱될 때에는 \$\$의 type가 IntArr 혹은 VoidArr이어야한다. 하지

만 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 LBACE num RBACE SEMI
          {
            $$ = newDeclNode(VarK);
            $$->type = IntArr;
            $$->attr.name = $2->attr.name;
            free($2);
            $$->child[0] = $4;
          }
| VOID id LBACE num RBACE 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

The screenshot displays the C-MINUS compiler interface. On the left, the source code in `test3.txt` is shown:

```
1 int check(int bag){  
2     int m;  
3     m = 0;  
4     return m >= M;  
5 }  
6 int main(int argc, void argv []){  
7     int left;  
8     int mid;  
9     while (left < right){  
10        mid = left + right / 2;  
11        if (check(mid)){  
12            left = mid;  
13        } else {  
14            right = mid;  
15        }  
16    }  
17    return 0;  
18 }
```

On the right, the generated syntax tree for `test3.txt` is displayed:

```
1 C-MINUS COMPILATION: test3.txt  
2  
3 Syntax tree:  
4   Function Declaration: name = check, return type = int  
5   Parameter: name = bag, type = int  
6   Compound Statement:  
7     Variable Declaration: name = m, type = int  
8     Assign:  
9       Variable: name = m  
10      Const: 0  
11   Return Statement:  
12     Op: >=  
13     Variable: name = m  
14     Variable: name = M  
15   Function Declaration: name = main, return type = int  
16   Parameter: name = argc, type = int  
17   Parameter: name = argv, type = void[]  
18   Compound Statement:  
19     Variable Declaration: name = left, type = int  
20     Variable Declaration: name = mid, type = int  
21     While Statement:  
22       Op: <  
23       Variable: name = left  
24       Variable: name = right  
25     Compound Statement:  
26       Assign:  
27         Variable: name = mid  
28       Op: +  
29       Variable: name = left  
30       Op: /  
31       Variable: name = right  
32       Const: 2  
33     If-Else Statement:  
34       Call: function name = check  
35       Variable: name = mid  
36     Compound Statement:  
37       Assign:  
38         Variable: name = left  
39         Variable: name = mid  
40     Compound Statement:  
41       Assign:  
42         Variable: name = right  
43         Variable: name = mid  
44     Return Statement:  
45     Const: 0
```