hw4b

B511209 최아성

May 2019

1 callgrap 설계

1.1 자료구조

1.line:
링크드리스트 구조체
이 객체는 피호출 함수의 라인을 저장한다.
같은 함수가 여러 라인에서 호출될경우 링크드리스트로 추가 연결
내부 멤버는 **int line_num과 line *next_Line**

2.inner:
정의된 함수를 표현한 구조체이다.
내부 멤버
inner *in[] : 함수가 정의부에서 호출한 함수 포인터(내장함수 X)
char bult_in[]: 함수가 정의부에서 호출한 내장함수

내장함수는 추가적인 콜그래프를 그리지 않는 리프노드라서 분리했다.
char func_name[]: 정의한 함수의 이름
line *in_line[]: 내장함수가 아닌 피호출함수의 라인저장
line *bult_line[]: 피호출 내장함수의 라인 저장

그 외 콜그래프체크,자신의 caller함수 포인터, 경로저장, 피호출함수의 총개수, 링크드리스트 마지막을 포인터 등등 알고리즘의 효율을 위한 멤버가 있습니다.
코드page에 상세하게 설명하겠습니다.

3. inner FUNC[]
선언되는 함수들을 저장할 배열입니다.
각 FUNC들은 내부멤버 포인터 in을 이용해서
서로를 피호출 함수로 참조할 수 있습니다.

4.findex(const char *name,int len)함수
FUNC배열의 인덱스 0 ~ len 중에서 함수이름이 name과 같은 객체를 찾아
인덱스를 반환해줍니다. 없을 경우 -1 반환

5.overLab(char * word,inner *FUNC,int len,int bult)함수
피호출 함수를 호출함수의 자료구조 안에 저장시켜주는 함수이다.
word는 피호출함수의 이름명이고, len은 호출함수의 FUNC 인덱스이다.
bult는 피호출함수의 FUNC 인덱스이며, -1일 경우 내장함수라는 뜻이다
만약 이미 저장되있는 함수라면 카운트만 증가시켜준다.

6.그 외 전역변수
name : lex에서 IDENTIFER의 문자열을 가져오는 배열
Line: lex에서 기본 1부터 시작해서 \n 토큰을 만날때마다 1증가
funcindex: 정의중인 함수의 FUNC 인덱스이다.
main_index: main함수의 FUNC 인덱스
funCall[]:호출,선언된 함수의 이름 저장
파라미터의 name과 섞이기 전에 빼오기 위해서.
len: 정의된 함수의 총 개수, FUNC의 추가와 접근을 위해 필요
checkf: 함수 호출인지 체크(안전성)

7.callgraph(inner *Temp,int deph) 함수
완성된 자료구조를 callgraph형식으로 출력해주는 함수

1.2 알고리즘

토큰을 읽으며 자료구조를 완성하는 알고리즘.

1. 함수의 정의문법 토큰을 간단하게

tpye function_name (parameter) statement 이렇게 나타낼 수 있다.

2. 이때 함수가 선언 될때마다 FUNC에 새로운 함수를 추가해야되는데,
statement에서 함수 호출 토큰인 tValue(밑의 코드참조)가 reduce 됨으로
적어도 그 전에 추가를 해야된다, 함수의 추가는 function_name(p) 부분의 토큰인

direct_declaration 토큰이 reduce 될 때 한다.
(함수의 이름은 function_name 다음 토큰 '('에서 저장한다.)
(parmeter 토큰의 name이 함수명으로 저장되는 것 방지하기 위해서 이다.)

3.FUNC에 함수를 추가할때는 findex함수를 사용하여
전에 이미 선언된적 있는 함수인지 검사한다.
만약 있다면 statement에서 호출될 함수들에게 funcindex만 넘겨준다

4.현재 statement에 해당하는 함수의 인덱스를 funcindex에 저장하여
statement 안에서 함수 호출토큰들(tValue)이 reduce 될 때
FUNC[funcindex]에 overLab함수를 이용하여 저장한다.

5.함수호출토큰(tValue)은 findex함수로 FUNC 배열에 있는 함수인지 찾는다
없다면 내장함수라 판단하고 저장한다.
(**함수를 호출하려면 이전에 분명 선언되었기 때문이다.**)
있다면 Caller함수 구조체 안의 포인터 in[]이 callee함수를 가르킨다

6.caller함수 inner 구조체 안에는 callee함수가 어느 라인에서 호출됐는지
저장해주는 line 구조체 객체가 있다.
callee함수마다 line 배열을 만들면 2차배열이므로 메모리 부하를 줄이기위해
링크드 리스트로 구현했다. overLab할시 line 객체를 할당하여

caller 구조체 내부의 line의 링크드 리스트와 연결한다.

7.yyparse()가 끝나면 자료구조 안에 데이터 셋이 완성된다.
출력 함수 callgraph(inner Temp, int depth) 를 사용하여 그래프를 텍스트형태로 출력한다.

callgraph는 Temp를 받아 Temp의 피호출함수들을 출력하고 피호출 함수들에 대해 depth을 1증가하여 다시 callgraph를 사용한다.
depth의 크기만큼 tab을 출력하여 caller함수와 callee함수가 구분되게 출력한다

callgraph은 callback함수의 무한루프와,과도한 그래프가 중복되는 것을 막기위해 한번 callgraph가 써졌던 함수가 다시 오면, 그래프를 출력하지 않고
전에 그려졌던 그래프의 경로를 출력하고 return한다.
caller와 callee가 같은 경우도 재귀함수라고 출력한뒤 callgraph를 실행하지 않는다

1.3 후기

직접 콜그래프를 그리면서 어떤 자료구조안에 함수들을 채우고
서로 관계를 지어야 할지 고민을 많이 했다
제가 생각한 아이디어는 inner라는 구조체로 각각의 함수를 객체화시켜
구조체 내부에서 피호출 함수 객체를 pointer로 가르키는 형태였다
즉 이 자료구조는 함수가 정의 될때 피호출함수가 먼저 객체화 되어있지 않으면 pointer로 가르킬 수가 없다는 것이다.
이는 c언어의 문법이 위에서 아래로 토큰화가 진행되기 때문이고
그리고 실제 c언어 또한 컴파일할때, 함수를 사용하기 위해서는
적어도 그 전에 선언을 해야된다.
그렇게 때문에 자료구조 또한 함수의 선언혹은 정의시에 객체화를 시켰다
C언어의 함수가 다루어지는 원리에 대해 조금은 느낌을 알 수 있었습니다.

2 Yacc 코드

hw3에서 input1.c ~ input5.c 파일이 오류없이 전부 돌아가게 수정한 뒤 콜그래프를 그리기 위해 추가된 코드들입니다.(hw3과 중복되는 코드는 생략)

```
%{
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int yylex();

 호출된 라인을 링크드 리스트로 저장한다 용량의 한계때문에 링크드리스트 활용
typedef struct line {
        int line_num;
        struct line *next_Line;
}line;

 정의된 함수 구조체
typedef struct inner{
    struct inner *in[4000]; //호출한 함수 최대 4000개
    int in_count[4000];     //같은 함수가 호출된 횟수 저장
    line *in_line[4000];    //호출된 라인의 링크드리스트 첫 포인터
    line *in_last_line[4000];//라인의 마지막포인터
    int num;                //내장함수가 아닌 콜한 함수의 개수
    char func_name[21];     //현재 함수의 이름, 이름은 20자가 최대
    int bnum;               //호출한 내장함수 개수
    char bult_in[4000][21]; //정의부에서 호출한 내장함수 최대 4천개
    int bult_count[4000];   //같은 내장함수가 호출된 횟수 저장
    line *bult_line[4000];  //호출된 라인의 링크드리스트 첫 포인터
    line *bult_last_line[4000];//라인의 리스트 마지막 포인터
    int checkCall;              그래프 출력 중복방지flag
    struct inner *caller;       자신을 부른 함수를 포인터
    struct inner *pathCallee;   caller를 역순으로 자신의 경로 포인터
}inner;
```

```
 FUNC: 현재까지 선언된 함수를 저장하고 있는 배열
 함수는 최대 5천개까지 생성 가능
inner FUNC[5000];

함수명으로 FUNC[]의 어느 배열에 있는지 인덱스를 찾아주는 함수
int findex(const char *name,int len) {
        int i;
        for (i = 0; i < len; i++) {
                if (strcmp(FUNC[i].func_name, name) == 0) {
                        return i; 인덱스 리턴
                }
        }
        return -1;
}
int Line=1;

 overLab: 함수 정의부에서 콜된 함수를 자료구조에 저장하는 함수
        전에 이미 저장된 함수는 또 저장하지 않고 카운트 증가
 word: 호출된 함수명
 len : 현재 검사중인 함수의 인덱스
 bult: 내장함수인지 flage, 아니라면 word명인 FUNC의 인덱스를 가지고 있음

void overLab(char * word,inner *FUNC,int len,int bult) {
    int i;
    //내장 함수라면 bult_in[]에 저장
    if (bult == -1) {
        예전에 호출했던 함수인지 검사
        for (i = 0; i < FUNC[len].bnum; i++) {
            if (strcmp(word, FUNC[len].bult_in[i]) == 0) {
            FUNC[len].bult_count[i]++;//맞으면 카운트 증가

        //몇줄에서 호출되는지 링크드 리스트로 저장한다.
```

```
            FUNC[len].bult_last_line[i] -> next_Line = (line *)malloc(sizeof(line));
            FUNC[len].bult_last_line[i] = FUNC[len].bult_last_line[i]->next_Line;
            FUNC[len].bult_last_line[i]->line_num = Line;
                                        return;
                                };
                    }
 처음 호출된 함수라면 자료구조에 추가
strcpy(FUNC[len].bult_in[FUNC[len].bnum],word);
FUNC[len].bnum += 1; FUNC[len].bult_count[FUNC[len].bnum-1]=1;
 호출된 라인 링크드리스트 추가
FUNC[len].bult_line[FUNC[len].bnum - 1] = (line *)malloc(sizeof(line));
FUNC[len].bult_line[FUNC[len].bnum - 1]->line_num = Line;
FUNC[len].bult_last_line[FUNC[len].bnum - 1] = FUNC[len].bult_line[FUNC[len].bnu
return;
            }

            정의한 함수라면 in[]에 저장
        else {
            예전에 호출했던 함수인지 검사
            for (i = 0; i < FUNC[len].num; i++) {
                if (strcmp(word, FUNC[len].in[i]->func_name) == 0) {
                    FUNC[len].in_count[i]++;맞으면 카운트 증가

                //몇줄에서 호출되는지 링크드 리스트로 저장한다.
                FUNC[len].in_last_line[i] -> next_Line = (line *)malloc(sizeof(line)
                FUNC[len].in_last_line[i] = FUNC[len].in_last_line[i]->next_Line;
                FUNC[len].in_last_line[i]->line_num = Line;
                return;
                }
                    }
 처음 호출된 함수라면 자료구조에 추가
FUNC[len].in[FUNC[len].num] = &FUNC[bult];
FUNC[len].num += 1; FUNC[len].in_count[FUNC[len].num-1]=1;
```

```
 호출된 라인 링크드리스트 추가
FUNC[len].in_line[FUNC[len].num - 1] = (line *)malloc(sizeof(line));
FUNC[len].in_line[FUNC[len].num - 1]->line_num = Line;
FUNC[len].in_last_line[FUNC[len].num - 1] = FUNC[len].in_line[FUNC[len].num - 1]
return;
        }
}
```

나머지 전역변수!

```
int funcindex = 0;   //현재 정의중인 함수가 FUNC[]에 저장된 인덱스
int main_index = -1; main 함수 인덱스 저장
char funcCall[21];   //호출,선언된 함수이름 저장
char name[21];       //lex파일에서 IDENTIFIER토큰의 이름을 가져옴
int len =0;          //지금까지 정의된 함수의 총개수
int checkf = 0;      //함수 호출인지 체크
%}
```

콜그래프를 구하기 위해 코드가 추가된 부분.
그 외 코드는 생략했습니다

함수 매개변수가 이름으로 읽히는 것 방지,'('전의 name을 funCall에 저장

```
fValue : id '(' {strcpy(funcCall, name); checkf = 1;}
       ;

tValue : id     후위 연산자 좌결합, 함수의 호출문법이 있다.
       | fValue ')'   함수가 호출되었을때
       {
       if(checkf == 1)//함수 호출이면
                {
         1.호출된 함수의 자료구조가 FUNC 몇번째 인덱스에 있는지 찾는다.
                        int temp = findex(funcCall, len);
                        못찾으면 '-1' 내장 함수!

         2.현재정의 중인 함수 자료구조 안에 저장한다.
                        overLab(funcCall,FUNC,funcindex,temp);
                }
                checkf=0;
       }

       //매개변수있는 함수 호출도 동일
       | fValue expression ')'
       {
       if(checkf == 1)
                {
                        int temp = findex(funcCall, len);
                        overLab(funcCall,FUNC,funcindex,temp);
                }
                checkf=0;
```

```
        }

        | tValue PP   {op++;}
        | tValue MM {op++;}
        | tValue  '['expression']'
        ;
```

=======생략=======

```
 함수일때 다음 파라미터의 name으로 섞이기 전 함수명 저장
funCheck:  '('  {strcpy(funcCall,name);}
        ;


 선언의 기본형 (변수,함수,배열)
direct_declarator: IDENTIFIER
    | '(' declarator ')'

    함수의 선언
    | direct_declarator funCheck ')'
    {
                예전에 선언됐던 함수인지 아닌지 체크
        int index = findex(funcCall,len);
        if( index == -1){     //처음 선언되는 함수일 때
                        FUNC에 새로운 함수 추가
            strcpy(FUNC[len].func_name,funcCall);
            FUNC[len].num = 0;    //어차피 전역 초기값은 0이다.
            FUNC[len].bnum = 0;
            funcindex = len;      //새롭게 추가된함수의 인덱스는 len
함수 정의부안에 호출되는 함수를 저장할때 어느 자료구조에 저장할지 알려줌

            len++;                //총길이 증가
```

```
        }

    else{      전에 선언했던 함수를 나중에 정의 경우
        funcindex = index;함수의 인덱스는 findex로 찾은 인덱스
        총길이의 증가는 없음
    }
    if(main_index == -1){ main 함수의 인덱스 따로 저장
        if(strcmp("main",funcCall)==0){
            main_index = len-1;
        }
    }
}


매개변수가 있는 함수도 동일
| direct_declarator funCheck parameter_list')'
    {
        int index = findex(funcCall,len);

       if( index == -1){
            strcpy(FUNC[len].func_name,funcCall);
            FUNC[len].num = 0;
            FUNC[len].bnum = 0;
            funcindex = len;
            len++;
            funct++; checkFunct = 1;
       }

       else{
            funcindex = index;
       }
       if(main_index == -1){
            if(strcmp("main",funcCall)==0){
```

```
                main_index = len-1;
            }
        }
}

| direct_declarator funCheck identifier_list')'
{
    int index = findex(funcCall,len);

            if( index == -1){
                strcpy(FUNC[len].func_name,funcCall);
                FUNC[len].num = 0;
                FUNC[len].bnum = 0;
                funcindex = len;
                len++;
                funct++; checkFunct = 1;
             }

            else{
                funcindex = index;
             }

            if(main_index == -1){
                if(strcmp("main",funcCall)==0){
                    main_index = len-1;
                }
             }
    }


| direct_declarator'['']'
| direct_declarator'[' cValue ']'
;
```

=======생략=======

%%

```
//자료구조를 콜그래프 형태로 출력하는 함수!!
//단정한 출력형태에 맞추다보니 부가적인 코드가 많습니다.
//알고리즘 적인 부분은 주석있는 부분만 봐도 괜찮습니다.
//DFS 방식으로 호출한 함수를 끝까지 찾고 재귀를 반복하는 함수 입니다.
//inner 자료구조를 받아 시작
//depth는 그래프 깊이에 따라 tab으로 구분하기 위한 변수

void callgraph(inner *Temp,int depth) {
    if (!Temp) return; 더이상 함수호출이 없으면 종료
    int i;
    int j;
    line *temp;
    현재 호출된 함수 이름 출력
    printf(" %s -> {", Temp->func_name);
    fprintf(f," %s -> {", Temp->func_name);

    check된 함수(이전에 그래프가 이미 그려진 함수)
    콜백함수같이 무한루프에 빠질 수 있는 케이스 처리가능
    check된함수는 그래프를 더 그래지 않고,예전에 그렸던 그래프 경로를 알려줌

    if (Temp -> checkCall) { 체크된 함수라면!

        inner *calltemp = Temp;
        printf(" 경로:");
        fprintf(f," 경로:");
```

```
        caller를 역순으로 pahtcallee에 저장
        호출한 함수를 순서대로 부르면 경로의 역순이다!
        Temp->pathCallee = NULL;
        while (calltemp->caller) {
            calltemp->caller->pathcaller = calltemp;
            calltemp = calltemp->caller;
        }

        경로를 순서대로 출력
        while (calltemp) {
            printf(" %s ", calltemp->func_name);
            fprintf(f," %s ", calltemp->func_name);
            calltemp = calltemp->pathcaller;
            if(calltemp) {printf("->"); fprintf(f,"->"); }
        }
        printf("에서 먼저 그래프 그려짐. ");
        fprintf(f,"에서 먼저 그래프 그려짐. ");
        printf("}\n\n\n");
        fprintf(f,"}\n\n\n");
        return;
    }

    처음 호출된 함수 처리
    Temp -> checkCall = 1;

    함수 내부에서 피호출된 함수들의 이름과 선언된 위치와 횟수를 출력
    for (i = 0; i < Temp->num; i++) {
        함수의 호출된 횟수 출력
        printf(" %s:%d.times(",Temp->in[i]->func_name,Temp->in_count[i]);
        fprintf(f," %s:%d.times(",Temp->in[i]->func_name,Temp->in_count[i]);
        temp = Temp->in_line[i];
        함수가 호출된 라인 출력
```

```
        while (temp) {
            printf("%d", temp->line_num);
            fprintf(f,"%d", temp->line_num);
            temp = temp->next_Line;
            if(temp){ printf(","); fprintf(f,","); }
        }

        printf(")");  fprintf(f,")");
        if(i<Temp->num-1 ){ printf(",  "); fprintf(f,",  "); }
    }
```

내장함수 출력!

```
    if(Temp->bnum > 0) {
        printf(" |내장함수|:"); fprintf(f," —내장함수|:");
    }
```

내장함수도 횟수와 라인 출력

```
    for (i = 0; i < Temp->bnum; i++) {
        printf(" %s:%d.times(", Temp->bult_in[i],Temp->bult_count[i]);
        fprintf(f," %s:%d.times(", Temp->bult_in[i],Temp->bult_count[i]);
        temp = Temp->bult_line[i];
        while (temp) {
            printf("%d", temp->line_num);
            fprintf(f,"%d", temp->line_num);
            temp = temp->next_Line;
            if(temp) {printf(","); fprintf(f,","); }
        }
        printf(")"); fprintf(f,")");
        if(i<Temp->bnum-1){ printf(",  "); fprintf(f,",  "); }
    }

        printf(" }\n\n"); fprintf(f," }\n\n");
```

```
내장함수가 아닌, 정의된 함수중에서
피호출함수도 재귀적으로 그래프 출력 start
    if(Temp->num > 0){
        for (j = 0; j < depth-1; j++) { printf("\t\t|"); fprintf(f,"\t\t|"); }
            printf("========== %s   Start ==========\n\n\n",Temp->func_name);
            fprintf(f,"========== %s  Start ==========\n\n\n",Temp->func_name);
        }
        else{printf("\n"); fprintf(f,"\n");}
        for (i = 0; i < Temp->num; i++) {
        for (j = 0; j < depth; j++) { printf("\t\t|"); fprintf(f,"\t\t|"); }

    피호출 함수는 자신을 호출한 함수를 저장한다.
    if(Temp->in[i]->checkCall !=1 ) {Temp -> in[i] -> caller = Temp;}
        Caller가 누군지 출력한뒤 피호출함수의 그래프도 시작
        printf(" (caller:%s)",Temp->func_name);
        fprintf(f," (caller:%s)",Temp->func_name);

        피호출 함수가 호출한 함수도 재귀적으로 출력
        피호출과 호출이 같으면(재귀함수)callgraph함수는 더 사용하지 않는
다.
        if(Temp == Temp->in[i]){
            printf(" %s -> {재귀함수}\n\n\n",Temp->in[i]->func_name);
            fprintf(f," %s -> { 재귀함수 }\n\n\n",Temp->in[i]->func_name);
        }
        else{ //callgraph를 depth 1증가하여 재귀적으로 실행
            callgraph(Temp->in[i],depth+1);
        }
    }
    내부의 그래프가 완성되면 end 출력
    if(Temp->num > 0){
        for (j = 0; j < depth-1; j++) { printf("\t\t|"); fprintf(f,"\t\t|"); }
            printf("========== %s   End ==========\n\n\n\n\n",Temp->func_name);
            fprintf(f,"========== %s   End==========\n\n\n\n\n",Temp->func_name);
```

```
                }
}
 main 함수 시작
int main(void){
        int i;
        FILE * f;
        f = fopen("B511209.txt","wt");
        if(f == NULL){
                puts("파일열기 오류, 같은 폴더에 B511209.txt 파일이
있는지 확인해주세요.\n");
                return 0;
        }

        yyparse();

    Temp는 main이 담겨있는 자료구조 변수 포인터
        inner * Temp = &FUNC[main_index];

        printf("\n"); fprintf(f,"\n");

        main부터 그래프 출력 시작!!
        callgraph(Temp,1);

        선언(정의)된 함수들 전부 출력
    fprintf(f,"\n\n\n      선언된 함수의 종류\n--------------------------------------\n");
    for(i = 0; i < len ; i++){
        printf(" %s \n",FUNC[i].func_name);
        fprintf(f," %s \n",FUNC[i].func_name);
    }
    printf("\n");
        fclose(f);
        return 0;
}
```