[Compiler]

Programming Project #3

- Bottom up parsing을 이용한 mini-c 인터프리터 개발 -

Professor : 류기열 교수님

name : 박수린

student number : 201720723

목차

2. 문제 분석

2.1. grammar rule 분석

- grammar 의 대부분은 과제문서 1 을 참고하였는데, 다른 부분을 중점적으로 설명하도록 하겠다.
- 먼저 시작은 program 이다.

```
program:
                                        { }
    | program stmt                      {execute($2);}
    | program fun_list stmt             {execute($2); execute($3);}
    | program fun_list                  {execute($2);}
    | error                             {printf("---line err\n");}
    ;
```

- 함수로 이루어질 수도 있고 , 문장만 있을 수도 있다. root 부분이므로 tree 를 순회하며 실행시키는 execute 를 호출하고 문장이나 if,while 단위로 오류가 있으면 에러 메시지를 출력한다.

```
fun_list:
    fun_def fun_list    {$$ = mkNode(0, $1, $2, NULL);}
    | fun_def           {$$ = $1;}
    ;

fun_def:
    DEF variable LP expr_list RP LEFT decl stmt_list RIGHT  {$$ = mkFunc(21, $2, $4, $7, $8);}
    | DEF variable LP expr_list RP LEFT stmt_list RIGHT     {$$ = mkFunc(21, $2, $4, NULL, $7);}
    | DEF variable LP RP LEFT decl stmt_list RIGHT          {$$ = mkFunc(21, $2, NULL, $6, $7);}
    | DEF variable LP RP LEFT stmt_list RIGHT               {$$ = mkFunc(21, $2, NULL, NULL, $6);}
    ;

decl:
    LOCAL var_list SEMICOLON    {$$ = $2;}
    ;
```

- 함수 구현 부분이다. 21 번 타입으로 함수 노드를 만들어준다.
- local 타입을 지원한다,

```
stmt:
    expr SEMICOLON          {$$ = $1;}
    | empty_stmt            {$$ = $1;}
    | print_stmt SEMICOLON  {$$ = $1;}
    | assign_stmt SEMICOLON {$$ = $1;}
    | control_stmt          {$$ = $1;}
    | return_stmt SEMICOLON {$$ = $1;}
    | block                 {$$ = $1;}
    ;
```

- 문장에는 여러 종류가 있다. 각각 문장 타입에 따라 수행된 action 의 결과를 root 로 올려준다.

```
expr:
    expr PLUS term                {$$ = mkNode(5, $1, $3, NULL);}
    | expr MINUS term             {$$ = mkNode(6, $1, $3, NULL);}
    | expr GT expr                {$$ = mkNode(7, $1, $3, NULL);}
    | expr GEQ expr               {$$ = mkNode(8, $1, $3, NULL);}
    | expr LT expr                {$$ = mkNode(9, $1, $3, NULL);}
    | expr LEQ expr               {$$ = mkNode(10, $1, $3, NULL);}
    | expr EQ expr                {$$ = mkNode(11, $1, $3, NULL);}
    | expr NEQ expr               {$$ = mkNode(12, $1, $3, NULL);}
    | MINUS expr %prec UMINUS     {$$ = mkNode(13, $2, NULL, NULL);}
    | variable LP expr_list RP    {$$ = mkNode(19, $1, $3, NULL);}
    | variable LP RP              {$$ = mkNode(19, $1, NULL, NULL);}
    | term                        {$$ = $1;}
    ;
```

- 가장 중요한 expr 에 대한 grammar 이다. 각각 토큰 타입에 따라 다른 노드를 만들고 각각 자식 노드를 지정한다.

```
term:
    term MUL factor     {$$ = mkNode(14, $1, $3, NULL);}
    | term DIV factor   {$$ = mkNode(15, $1, $3, NULL);}
    | factor            {$$ = $1;}
    ;

factor:
    LP expr RP  {$$ = $2;}
    | INT       {$$ = mkLeaf(16, $1, 0.0, NULL);}
    | REAL      {$$ = mkLeaf(17, 0, $1, NULL);}
    | variable  {$$ = $1;}
    | LP expr error {yyerrok; yyclearin; printf("---missing RP\n"); $$ = $2;}
    ;

variable:
    ID        {$$ = mkLeaf(18, 0, 0.0, $1);}
    ;
```

- expr 의 하위 부분인 term 과 factor 에 대한 설명이다. 숫자는 factor 에 있는데 숫자와 id 만 leaf 에 해당한다.

2.2. grammar rule 이 LALR(1) parsing 이 가능한지 분석

```
bagsulin-ui-MacBook-Pro-6:hw3_201720723 surin$ make
make clean
rm -rf *.tab.c *.tab.h *.yy.c mc *.output
bison -d -b y mc.y
mc.y: conflicts: 12 shift/reduce
flex mc.l
gcc -o mc y.tab.c lex.yy.c -ly -ll
```

- 실행결과는 위와 같다. 12 shift/reduce 가 발생하였다고 분석되었다. 이를 -v 옵션을 주어 y.output 에서 확인할 수 있었다.

3. 설계

3.1. 주요 자료 구조

- 복잡했던 이전 과제의 lex 파일에 비해 이번에는 bison 을 활용하여 컴파일 하기 때문에 매우 단순해졌다. 그렇기 때문에 lex 파일에 “y.tab.h”의 헤더파일을 추가하고 YYSTYPE 을 이용하여 lexical value 을 공유할 수 있다.
- .y 파일에는 함수를 구현함으로써 조금 복잡해졌지만 module 화 하려고 노력하였다. 먼저 구조체에 대해 설명하겠다

```
typedef struct symbol{
    char id[SYMBOL_MAX+1];
    double val;
}SYMBOL;
```

```
SYMBOL symbol_table[SYMBOL_TABLE_MAX];
```

- SYMBOL 구조체 이다. symbol table 에 id 와 그에 맞는 value 를 저장한다.

```
typedef struct Node{
    int t;
    union val{
        double real_constant;
        int integer_constant;
    }value;
    char symbol[11];
    struct Node* left;
    struct Node* right;
    struct Node* mid;
    struct Node* name;
}NODE;
```

- NODE 구조체 이다. 멤버에 대한 설명을 해보자면 t 는 NODE 의 타입이다. 각각 정수 또는 실수의 값을 가지므로 공용체를 이용하였고, id 일 경우에는 symbol 에

id 를 string 으로 저장한다. 그리고 각각 자식노드를 가지는데, left, right 가 왼쪽 child, 오른쪽 child 이다. 그리고 if 문은 조건을 확인하는 자식이 하나 더 필요해서 mid 라는 Node 포인터 타입의 멤버를 하나 더 선언하였고, 함수의 정의 부분에서 함수의 이름을 저장해야 하므로 name 멤버 또한 추가로 선언하였다.

```
typedef struct func{
    char id[FUNC_NAME_MAX + 1];
    SYMBOL local_table[LOCAL_TABLE_MAX];
    int arg_index[LOCAL_TABLE_MAX];
    int local_number;
    int arg_number;
    NODE* statement_list;
    double return_value;
    int alloc_argnum;
}FUNC;
```

- 이어서 func 구조체 이다. 우선 func 의 Id 가 string value 로 있고, 함수마다 local_table 이 있기 때문에 SYMBOL 타입의 리스트를 선언해주었다.
- 함수를 호출할 때 argument 를 local 변수로서 값을 저장해주어야 하기 때문에 local table 에서 argument 로 들어오는 변수들은 각각 local table 의 어디에 위치하는 지 알려주는 int 배열을 선언하였다.
- local 변수와 parameter 로 들어오는 변수의 개수를 알기 위해 local_number 와 arg_number 을 선언하였다.
  - 이 arg_number 을 함수를 호출할 때 parameter 개수에 맞지 않는 expr_list 가 들어왔을 때 handle 하기 위한 변수로서 선언해두었으나 시간이 부족하여 함수에 대한 error 는 처리하지 못하였다.
- 함수가 호출됐을 때 실행해야 할 statement 들의 시작 주소를 나타내는 NODE 포인터 타입의 statement_list 를 선언하였다.
- 함수에 return 문이 있을 때, return 값을 저장하기 위해 return_value 를 선언하였다.
- argument 를 차례로 인식하여 각각의 local table 의 값을 업데이트 해주기 위해여 현재까지 alloc 된 alloc_argnum 변수를 통해 차례대로 할당해줄 수 있다.

```
FUNC func_table[FUNC_TABLE_MAX];
```

- function 들이 여러개 선언될 수 있기 때문에, 심볼과 마찬가지로 Func_table 을 선언하여 index 값을 알면 func 을 호출하고 값에 접근할 수 있도록 하였다.

3.2. 프로그램 module hierarchy 및 module 에 대한 설명

```
void initialize_symbol_table();
void initialize_func_table();
int find_symbol(char* target);
int find_local(int index, char* target);
int find_func(char* target);
```

- 처음 main()함수가 실행되면 yyparse()를 호출하기 전에 initialize_symbol_table()과 initialize_func_table()을 호출하여 각각의 테이블의 값들을 초기화한다.
- 또한 parsing 을 수행하면서 테이블에 계속 접근할 수 있도록 find_*함수를 이용하여 반환된 index 값을 활용한다. 값을 찾지 못하면 -1 을 반환한다.
- find_local()함수는 함수의 index 를 미리 알아야 하기 때문에 parameter 가 추가되었다.

```
NODE* mkNode(int t, NODE* left, NODE* right, NODE* contidition);
NODE* mkLeaf(int t, int ival, double dval, char*s);
NODE* mkFunc(int t, NODE* name, NODE* arg, NODE* local, NODE* exe);
double execute(NODE* root);
void print_exp(NODE* root);
void alloc_local(int index, NODE* left, NODE* mid);
void alloc_arg(int index, NODE* arg);
void save_var_list(int index, NODE* root, bool islocal);
```

- mkNode(), mkLeaf(), mkFunc 은 모두 NODE 의 포인터 타입을 반환하여 grammar 의 action 으로 트리의 노드를 만들어 주는 구조이다. 일반 노드와 leaf, 그리고 함수를 구분하지 않으면 함수의 parameter 가 많아지기 때문에 이와 같이 구분하였다. if 문의 경우가 자식이 3 개여서 특별하기 때문에 mkIFNode()와 같이 구분했다면 코드의 가독성이 높아졌을 것이다.
- mc.y 파일을 컴파일 하면 yyparse()에 의해 파싱이 수행되면서 node 를 만들고 최상위 노드에서 execute()함수를 통하여 트리를 순회하면서 수식을 계산한다. execute 함수에는 node 의 root 가 파라미터로 들어오는데 root->t, 즉 노드의 타입을 switch 문을 이용해 case 를 나누어서 적합한 값을 반환한다.
- 프로그램의 가독성을 위하여 복잡한 함수를 분리하였는데 먼저 print_exp 함수가 있다. 이는 Id 면 symbol table 에서 값을 찾아 출력하고 숫자인 경우에는 정수와 실수를 구분하여 출력하도록 하였다.

- 또한 id 의 경우에는 함수를 추가적으로 구현했을 때, local_table 의 값을 탐색해야 하는 경우가 있었다. 이를 위하여 아래의 그림처럼 if 문을 사용하여 분기하였는데, 18 은 root 의 타입이 ID 임을 의미한다.

```
if(root->t == 18){
    int index = find_symbol(root->symbol);
    double val;
    if(index == -1){
        if(flag == -1){
            yyerror("syntax error : a variabl
```

- 먼저 symbol 테이블에서 index 를 찾았으나 실패한 경우 local_table 을 탐색해야하는데, 여기서 중요한 점은 local 값에 접근할 수 있는 경우는 함수가 실행 중일 때이다.
- 이를 flag 을 이용하여 구분하였는데, flag 는 전역변수로서 함수가 실행중이면 함수 테이블의 Index 값을 가지고, 그렇지 않을 경우에는 -1 을 가진다.

```
flag = index;
NODE* temp = func_table[index].statement_list;
execute(temp);
flag = -1;
```

- 잠깐 밑에서 설명할 execute()함수에서 function_call 에 대한 부분을 참고하자면, temp 변수로 statement 의 시작 주소를 가져와 함수를 실행시키는데, 이때 flag 의 값에 함수의 index 값을 넣어준다. 그리고 나서 함수의 실행이 끝나면 flag 의 값을 다시 -1 로 되돌려 놓는다.
- 그러므로 symbol_table 에도 없고 local_table 에도 없는 값을 출력하려 한다면 에러 메시지를 출력한다.

```
if(val - (int)val == 0)
    printf("%d\n", (int)val);
else
    printf("%lf\n",val);
```

- 그렇지 않은 경우에는 드디어 값을 출력하는데, 위에서 언급되었듯이 SYMBOL 구조체에는 실수형 멤버변수만이 있기때문에 출력시 정수인지 확인한는 과정을 거친다.

```
double execute(NODE* root)
{
    if(!root)
        return 0;

    switch(root->t){
        case 0 :
            execute(root->left);
            execute(root->right);
            break;
        case 1 :
            print_exp(root->left);
            break;
        case 2 :
        {
            int index = find_symbol((rc
```

- 가장 중요한 execute 함수에 대해서 설명하도록 하겠다. 우선 root 에 NULL 값이 들어오면 그 즉시 함수를 종료하고 그렇지 않은 경우에는 root 구조체의 타입에 따라 다른 문장이 실행된다.
- 타입은 정수형 숫자로 구분하였다. 중요한 경우만 설명하자면 0 은 list 타입의 노드로서 왼쪽자식과 오른쪽 자식을 차례대로 수행한다.
- 1 의 경우에는 출력문이다.
- 2 의 경우에는 assign statement 에 해당된다. 이때도 local 인지 전역변수인지에 따라 경우를 나누었다.
- 3 번은 if, 4 번은 while 문이다.

```
case 3 :
    if(root->mid == NULL)    return 0;
    if(execute(root->mid))  execute(root->left);
    else execute(root->right);
    break;
case 4 :
    while(execute(root->left)){
        execute(root->right);
    }
```

- 5~15 번까지는 사칙연산을 수행한다. 이때 0 으로 나눌 경우에는 에러 메시지를 출력하도록 하였다.
- 16, 17, 18 번은 leaf 노드의 값을 반환하는 경우이다.
- 19~21 번은 함수 수행에 관련된 것인데, 19 번은 함수가 call 되었을 때 왼쪽 자식으로 함수의 id 를 가지고 오른쪽 자식으로 parameter 들의 list 형식으로 들어오게 된다.

-
```
if(root->right)
    alloc_arg(index, root->right);
```
argument 값을 alloc_arg 함수를 통해서 local_table 에 저장해준다.
-
```
case 20 :
{
    if(flag == -1){
        yyerror("syntax error : you can only return in function");
        return 0;
    }
    else{
        double val = (double)execute(root->left);
        func_table[flag].return_value = val;
        break;
    }
}
```
- 20 번의 경우는 return 문을 처리하기 위함이다. 왼쪽 자식에 return 토큰 옆에 있는 expr 의 포인터가 저장되어 있다.
- 함수 수행중이 아니면 Return 문을 수행할 수 없으므로 오류 메시지를 출력한다.
- 그렇지 않은 경우에는 flag 에 함수의 index 가 저장되어 있으므로 left node 를 실행시켜 돌아오는 값을 val 에 임시로 저장한 후 알맞은 func_table 의 인덱스에 해당하는 return_value 에 저장한 후 case 를 종료한다.
-
```
case 21:
{
    int index = find_func((root->name)->symbol);
    if(index == -1){
        strcpy(func_table[func_number].id, (root->name)->symbol);
        func_table[func_number].statement_list = root->right;
        alloc_local(func_number, root->left, root->mid);
        func_number++;
        break;
    }
    else{
        yyerror("syntax error: the name of the function that is already defined.");
        return 0;
    }
}
```
- 마지막으로 21 번의 경우에는 func 을 새로 정의하는 부분인데, 이미 정의된 id 라면 오류 메시지를 출력한다.
- 그렇지 않은 경우에는 func_table 에 새로운 id 를 추가하고, 실행해야할 stmt_list 의 포인터를 저장해주어 함수가 호출되었을 때 참조할 수 있도록 한다.
- 그리고 local 변수 들이 선언되었을 때 이를 local table 에 id 를 추가해주어야 한다. 또한 argument 또한 함수 내에서 local 변수처럼 사용되므로 이또한 추가해주어야 한다.
- 두 경우 모두 alloc_local()이라는 함수를 이용하여 처리하는데, 자세한 실행과정은 아래와 같다.

-
```
void alloc_local(int index, NODE* left, NODE* mid)
{
    save_var_list(index, mid, false);
    save_var_list(index, left, true);
}
```
이와 같이 variable 의 list 를 post order 로 순회하여 저장하여야 하기 때문에 argument list 와 local 변수들의 list 를 각각 따로 전달하고 이는
-
```
void save_var_list(int index, NODE* root, bool islocal)
```
을 활용하여 islocal 이라는 boolean 값으로 구분한다.
-
```
if(islocal){
    if(find_symbol(root->symbol) != -1){
        yyerror("syntax error : already declared global variable");
        return;
    }
    strcpy(func_table[index].local_table[count_local].id, root->symbol);
    func_table[index].local_number++;
}
else{
    int argn = func_table[index].arg_number;
    strcpy(func_table[index].local_table[count_local].id, root->symbol);
    func_table[index].local_number++;
    func_table[index].arg_index[argn] = find_local(index, root->symbol);
    func_table[index].arg_number++;
}
```
- 값을 저장하는 과정은 local 변수라면 단순히 테이블에 추가해주면 되고, 이미 선언된 전역변수라면 에러 메시지를 출력한다.
- argument list 라면 local 테이블에 추가하고 이 index 을 arg_index 리스트에 추가해준다.
-
```
int main(int argc, char* argv[])
{
    if(argc > 1)
    {
        FILE* file;
        file = fopen(argv[1], "r");
        if(!file)
        {
            fprintf(stderr, "could not open %s!\n", argv[1]);
            exit(1);
        }
        yyin = file;
    }
    flag = -1;
    initialize_func_table();
    initialize_symbol_table();
    yyparse();

    return 0;
}
```

- 마지막으로 main 함수는 sample.mc 파일을 받아오기 위해 파일 포인터를 이용하였다.
- 위에서 설명했듯이 먼저 table 들을 초기화 하고 yyparse()를 호출한다.

4. 수행 결과

- 먼저 sample.mc 와 func.mc 두 가지 예시파일을 작성하였는데 sample.mc 는 선택사항인 함수를 제외한 예시이다. 또한 sample.mc 에는 에러가 포함된 경우를 포함하였다.

```
 1 a = 10;
 2 a = 30
 3 a = 30;
 4
 5 print a;
 6
 7 b = 30.5;
 8 c = (a+b);
 9 print c;
10
11 c = (a-b;
12 print c;
```

```
14 if(a>b
15     max = a;
16 else
17     max = b;
18
19 print max;
20
21 if(a>b)
22     max = a;
23 else
24     max = b;
25 print max;
```

```
27 i;
28 print i;
29
30 i = 0; sum = 4;
31 sum = sum/i;
32 print sum;
33
34 i = 1;
35 sum = sum + i;
36 print sum;
37
```

```
38 n = 7;
39 while(i < n){
40     i = i + 1;
41     print i;
42 }
43
44 n = -n;
45 print n;
46
47 { print 10.5;   print 9;
48
49 {print 10;}
```

각각의 line 에 대하여 설명하겠다.

- 1~12 까지의 실행결과는 아래와 같다.

```
syntax error
---line err
30
60.500000
syntax error
---missing RP
syntax error
---line err
60.500000
```

- 먼저 1~12 까지의 경우는 assign statement 와 print_stmt 에 대한 부분인데, 2 번줄의 세미콜론이 없는 경우는 syntax error –line err 를 출력한다.

- 하지만 3 번에 의해 제대로 수행된 결과 a 에는 30 이 저장되고 출력된다.
- 그리고 c 는 덧셈 결과가 정상적으로 출력되지만, 오른쪽 괄호가 닫힌 상태로는 수행되지 않아 오류 메시지를 출력 한 후 그전에 저장된 값이 출력 된다.

```
syntax error
---missing RP
syntax error
---if_stmt error
syntax error : a variable whose value is not stored in symbol_table
30.500000
```

- 14~25 의 수행 결과이다.
- 처음 if 문은 조건식에서 괄호가 닫히지 않았기 때문에 statement 를 수행하지 않아 max 가 선언되지 않은 상태라서 symbol_table 에 없다고 에러메시지가 출력되지만 그 다음에 정상적으로 수행된 후에는 a 와 b 중에 큰 값인 30.5 를 정상적으로 출력함을 확인할 수 있다.

```
syntax error : a variable whose value is not stored in symbol_table
syntax error : divide by zero
4
5
```

- 27~36 까지의 수행 결과이다.
- 사용하지 않은 i 를 출력하려고 하자 에러 메시지가 출력되고 i 에 0 을 저장하고 나누려 하자 오류 메시지가 출력된다.
- 처음 선언한 대로 sum 을 출력하면 4 가 출력 되고 sum=sum+i 를 실행한 후에는 5 가 출력된다.

```
2
3
4
5
6
7
-7
syntax error
---missing RIGHT
10.500000
9
10
```

- 마지막 수행 결과이다. 이는 while 문과 block 이 제대로 실행되는지 확인하기 위함인데, i=1 인상태에서 7 까지 계속 1 을 더했을때 정상적으로 출력된다.
- 또한 UMINUS 연산 또한 정상적으로 이루어지고
- block 이 닫히지 않으면 오류 메시지가 출력된다. 하지만 block 안의 statement 는 정상적으로 실행된다.
- 다음으로는 함수가 제대로 실행되는지 확인하기 위함인, func.mc 의 내용이다.

```
1 def f(a, b){
2     local max;
3     if(a>b)
4         max = a;
5     else
6         max = b;
7     return max;
8 }
```

```
10 def i(a, b, c){
11     count = a+b+c;
12     print count;
13 }
```

- 두 개의 함수 f 와 i 를 선언하였고,

```
15 r = f(18, 20);
16 i(1,2,3);
17 print r;
```

- 각각 함수를 호출한다.
- 실행결과는 아래와 같다.

```
bagsulin-ui-MacBook-Pro-6:hw3_201720723 surin$ ./mc func.mc
6
20
```

- f(18,20)을 통해 더 큰 값인 20 이 return 되어 r 에 저장되었고,
- I 함수는 return value 는 없지만 1,2,3 을 각각 parameter 로 넘겨주고 이를 모두 더해 함수 내에서 출력한다.
- 출력 순서대로 6 과 20 이 정상적으로 출력 되었음을 확인할 수 있다.