[Compiler]

Programming Project #2

- 수식 인터프리터 개발 -

Professor : 류기열 교수님

name : 박수린

student number : 201720723

목차

1. 서론

1.1. 과제 소개

- 이번 과제는 2 번째 programming project 로 지난 1 차 과제에서 구현하였던 lexical analyzer 을 활용하여 수식 interpreter 을 개발하는 것이다. 그 방법으로 recursive descent parsing 기법을 이용하였고 간단한 expression 들을 다루었다. expression 은 다음과 같다.
  - constants : integer, real number
  - variables : 1 차과제에서의 id 기준에 따른다.
  - binary operators : +, -, *, /
  - unary operators : -
  - assignment expression
  - 괄호 : (,)
- expression 의 문법 규칙은 제공된 문서를 참고하였고, 다음과 같다.
  - A -> idA' | F'T'E'
  - A' -> =A | T'E'
  - E' -> +TE' | -TE' | ε (- operator 을 위하여 -TE'을 추가하였다.)
  - T -> FT'
  - T' -> *FT' | /FT' | ε (/ operator 을 위하여 /FT'을 추가하였다.)
  - F -> id | F'
  - F' -> (E) | inum | fnum | -F

1.2. 구현된 부분과 구현되지 않은 부분

- interpreter 을 구현하기 위해서는 먼저 lexical analysis 가 필요한데 이는 1 차과제에서 구현하였던 어휘 분석기와 조교님이 제공해주신 예시 어휘 분석기를 활용하였다.
- 먼저 expression 을 프로시져별로 나누어서 recursive descent parser 을 개발하였고, syntax tree 를 생성하기 위해 return 을 struct node* 로 구현하였다.
- 또한 syntax tree 를 계산하는 evaluator 를 만들어 숫자만 입력하여 출력하거나, 가감승제 수식처리, assignment expression 을 이중으로 처리하는 것 또한 가능하게 했다. 그리고 음수를 나타내는 부호의 처리까지 해결하였다.
- 또한 과제 안내에는 정수와 정수끼리의 계산에도 실수로 출력하여도 무방하다고 하였으나, 정수끼리의 계산은 정수로 출력되도록 하였다.
- 다만, 이 mini 수식 interpreter 에서는 하나의 수식은 하나의 line 에 입력되는 것으로 가정하였다.
- 맨 처음에 prompt 를 출력하고 아무 입력도 없으면 error 가 나지 않고 다음 prompt 를 출력하도록 하였다.

2. 문제 분석

2.1. grammar rule 분석

- 처음에는 assign 이 고려되지 않은 grammar 였다,.

  A -> id = A | E

  E -> T E'

  E' -> + T E' | ε

  T -> F T'

  T' -> * F T' | ε

  F -> id | (E)

- 하지만 T 와 T'을 F 와, E 를 T 와 A 를 E 와 substituting 하였다.

  A -> id = A | id T' E' | (E) T' E'

  E' -> + T E‘ | ε

  T' -> * F T' | ε

  T -> F T'

  F -> id | (E)

- A 를 factoring 한 후, F'으로 확장시킨 후 정수, 실수를 추가한 최종 결과가 이번 과제에서 활용한 grammar 이다.

2.2. recursive-descent parsing 을 이용한 수식 계산기의 기본 개념정리

- syntax 를 분석하는 과정은 recursive 한 procedure 들의 집합이라고 볼 수 있다. 논터미널은 하나의 procedure 을 구성하고 A 의 문법은 procudure 의 정의이다. 논터미널 A 의 body 에 있는 다른 논터미널 B 는 B 에 해당하는 procedure 을 다시 불러온다. A 의 body 에 있는 터미널은 입력된 토큰에 match 됨을 뜻한다.
- syntax tree 를 구성하는 과정은 다음과 같다
  - F' -> (E) : input 이 ‘(‘일 때 expression_count 를 추가하고 E() precedure 을 호출한다. E()가 반환한 tree 는 그 다음 입력이 ‘)’일 때 F'이 반환한다. 그 다음 입력이 ‘)’로 matching 되지 않는다면 syntax_error 를 출력한다.
  - F'->inum : interger node 를 만든 후 expression_count 를 추가한 후 반환한다.

- F'->fnum : real node 를 만든 후 expression_count 를 추가한 후 반환한다.
- F'->-F : F 가 반환한 tree 를 왼쪽 자식 node 로 하는 – 노드를 만든 후 expression_count 를 추가한 후 트리를 반환한다.
- F->id : id 노트를 생성한 후 expression_count 추가 후 반환한다.
- F->F' : F'이 반환한 트리를 반환한다.
- E, E', T, T'은 강의노트의 내용을 참고하였다.

T->FT'	F가 반환한 tree를 T'의 parameter로 전달 rhs T'이 반환한 tree를 반환
T'->*FT'	parameter로 전달받은 tree와 F에서 반환된 tree를 두 child로 한 * node를 만들어 T'에 parameter로 전달 rhs T'이 반환한 tree를 반환
T' ->ε	Parameter로 전달받은 tree를 반환
E->TE'	T가 반환한 tree를 E'의 parameter로 전달 rhs E'이 반환한 tree를 반환
E'->+TE'	parameter로 전달받은 tree와 T에서 반환된 tree를 두 child로 한 + 노드 트리를 만들어 E'에 parameter로 전달 rhs E'이 반환한 tree를 반환
E'->ε	parameter로 전달받은 tree를 반환

이 내용에서 E'->-TE'와 T'->/FT'이 추가되었는데, 이도 마찬가지로 parameter 로 전달받은 tree 와 각각 T 와 F 에서 반환된 tree 를 자식 node 로 한 -, / node 를 만들어 E'과 T'에 전달 한 후 body 가 반환한 tree 를 return 한다.

- A'->=A : A 가 반환한 tree 와 A'의 parameter 을 자식으로 하는 = 노드 트리를 만들어 반환한다.
- A'->T'E' : A'의 parameter 을 T'에게 넘겨주고 T'의 반환 tree 를 E'의 parameter 로 하여 넘겨준다. 최종 반환 결과를 return 한다.
- A-> idA' : id 노드를 만들어 A'의 parameter 로 넘겨준 후, A'의 반환 결과를 return 한다.
- A->F'T'E' : F'이 반환한 결과를 T'의 parameter 로 넘겨주고, 마찬가지로 T'이 반환한 결과를 E'의 parameter 로 넘겨준 후 최종적으로 E'이 반환한 결과를 return 한다.

3. 설계

3.1. 주요 자료 구조

```
typedef struct {
        TOKEN token;
        char value[TOKEN_VALUE_MAX+1];
}TOKEN_LIST;

typedef struct {
        char symbol[SYMBOL_MAX+1];
        bool visit;
        double value;
}SYMBOL_TABLE;
```

TOKEN_LIST 는 예제 파일에서 제공된 것을 사용하였고 SYMBOL_TABLE 을 수정하였다. 정의되지 않은 변수를 사용할 때 error 을 출력하기 위해서 bool visit 이라는 멤버를 생성하였고, union 으로 되어있는 value 를 double 로 통일한 뒤, 정수에 대한 계산은 main()에서 핸들링하였다.

```
typedef struct node{
        int index;
        struct node* left;
        struct node* right;
        bool hasleft;
        bool hasright;
}NODE;
```

syntax_tree 를 생성하기 위하여 node 구조체를 선언해주었다. 멤버 index 는 이 node 가 token_list 에서 어느 token 을 가리키고 있는지를 알 수 있게 하고, tree 를 구성해야 하므로 자기 참조 구조체로 구현하였다. 또한 왼쪽자식과 오른쪽 자식이 있는지 확인할 수 있는 bool 변수를 두었다.
이는

```
NODE* syntax_tree;
```

의 형태로 main()함수 위에 선언해주었고,

```
syntax_tree = (NODE*)malloc(sizeof(NODE));
alloc(syntax_tree, -1, NULL, NULL, 0, 0);

 while(true)
 {
        syntax_tree = A();
        if(token_list[expr_count].token == 0)
                break;
 }
```

메인 함수 안에서 먼저 메모리를 할당해 준 후, index 는 -1, 자식 노드는 NULL 으로 초기화 해주었다. 그래서 syntax_tree 를 생성하는 프로시져 A()를 호출하고 반환된 값을 저장하였다. 그 line 의 토큰이 더 이상 없을 때 까지 parsing 하는 구조이다.

3.2. 프로그램 module hierarchy 및 module 에 대한 설명

```
int expr_count;
int flag;

NODE* A();
NODE* Aprime(NODE* n);
NODE* E();
NODE* Eprime(NODE* n);
NODE* T();
NODE* Tprime(NODE* n);
NODE* F();
NODE* Fprime();

double calculator(NODE* tree);
```

expr_count 변수를 통해 어디까지 token 을 검사했는지 체크할 수 있게 하였고, flag 를 이용해 error checking 을 하였다. error 의 타입을 구분하기 위해 int 형으로 선언하였다. 또한 각각의 논터미널에 대한 procedure 를 위해 함수를 구현하였고, 모두 반환형은 struct node* 타입이다. Aprime, Eprime, Tprime 은 2.2 의 syntax_tree 를 구성하는 방법에 대해 설명했다시피 parameter 가 필요하다. 또한 evaluator 를 구현하기 위해서 tree 를 parameter 로 받아 계산결과를 반환하는 calculator 함수를 생성하였다. 이에 대한 자세한 설명은 아래에 이어서 하도록 하겠다.

- main()

```
while(!feof(stdin)) {
    initialize_token_list();        // 토큰 리스트를 초기화 한다.
    printf(">");                    // 프롬프트를 출력한다.
    expr_count = token_number;
    flag = 0;
    isReal = false;
    do_lexical_analysis();          // 어휘 분석기를 호출한다.

    if(flag == 1){
        yyerror(1, "");
        continue;
    }
```

syntax_tree 가 구성되는 과정은 3.1.에서 설명하였다. 그 윗부분은 while 을 이용해 feof 가 아닐때까지 계속해서 실행하는데 먼저 token_list 를 초기화하고 prompt > 를 출력한 후 expr_count 를 현재까지 읽은 token_number 로 맞춰준다. 그리고 오류 검사를 할 flag 변수를 0 으로 (오류가 없음을 의미) 초기화 시켜준 후, 실수는 사용되지 않았다고 먼저 가정해둔다. 그리고나서 어휘분석을 하는데 flag=1 은 lexical error 를 의미한다. 어휘 오류가 있을 경우에는 syntax_tree 를 생성하지 않고 다음 loop 를 진행한다.

```
    if(token_number == 0)
        continue;
    else if(flag == 3)
        yyerror(flag, "");
    else
        result = calculator(syntax_tree);

    if(flag == 0){
        if(isReal)  printf("%lf\n", result);
        else        printf("%d\n",(int)result);
    }
}
```

그 아랫부분은 입력값이 없으면 다음 loop 를 실행하고 error 를 checking 하고 error 가 있을 시에는 yyerror 함수를 통해 error 타입을 출력한다. error 가 없으면 calculator 함수를 통해 수식 계산 결과를 출력한다. isReal 이라는 bool 형 변수를 통해 실수 값이면 그대로 출력하고 실수가 한번도 계산에 쓰이지 않았을 경우에는 이를 (int)로 형변환 하여 출력한다.

- yyerror(int type, char* e)

```
void yyerror(int type, char* e)
{
    switch(type)
    {
        case 1 : printf("error : lexical error\n");  break;
        case 2 : printf("error : %s는 정의되지 않음\n", e); flag = 2; break;
        case 3 : printf("error : syntax error\n");  break;
    }
}
```

타입별로 에러 내용을 출력하며, 특히 정의되지 않은 변수 error 에 대해서는 변수명도 함께 출력한다.

- A();

```
NODE* A()
{
    if(token_list[expr_count].token == ID)
    {
        NODE* new;
        new = (NODE*)malloc(sizeof(NODE));
        alloc(new, expr_count, NULL, NULL, 0, 0);
        expr_count++;
        return Aprime(new);
    }
    else{
        NODE* n = Fprime();
        NODE* m = Tprime(n);
        return Eprime(m);
    }
}
```

procedure 에 관한 함수들은 대체로 위에서 설명한 내용이고 구조가 비슷하기 때문에 대표적으로 A() 프로시져에 대해서만 설명하도록 하겠다. 다음 input 을 보았을 때 token_list 에 저장된 token 의 타입이 id 일 경우에는 새로운 node 를 생성하고 메모리를 할당한다. 그리고 id 노드는 자식이 없기에 NULL 값과 hasleft, hasright 멤버변수를 0 으로 초기화 해준다. 그리고 다음 input 을 읽기 위해서 expr_count 를 1 더해준 후 Aprime() 프로시져를 호출하면서 parameter 로 방금 만든 id 노드를 전달한다. 그리고 Aprime()프로시져가 끝난 후 반환된 tree 가 최종적으로 반환된다. 반면, 다음 input 이 id 가 아닐 경우 Fprime()프로시져를 호출한 후 그 반환된 tree 를 NODE* n 에 임시로 저장하고, 그 n tree 를 Tprime()의 parameter 로 넘겨준다. 마찬가지로 Tprime 이 반환한 tree 를 m 에 임시로 저장하고, 이를 Eprime()의 parameter 로 넘겨준다. 최종적으로 Eprime()이 반환한 tree 를 반환하면 A()프로시져가 끝나기때문에 이는 syntax_tree 에 저장된다.

- calculator(NODE* tree)

```
double calculator(NODE* tree)
{
    int temp;
    switch(token_list[tree->index].token)
    {
        case INT :  return atoi(token_list[tree->index].value);
        case REAL : isReal = true; return atof(token_list[tree->index].value);
        case PLUS : return calculator(tree->left) + calculator(tree->right);
        case MINUS :
            if(tree->hasright == 0 && tree->hasleft == 1)   return calculator(tree->left)*(-1);
            else    return  calculator(tree->left) - calculator(tree->right);
        case MUL :  return calculator(tree->left) * calculator(tree->right);
        case DIV :  return calculator(tree->left) / calculator(tree->right);
```

다음으로는 calculator 함수이다. tree 전체를 parameter 로 입력받아 node 의 타입에 따라 행동이 결정된다. tree 의 멤버 index 가 가리키고 있는 token_list 에 저장된 값을 통해 그를 판단한다. 숫자, 즉 정수와 실수일 경우에는 token_list 의 value 를 atoi(), atof()를 이용해 string 에서 숫자로 변환한다. 사칙연산의 경우에는 왼쪽 자식 tree 와 오른쪽 자식 tree 가 반환된 결과를 각각 recursive 하게 계산한다. 주목해야할 점은 'MINUS'에 관해서 이다. MINUS 는 unary operator 로도 작용하기 때문에 만약 왼쪽 자식만 있고 오른쪽 자식이 없다면 unary operator 라고 판단을 하여 -의 왼쪽 자식이 가지고 있는 값에 (-1)을 곱하여 반환해준다. 그 이외의 경우에는 평범한 사칙연산과 동일하게 작동한다.

```
case ASSIGN :
    temp = find_symbol(token_list[tree->right->index].value);
    if(temp != -1){
        symbol_table[temp].visit = true;
        symbol_table[temp].value = calculator(tree->left);
    }
    return calculator(tree->right);
case ID :
    temp = find_symbol(token_list[tree->index].value);
    if(symbol_table[temp].visit == false){
        yyerror(2, token_list[tree->index].value);
    }
    return symbol_table[temp].value;
case LP : break;
case RP : break;
default : break;
```

Assign 은 먼저 오른쪽 자식의 Index 가 어떤 symbol 을 가리키고 있는지, symbol_table 에서 index 를 찾은 후 temp 변수에 임시로 저장한다. symbol_table 에서 이를 찾았다면 그 symbol_table 에 id 의 값을 왼쪽 자식 tree 가 계산된 결과로 저장해주고, visit 멤버를 true 로 바꾼다. 이는 ID 에서 마찬가지로 index 가 어떤 symbol 을 가리키고 있는지 찾았을때, 값이 저장되어 있지 않다면(즉 visit 이 false 라면) 처음 선언된 것이므로 error type2 를 출력하게 한다. 그렇지 않은 경우에는 symbol_table 에서 value 을 찾아 반환한다. 왼쪽 괄호와 오른쪽 괄호와 같은 경우에는 계산에 지장을 주지 않으므로 calculator 에서 역할이 없어 break; 시켰다.

4. 수행 결과

```
bagsulin-ui-MacBook-Pro-6:hw2_201720723 surin$ ./hw2
>
>
>10 + 5
15
>10
10
>val = 10
10
>val + 10
20
>i = j = 20
20
>val = val + i
30
>(val+val)*3
180
>-val-100
-130
>abc+10
error : abc는  정 의 되 지  않 음
>val + + i
error : syntax error
>val << 10
error : lexical error
>
```

먼저 과제 문서에 예시로 제공된 입력 형식과 실행 결과를 만족하였음을 알 수 있다. 변수에 값을 저장하고 이중으로 assign 되며 왼쪽 value 의 값을 출력한다. 또한 정수 끼리의 계산은 결과가 정수로 출력된다.

아래는 또 다른 예시이다.

```
bagsulin-ui-MacBook-Pro-6:hw2_201720723 surin$ ./hw2
> a = b = 10
10
>a
10
>b
10
>c = 15
15
>(a+b)*c
300
>a*b+c
115
>9/4
2
>9/4.0
2.250000
>-5*3
-15
>--15
15
>4^2
error : lexical error
>a+d
error : d는  정 의 되 지  않 음
>((a+b)
error : syntax error
```

a 와 b 에 차례대로 assign 된 것을 확인할 수 있고, 3 개의 variables 도 계산이 가능하다. 또한 9 를 4 로 나누었을 때랑 4.0(실수)로 나누었을 때의 차이를 보여준다. unary operator 의 활용 또한 보여지는데 -5*3 의 계산결과가 정상적이고 - - 15 의 형태로 unary operator 을 두번 입력했을때 사라지는 것을 볼 수 있다. 그리고 정의되지 않은 lexeme 을 활용했을 때의 에러 처리와 Syntax error 의 예시도 하나 더 추가해보았다.