컴파일러 2차 과제 보고서

소프트웨어학과 201620908 편동혁

담당 교수 : 류기열

1. 서론

1.1 과제 소개

2차 과제는 Syntax Analyzer를 이해하기 위한 과제로 이전 1차 과제에서 수행한 Lexical analyzer의 tokenize와 lex를 활용하여 수식 interpreter를 개발하는 것에 목적을 둔다.

Basic recursive descent parsing 기법을 활용하여 regular expression을 다룬다.

Expression은 다음과 같다.

Integer : 정수형,

double : 실수형 ,

identifier : 1차 과제에서 다룬 id,

operator : +, -, /, *, =

unary operator : -

assignment expression

괄호에 해당하는 기호 : (,)

1.2 구현된 부분과 구현되지 않은 부분

Interpreter를 위해 lexical analyzer의 구현이 필요했고 이부분은 lex와 강의노트의 어휘 분석기의 예시를 활용하여 사용했다.(이전 1차과제에서 만든 어휘 분석기는 문제가 많았기에 사용 x)

먼저 basic recursive descent Parser를 구현하고 syntax tree를 생성하기 위해 struct Tree*라는 자료형을 사용했다.

실수와 실수의 계산에는 실수를 반환하도록 하였고 정수는 마찬가지로 정수를 반환하게 구현했다.

하나의 수식은 하나의 line에 입력하는 것으로 가정을 두었다.

다만 음수의 경우 제대로 계산이 되지 않으므로 이부분은 구현을 하지 못했다고 할 수 있다.

또한 예시의 String에 대한 연산 역시 구현하지 못한 부분이다.

2. 문제 분석

2.1 grammar 분석

기본적으로 grammar rule은 다음과 같다.

A -> id = A | E

E -> TE′

E′ -> +TE′ | ε

T -> F T′

T′ -> *FT′|ε

F -> id|(E)

그렇지만 ASSIGN에 해당하는 경우를 고려하기 위해 rule을 수정하였고 수정한 결과는 다음과 같다.

A -> id = A| id T′E′| (E)T′E′

E′ -> +TE′ | ε

T -> F T′

T′ -> *FT′|ε

F -> id|(E)

3. 설계

3.1 주요 자료 구조

```
typedef struct SymbolTable {
    char symbol[17];
    double symbolValue;
    bool isVisited;
} SymbolTable;
```

Symbol table의 경우 error 출력을 위해 isVisited라는 변수를 사용했다.

또한 union으로 구성되어 있는 value를 double로 일치시킨 후 , 정수는 main에서 해결했다.

```
typedef struct Token {
    tokenType token;
    char tokenValue[11];
}Token;
```

Token은 예제 파일에서 제공된 부분을 이용했다.

```
typedef struct Tree {
    int idx;
    struct Tree* leftChild;
    struct Tree* rightChild;
}Tree;
```

Tree는 Parse Tree를 생성하고 왼쪽 자식과 오른쪽 자식을 구성하기 위해 leftChild, rightChild를 사용했다.

```
Tree* grammarA();
Tree* grammarAprime(Tree* rightChild);
Tree* grammarE();
Tree* grammarEprime(Tree* leftChild);
Tree* grammarT();
Tree* grammarTprime(Tree* leftChild);
Tree* grammarF();
Tree* grammarFprime();
double interpreter(Tree* parseTree);

int count;
Tree* parseTree;
double value;
int checkDouble;
```

Count 변수를 통해 어디까지 token을 검사했는지 확인할 수 있게 하고 각각의 grammar는 procedure를 위해 구현한 함수이다. 반환형은 모두 Tree로 통일 시켰다.

위 grammar는 문법 분석에서 설명한 내용에 맞춰 구현했다. 또한 parameter가 필요했으므로 parameter 역시 건네 받는 자식 node의 타입인 Tree로 일치 시켰다. interpreter함수는 계산된 결과를 반환하는 함수이다.

```
while(!feof(stdin)){
    for(int i = 0; i < 100; i++) {
        token[i].token = 0;
        token[i].tokenValue[0] = '\0';
    }
    printf(">");
    errorCheck = 0;
    count = token_num;
    yyin = stdin;
    checkDouble = 0;
    yylex();
    yyrestart(yyin);
    if(errorCheck == -1) {
        printf("lexical error 발생\n");
        break;
    }
    parseTree = (Tree*)malloc(sizeof(Tree));
    parseTree->idx = -1;
    parseTree->leftChild = NULL;
    parseTree->rightChild = NULL;
    while(true) {
        parseTree = grammarA();
        if(parseTree == NULL) {
            break;
        }
        if(token[count].token == 0) {
            break;
        }
    }
```

해당 문구에서 parse tree를 생성한다. 또한 파일 상단에 선언한 errorcheck 변수를 통해 오류를 검사하고 yylex를 통해 어휘 분석을 실행한다. 이후 parseTreee를 생성하고 grammar 함수를 통해 Tree를 채워나간다. 오류가 있을 경우 while문을 탈출시킨다.

```
    if(token_num == 0) {
        continue;
    } else if(errorCheck == 3){
        printf("syntax error 발생!\n");
        break;
    } else {
        value = interpreter(parseTree);
    }
    if(errorCheck == 0) {
        if(checkDouble == 0)
            printf("%d\n",(int)value);
        else printf("%lf\n",value);
    }
}
```

입력값이 없으면 다음 loop를 실행하고 error를 체크한다. Error가 없다면 interpreter함수를 통해 계산 결과를 출력한다. checkDouble은 double 타입의 결과인지 확인하기 위해 만든 변수이다.

```
Tree* grammarA(){
    if(token[count].token != ID) {
        Tree* first = grammarFprime();
        Tree* second = grammarTprime(first);
        return grammarEprime(second);
    } else {
        Tree* id = (Tree*)malloc(sizeof(Tree));
        id->idx = count;
        id->leftChild = NULL;
        id->rightChild = NULL;
        count++;
        return grammarAprime(id);
    }
}
```

해당부분은 시작하는 grammar인 A procedure이다. 다음 input을 보았을 때 token에 저장된 타입에 따라 tree 변수를 생성하고 메모리를 할당하게 했다.

Id tree는 기본적으로 자식이 없기에 null를 이용하여 초기화시켰다.

다음 input을 읽기 위해서 count를 더해주고 Aprime을 호출과 동시에 parameter로 방금 만든 id를 전달한다. Aprime이 끝나면 반환된 tree가 최종적인

결과가 된다.

다음 input이 id가 아니면 Fprime을 호출하고 반환된 tree를 임시 저장하고 그 임시저장한 tree를 Eprime으로 넘겨준다. 최종적으로 Eprime이 반환한 tree가 parse tree에 저장된다.

```
double interpreter(Tree* parseTree) {
    int temp;
    switch(token[parseTree->idx].token) {
        printf("%s",token[parseTree->idx].tokenValue);
        case PLUS :
            return interpreter(parseTree->leftChild) + interpreter(parseTree->rightChild);

        case MINUS :
            if(parseTree->rightChild == NULL && parseTree->leftChild != NULL){
                return interpreter(parseTree->leftChild)*(-1);
            } else {
                return interpreter(parseTree->leftChild) - interpreter(parseTree->rightChild);
            }

        case MUL :
            return interpreter(parseTree->leftChild) * interpreter(parseTree->rightChild);

        case DIV :
            return interpreter(parseTree->leftChild) / interpreter(parseTree->rightChild);

        case INT :
            temp = atoi(token[parseTree->idx].tokenValue);
            return temp;

        case DOUBLE :
            checkDouble = 1;
            return atof(token[parseTree->idx].tokenValue);
```

Interpreter 함수는 최종 tree에 맞춰 계산하기 위한 함수이다. Tree의 타입에 따라 계산 결과가 달라지며 정수 실수는 atoi atof로 처리했다.

4. 수행 결과

```
>10 + 2
12
>10
10
>val = 5
5
>val + 20
25
>i = j = 15
15
>(val+val)*4
40
>abc+10
error : abc는 정의되지 않음
>
```