컴파일러 프로젝트 보고서

날짜 : 2021.06.06.

소프트웨어학과 3학년

2017038064 김동용

목차

-토큰 종류-

1. 사칙연산

2. 괄호 우선순위

3. if문

- 추가 토큰 종류 -

```
add     {return(ADD); }
sub     {return(SUB); }
mul     {return(MUL); }
div     {return(DIV); }
"("     {return(LPARA); }
")"     {return(RPARA); }
"{"     {return(LP); }
"}"     {return(RP); }
"-"     {return(MINUS); }
if      {return(IF_ST); }
">"     {return(GT); }
"<"     {return(LT); }
">="    {return(GE); }
"<="    {return(LE); }
"=="    {return(EE); }
"!="    {return(NE); }
:=      {return(ASSGN); }
;       {return(STMTEND); }
start       {return(START);}
end     {return(END);}
```

1) 사칙연산

edragon.l

```
add     {return(ADD); }
sub     {return(SUB); }
mul     {return(MUL); }
div     {return(DIV); }
```

add(더하기), sub(빼기), mul(곱하기), div(나누기) 각 토큰을 l파일에서 선언하였습니다.

edragon.y

```
%token  ADD SUB MUL DIV
```

definition에서 토큰을 선언하였습니다.

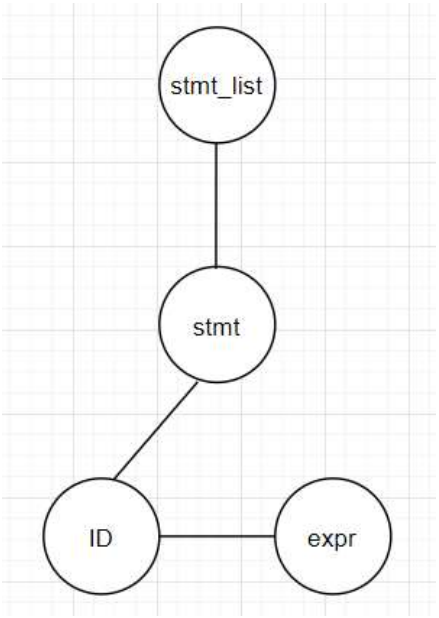
```
%right ASSGN
%left ADD SUB
%left MUL DIV
```

마찬가지로 우선순위를 선언하여 Yacc 충돌 해결 규칙을 사용하였고 shift/reduce 충돌에서, 우선된 문법 규칙에 따라 reduce를 실행하도록 MUL, DIV를 가장 높은 우선순위로 왼쪽부터 결합법칙을 사용할 수 있도록 하였습니다. 그 다음 우선순위로 ADD, SUB를 정하였고 마찬가지로 왼쪽부터 결합법칙을 사용할 수 있도록 하였습니다. ASSGN은 가장 낮은 우선순위로 오른쪽부터 결합법칙을 사용할 수 있도록 하였습니다.

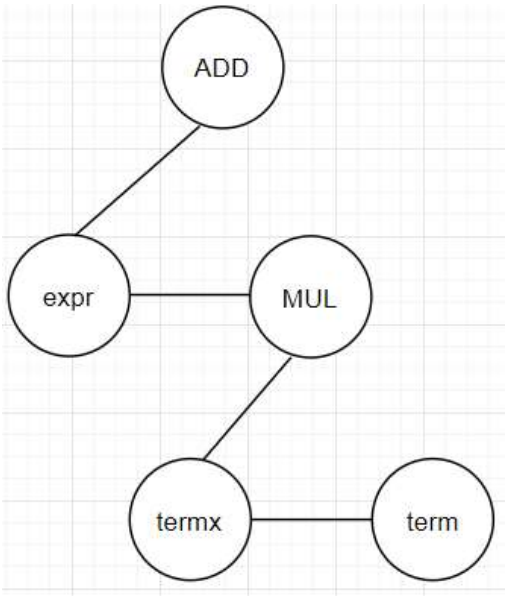
```
program : START stmt_list END   { if (errorcnt==0) {codegen($2); dwgen();} }
        ;

stmt_list:  stmt_list stmt  {$$=MakeListTree($1, $2);}
        |   stmt            {$$=MakeListTree(NULL, $1);}
        |   error STMTEND   { errorcnt++; yyerrok;}
        ;

stmt    :   ID ASSGN expr STMTEND   { $1->token = ID2; $$=MakeOPTree(ASSGN, $1, $3);}
expr    :   expr ADD termx  { $$=MakeOPTree(ADD, $1, $3); }
        |   expr SUB termx  { $$=MakeOPTree(SUB, $1, $3); }
        |   termx
        ;

termx   :   termx MUL term  { $$=MakeOPTree(MUL, $1, $3); }
        |   termx DIV term  { $$=MakeOPTree(DIV, $1, $3); }
        |   term
        ;

term    :   LPARA expr RPARA { $$=$2; }
        |   MINUS term       { $$=$2; }
        |   ID      { /* ID node is created in lex */ }
        |   NUM     { /* NUM node is created in lex */ }
        ;
```

사칙연산에서의 rule은 다음과 같이 구현하였고 expr에서 termx로 파싱하는 것이 특징입니다.

$$=MakeOPTree(ADD, $1, $3) 함수를 통해 다음과 같은 그림의 AST 를 만들 수 있습니다.



먼저, stmt_list부터 파싱하면 ID = expr이라는 문장의 형태는 다음과 같습니다. 이때, expr을 파싱하면 아래 그림과 같습니다.



이 트리는 expr + termx x term의 문장을 파싱한 형태로 이 그림과 같이 MUL, DIV가 먼저 계산되어 expr과 ADD하여 사칙연산의 법칙을 따르는 것을 확인할 수 있습니다. 이때, termx, expr는 또 다시 termx MUL term, term으로 파싱할 수 있으며 앞뒤에 따라오는 모든 문장의 형태를 이 rule을 통해 나타낼 수 있다는 사실을 확인할 수 있습니다.

expr또한, expr ADD termx, expr ADD termx MUL term, expr SUB termx ADD termx MUL term 등 모든 문장을 나타낼 수 있습니다.

이렇게 AST를 구축하여 dfs함수를 실행하게 되면 son, 본인, brother 순으로 토큰을 출력하게 되고 이 토큰들은 a.asm에 순서대로 출력합니다. a.asm에 있는 이 가상스택기계의 어셈블리어를 asm.exe를 통해 컴퓨터에 명령하게 되고 이 명령들을 수행하면 저희가 원하는 올바른 값들을 도출할 수 있습니다

실행결과

예제1

```
start
가:=2;
A:=4;
나:=가 add A sub 2;
다:= 나 mul 4 sub 40 div A add 나;
end
```

나:= 가 add A sub 2 = 4

다:= 나 mul 4 sub 40 div A add 나 = 16 - 10+4 = 10

```
                LVALUE 다
                RVALUE 나
                PUSH 4
                *
LVALUE 나       PUSH 40
RVALUE 가       RVALUE A
RVALUE A        /
+               -
PUSH 2          RVALUE 나
-               +
:=              :=
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                2
   1  A                 4
   2  나                4
   3  다               10
[End of Dump]
```

예제 2

```
start
가:=2;
A:=4;
나:=A mul A sub 2 div 가;
다:= 가 add 4 sub 40 div A add 나;
end
```

나:= A mul A sub 2 div 가 = 15

다:= 가 add 4 sub 40 div A add 나 = 6 - 10 + 15 = 11

```
                LVALUE 다
                RVALUE 가
LVALUE 나       PUSH 4
RVALUE A        +
RVALUE A        PUSH 40
*               RVALUE A
PUSH 2          /
RVALUE 가       -
/               RVALUE 나
-               +
:=              :=
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                2
   1  A                 4
   2  나               15
   3  다               11
[End of Dump]
```

예제3

```
start
가:=2;
A:=4;
나:=5;
다:= 가 div 4 mul 40 div A add 나;
end
```

다 :=가 div 4 mul 40 div A add 나 = 5

```
LVALUE 다
RVALUE 가
PUSH 4
/
PUSH 40
*
RVALUE A
/
RVALUE 나
+
:=
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                2
   1  A                 4
   2  나                5
   3  다                5
[End of Dump]
```

2) 괄호 우선순위

edragon.l

```
"("     {return(LPARA); }
")"     {return(RPARA); }
```

edragon.y

```
RPARA LPARA
```

위 사진들과 같이 y, l 파일에서 괄호 토큰을 선언해주었습니다.

```
term    :   LPARA expr RPARA { $$=$2; }
        |   MINUS term       { $$=$2; }
        |   ID      { /* ID node is created in lex */ }
        |   NUM     { /* NUM node is created in lex */ }
        ;
```

괄호는 위 사진과 같이 yacc파일의 rule에서 term으로 정의해주었고 term에서 '(expr)'으로 파싱하여 괄호 안의 expr이 먼저 연산되도록 구현하였습니다.

실행결과

예제 1

```
start
가:=2;
A:=4;
나:=5;
다:= 가 add 4 mul 40 div (A add 나);
end
```

다:= 가 add 4 mul 40 div (A add 나) = 2+160/9= 19

```
LVALUE 다
RVALUE 가
PUSH 4
PUSH 40
*
RVALUE A
RVALUE 나
+
/
+
:=
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                2
   1  A                 4
   2  나                5
   3  다               19
[End of Dump]
```

예제2

```
start
가:=2;
A:=4;
나:=5;
다:= (가 add 4) mul (40 div A) add 나;
end
```

다:= (가 add 4) mul (40 div A) add 나 = 65

```
LVALUE 다
RVALUE 가
PUSH 4
+
PUSH 40
RVALUE A
/
*
RVALUE 나
+
:=
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                2
   1  A                 4
   2  나                5
   3  다               65
[End of Dump]
```

예제3

```
start
가:=2;
A:=4;
나:=5;
다:= 가 mul 4 sub (40 sub (A add 나));
end
```

다:= 가 mul 4 sub (40 sub (A add 나)) = 8 - 31 = -23

```
LVALUE 다
RVALUE 가
PUSH 4
*
PUSH 40
RVALUE A
RVALUE 나
+
-
-
:=
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                2
   1  A                 4
   2  나                5
   3  다              -23
[End of Dump]
```

3) if 문

edragon.l

```
"{"     {return(LP); }
"}"     {return(RP); }
"-"     {return(MINUS); }
if      {return(IF_ST); }
">"     {return(GT); }
"<"     {return(LT); }
">="    {return(GE); }
"<="    {return(LE); }
"=="    {return(EE); }
"!="    {return(NE); }
```

edragon.y

```
IF_ST LT GT LE GE EE NE label LP RP
```

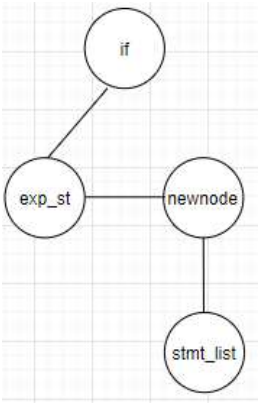
위 사진과 같이 토큰을 정의하였고 아래 사진과 같이 yacc파일의 if문 rule을 정의하였습니다.

```
stmt    :   ID ASSGN expr STMTEND   { $1->token = ID2; $$=MakeOPTree(ASSGN, $1, $3);}
        |   IF_ST LPARA exp_st RPARA LP stmt_list RP    {$$=MakeOPTree(IF_ST, $3, label_node($6)); }
        ;

exp_st  :   expr GT expr    { $$=MakeOPTree(GT, $1, $3); }
        |   expr LT expr    { $$=MakeOPTree(LT, $1, $3); }
        |   expr GE expr    { $$=MakeOPTree(GE, $1, $3); }
        |   expr LE expr    { $$=MakeOPTree(LE, $1, $3);}
        |   expr EE expr    { $$=MakeOPTree(EE, $1, $3);}
        |   expr NE expr    { $$=MakeOPTree(NE, $1, $3);}
```

stmt에 if문 rule을 추가하여 stmt에 배정문, if문이 올 수 있도록 구현하였습니다.
이 rule은 if(condition){stmt_list} 형식의 문장으로 $$=MakeOPTree(IF_ST, $3, label_node($6)); 함수를 사용하여 AST를 구현하였습니다. 중괄호 안에 stmt_list를 넣어 여러 stmt가 if문에 포함될 수 있도록 구현하였고 지금까지의 다른 문장과는 다르게 label_node($6) 라는 함수를 따로 만들어 넣었습니다.

```
Node* label_node(Node* operand1){
    Node * newnode;

    newnode = (Node *) malloc(sizeof (Node));
    newnode->token = label;
    newnode->tokenval = label;
    newnode->son = operand1;
    newnode->brother = NULL;
    return newnode;
}
```

이 함수는 Node* 타입을 리턴하여 newnode와 exp_st를 연결하고



stmt_list를 인자로 하여 newnode->son으로 stmt_list를 연결합니다.
이렇게 연결해주는 이유는 if문이 종료되는 시점에 탈출문인 LABEL memloc을 asm파일에 출력하기 위해서입니다. 이 프로그램의 dfs을 따라가보면 newnode가 가장 나중에 출력되며 if문이 종료되는 시점 즉, 맨 마지막에 출력하기 위해서는 위 그림과 같이 연결해주는 방법이 있습니다. 따라서 이 방법으로 LABEL 출력을 구현하였습니다.

또한, if문에서 condition에 사용되는 >, <, >=, <=, !=, == 문장을 넣어주기 위해 exp_st를 따로 정의하여 부등호 양쪽에 expr이 올 수 있도록 구현하였습니다.

```
case GT:
    fprintf(fp, "-\nGOMINUS memloc\n");
    break;
case LT:
    fprintf(fp, "-\nGOPLUS memloc\n");
    break;
case GE:
    fprintf(fp, "-\nGOMINUS memloc\nGOFALSE loc\nLABEL loc\n");
    break;
case LE:
    fprintf(fp, "-\nGOPLUS memloc\nGOFALSE loc\nLABEL loc\n");
    break;
case EE:
    fprintf(fp, "-\nGOTRUE memloc\n");
    break;
case NE:
    fprintf(fp, "-\nGOFALSE memloc\n");
    break;
case label:
    fprintf(fp, "LABEL memloc\n");
    break;
```

asm에서는 부등호 기호를 출력하면 invalid token라는 오류가 발생합니다. 따라서 asm 전용 명령어를 사용해야 하며 GT 즉, >를 예시로 들면 a>b에서 a-b의 값이 -이면 즉, a가 b보다 작으면 if문을 탈출해야 합니다. 따라서 prtcode함수에서 위 사진과 같이 작성하여 조건에 맞지 않으면 바로 if문을 탈출할 수 있도록 구성하였습니다.

특이하게 GE 즉, >=의 경우 a>=b에서 a-b가 음수이면 탈출하고 a-b가 0이면 즉, 같으면 바로 뒤에 있는 LABEL loc으로 점프하여 정상적으로 if문을 수행하도록 구성하였습니다.

또한, EE ==의 경우 a-b가 0이 아니면 GOTRUE이면 memloc으로 점프하도록 구성하였고 NE의 경우 반대로 GOFALSE일 때 점프하도록 구현하였습니다.

실행결과

예제1

```
start
가:=2;
A:=4;
나:=5;
if(나>A){
    다:=나 sub 가;
    라:=가 mul A;
}
마:=A add 다;
end
```

나(5)>A(4)이므로 정상적으로 동작

```
RVALUE 나
RVALUE A
-
GOMINUS memloc
LVALUE 다
RVALUE 나
RVALUE 가
-
:=
LVALUE 라
RVALUE 가
RVALUE A
*
:=
LABEL memloc
LVALUE 마
RVALUE A
RVALUE 다
+
:=
```

위 가상스택기계에서의 코드와 같이 나-A를 수행하여 음수가 아니므로 정상적으로 아래 문장들을 실행하고 if문의 마지막에 LABEL memloc을 선언하여 if문을 탈출할 수 있도록 구현하였습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                2
   1  A                 4
   2  나                5
   3  다                3
   4  라                8
   5  마                7
[End of Dump]
```

예제2

```
start
가:=4;
A:=4;
나:=5;
if(나!=A){
    나:=A add 가 mul A;
}
마:=A add 다;
end
```

나:=A add 가 mul A = 20

```
RVALUE 나
RVALUE A
-
GOFALSE memloc
LVALUE 나
RVALUE A
RVALUE 가
RVALUE A
*
+
:=
LABEL memloc
LVALUE 마
RVALUE A
RVALUE 다
+
:=
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol        Value
   0  가                4
   1  A                 4
   2  나               20
   3  마                4
   4  다                0
[End of Dump]
```

예제3

```
start
가:=4;
A:=4;
나:=5;
if(가>=A){
    나:=나 mul (가 add A);
}
end
```

가(4)>=A(4) 이므로 나=5 * 8=40

```
LVALUE 가
PUSH 4
:=
LVALUE A
PUSH 4
:=
LVALUE 나
PUSH 5
:=
RVALUE 가
RVALUE A
-
GOMINUS memloc
GOFALSE loc
LABEL loc
LVALUE 나
RVALUE 나
RVALUE 가
RVALUE A
+
*
:=
LABEL memloc
```

실제 asm파일의 가상스택기계 코드를 보면 올바른 결과가 출력되고 있다는 것을 알 수 있습니다.

```
[DATA Location Dump]
Loc#  Symbol          Value
   0  가                  4
   1  A                   4
   2  나                 40
[End of Dump]
```