

컴파일러 기말프로젝트

부탁어(명령어 **X**) 컴파일러

강지원
2018038013

0. 서론	3
1. Entreaty language 소개	3
2. 설계 및 구현하면서 가장 고민했던 점	5
3. 설계 및 구현	7
4. 결과	25
if-else	25
5. 토론 및 결론	34

0. 서론

우리는 여태까지 코드에 명령어를 쓰고 컴파일러는 그 명령어를 실행해왔다.

하지만 인공지능이 발달함에 따라 언젠가 컴파일러도 의식을 갖는 날이 오지않을까?

그렇게 된다면 명령조로 말하는것이 기분이 나쁠수도 있지 않을까? 라는 생각에서
난 부탁어 컴파일러를 만들었다.

내가 만든 문법은 모두 컴파일러에게 부탁하도록 되어있다.

1. Entreaty language 소개

C	Entreaty language
+ - * /	더하기 빼기 곱하기 나누기
++ --	++ --
+= -= *= /=	+= -= *= /=
if (a > b) { max = a; }	만약 a가 b보다 크다면 여기부터 max에 a를 저장해주세요 여기까지 실행 해주세요
if (a > b) { max = a; } else { max = b; }	만약 a가 b보다 크다면 여기부터 max에 a를 저장해주세요 여기까지 실행 해주세요 아니라면 여기부터 max에 b를 저장해주세요 여기까지 실행 해주세요
while (a < 10) { a++; }	a가 10보다 작다면 여기부터 a++ 해주세요 여기까지 반복 해주세요
for(i=0; i<10; i++){ a++; }	일단 i에 0을 저장해주세요 i가 10보다 작다면 마지막에 i++하면서 여기부터 a++ 여기까지 반복 해주세요
max = (a > b) ? a : b;	max에 a가 b보다 크다면 a를 저장해주세요 아니라면 b를 저장해주세요

C	Entreaty language
temp=a; a=b; b=temp;	a와 b의 값을 서로 바꿔주세요
printf("%d", a);	a의 값을 출력 해주세요
scanf("%d", &a);	a의 값을 입력 받아주세요
void func_abc { /* 함수 내용*/ } ... func_abc(); //호출	이름이 func_abc 이고 내용은 여기부터 /* 함수 내용*/ 여기까지인 함수를 선언해주세요 ... func_abc를 호출해주세요
switch (a) { case 1: b=1; break; case2: b=2; break; default: b=3; break; }	아래 경우 중 a가 1일 경우: a에 1을 저장해주세요 그리고 나가주세요 2일 경우: a에 149를 저장해주세요 그리고 나가주세요 다 아닌경우: b에 3을 저장해주세요 그리고 나가주세요

2. 설계 및 구현하면서 가장 고민했던 점

Entreaty language 문법의 설계 및 구현을 설명하기 전에 문법을 만들면서 가장 고민했던 점을 말씀드리고 가겠습니다. 문법에서 조건문에 따라 실행 위치가 달라지는 즉 LABEL을 사용하는 문법을 만들때 문법 자체를 만드는것은 어렵지 않았지만 이들을 중첩해서 여러개를 한번에 실행했을 때 문제가 발생했습니다. 이는 LABEL명이 서로 중복되면서 발생한 문제였습니다.

예를 들어 가장 밖에 if문이 있고 그 if문 안에 여러개의 중첩된 if문이 있다고 했을 때 단순히 LABEL의 cnt를 1씩 더하는 방법으로는 정상적으로 코드가 실행되지 않았습니다. 이 문제의 해결책은 어떤 if문의 내부에 if가 몇개가 있는지 알 수있으면 해결할수있었습니다. 따라서 저는 DFS를 순회할때 IF노드를 만나면 DFS탐색을 활용한 아래 사진의 함수를 사용해서 그 IF노드의 내부의 IF노드 개수를 센 뒤에 나중에 가장 밖의 IF노드의 LABEL을 찍어줄때 안에 있는 IF노드의 개수만큼 빼준뒤에 LABEL을 찍어주면서 해결해주었습니다.

```
int count_if_nodes(Node *n) {
    if (n == NULL) {
        return 0;
    }

    int count = 0;

    if (n->token == IF) {
        count++;
    }

    count += count_if_nodes(n->son);
    count += count_if_nodes(n->brother);

    return count;
}
```

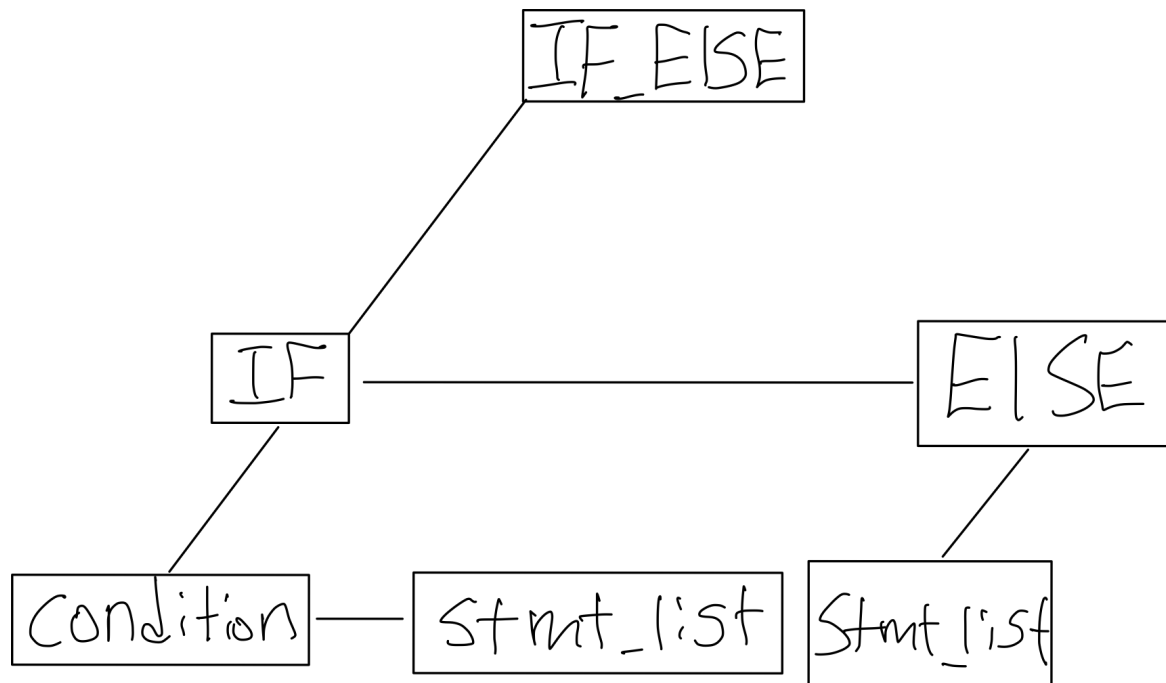
저의 모든 LABEL을 사용하는 문법에서는 이와 같은 함수를 사용하여 여러개를 중첩해서 코드를 써도 LABEL명에 문제가 생기지 않도록 만들었습니다.

또한 아래와 같은 함수를 사용해 어디의 조건문인지에 따라 다른 LABEL명을 갖도록 하였습니다.

```
int if_cond_check(int token)
{
    if (token == EQ) {
        return IF_EQ;
    }
    if (token == LT) {
        return IF_LT;
    }
    if (token == LE) {
        return IF_LE;
    }
    if (token == GT) {
        return IF_GT;
    }
    if (token == GE) {
        return IF_GE;
    }
    if (token == NE) {
        return IF_NE;
    }
}
```

3. 설계 및 구현

if-else



우선 위와 같은 형태로 문법을 설계했습니다. DFS탐색 순서에 따라 실행되는 순서는 condition -> stmt_list -> IF -> ELSE -> stmt_list -> IF_ELSE입니다.

따라서 조건을 확인하고 조건이 참이라면 안의 stmt_list를 실행하고 아니라면 IF에서 찍어줄 LABEL로 점프합니다. 그리고 ELSE의 stmt_list를 실행하고 IF_ELSE 노드에서 전체 IF_ELSE에 대한 LABEL을 찍어줍니다. 이 LABEL로 이동하는 경우는 IF가 참이고 stmt_list를 실행한후에 바로 밖으로 빠져나가야하기때문에 GOTO를 만나 이 LABEL로 빠져나오게 됩니다. 여기서 GOTO문을 IF노드에서 IF에 대한 LABEL을 찍어주기전에 만들기위해 아래와 같은 방법을 사용했습니다.

```

void DFSTree(Node * n)
{
    Node * brother;
    if (n==NULL) return;

    DFSTree(n->son);
    if (n->token == IF) {
        inner_if_cnt = count_if_nodes(n->son);
        if(n->brother != NULL) {
            if(n->brother->token == ELSE) {
                fprintf(fp, "GOTO ELSELABEL%d\n", (++ife_cnt));
            }
        }
    }
    if(n->token == WHILE) {
        inner_w_cnt = count_while_nodes(n->son);
    }
    if(n->token == FOR) {
        inner_f_cnt = count_f_nodes(n->son);
    }
    if(n->token == ELSE) {
        inner_e_cnt = count_e_nodes(n->son);
    }
    prtcode(n->token, n->tokenval);
    DFSTree(n->brother);
}

```

위는 DFS로 코드를 생성하는 함수인데 이 안에서 자식들을 순환한 뒤에 IF노드로 돌아올 경우 brother ELSE라면 GOTO문을 찍도록 만들었습니다.

아래의 코드는 위 트리를 규칙으로 만든 형태입니다.

```

if_stmt      :      IF condition LB stmt_list RB RUN STMTEND {$2->token=if_cond_check($2->token); $$=MakeOPTree(IF, $2, $4); }
;

else_stmt    :      ELSE LB stmt_list RB RUN STMTEND { $$=MakeOPTree(ELSE, $3, NULL); }

```


이렇게 만들어 준뒤에 아래처럼 stmt의 규칙에도 포함시켰습니다.

```
|      if_stmt else_stmt { $$=MakeOPTree(IF_ELSE, $1, $2); }  
|      if_stmt
```

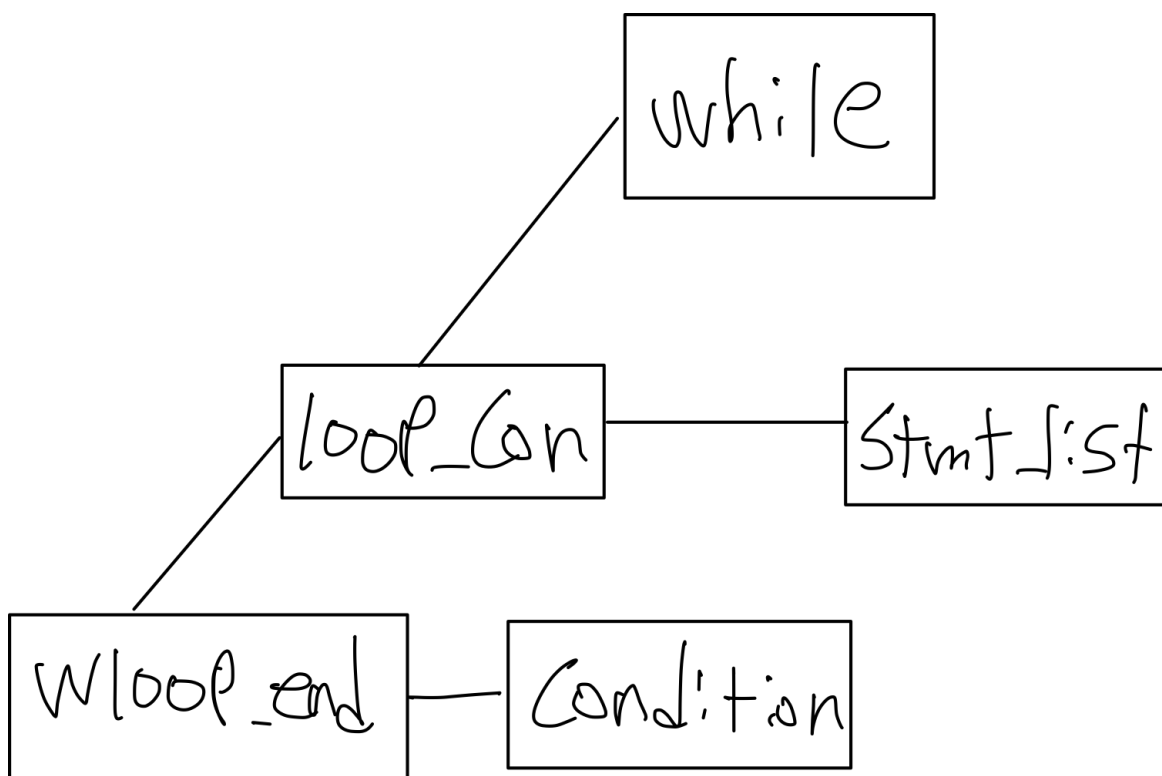
마지막으로 아래는 노드를 만났을 때 찍어주는 코드입니다.

```
case IF:  
    fprintf(fp, "LABEL IFLABEL%d\n", if_cnt - inner_if_cnt);  
    break;  
case IF_ELSE:  
    fprintf(fp, "LABEL ELSELABEL%d\n", ife_cnt-inner_e_cnt);  
    break;
```

```
case IF_EQ:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOTRUE IFLABEL%d\n", ++if_cnt);  
    break;  
case IF_NE:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOFALSE IFLABEL%d\n", ++if_cnt);  
    break;  
case IF_LT:  
    fprintf(fp, "-\n");  
    fprintf(fp, "COPY\n");  
    fprintf(fp, "GOPLUS IFLABEL%d\n", ++if_cnt);  
    fprintf(fp, "GOFALSE IFLABEL%d\n", if_cnt);  
    break;  
case IF_GT:  
    fprintf(fp, "-\n");  
    fprintf(fp, "COPY\n");  
    fprintf(fp, "GOMINUS IFLABEL%d\n", ++if_cnt);  
    fprintf(fp, "GOFALSE IFLABEL%d\n", if_cnt);  
    break;  
case IF_LE:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOPLUS IFLABEL%d\n", ++if_cnt);  
    break;  
case IF_GE:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOMINUS WLABEL%d\n", ++if_cnt);  
    break;
```

while

우선 while을 만들면서 어려웠던 점은 while문을 만나면 가장 먼저 LOOP LABEL을 찍어줘야하는데 찍어줄곳이 마땅치 않았습니다. 그래서 저는 wloop_end라는 새로운 노드를 만들어서 첫 LOOP LABEL을 찍어줄수 있도록 만들고 아래와 같이 트리를 만들었습니다.



위의 트리를 이와 같이 규칙으로 만들어 아래와 같이 stmt의 규칙에 포함시켰습니다.

```
loop_con LB stmt_list RB LOOP STMTEND { $$=MakeOPTree(WHILE, $1, $3); }
```

```
loop_con      :      condition wloop_end { $1->token=w_cond_check($1->token); $$=MakeOPTree(LOOPCON, $2, $1); };  
wloop_end     :      { $$=MakeNode(WLOOPEND, WLOOPEND); };
```

그리고 아래와 같이 노드에 따라 코드를 생성해주었습니다.

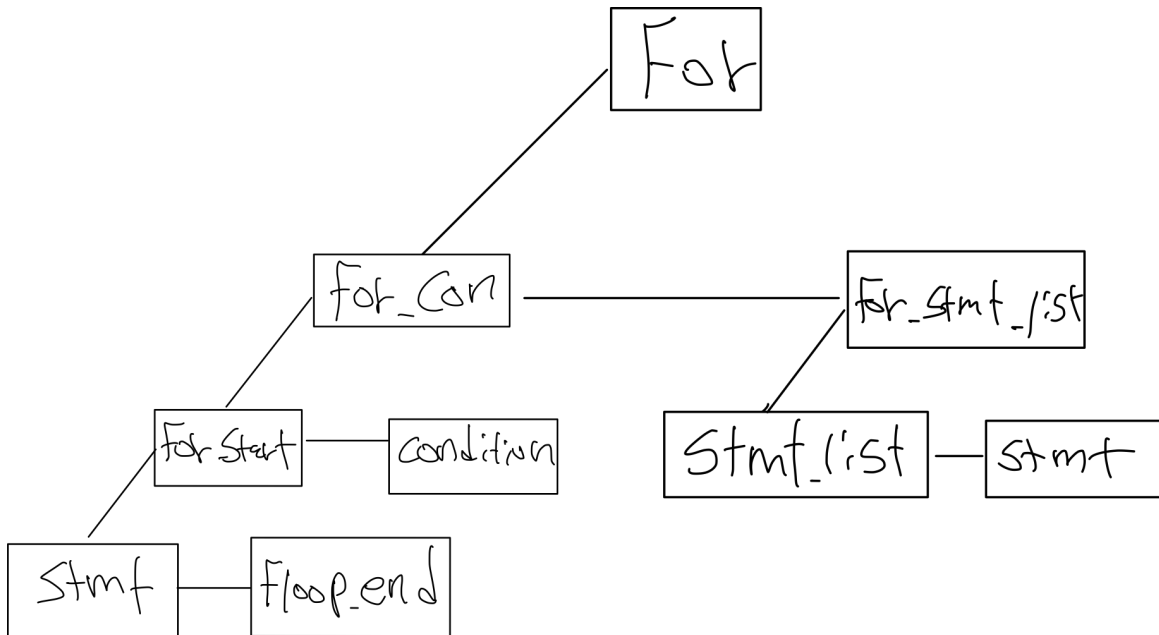
```
case WLOOPEND:
    fprintf(fp, "LABEL WLOOP%d\n", ++w_cnt);
    break;
```

```
case WHILE:
    fprintf(fp, "GOTO WLOOP%d\n", w_cnt-inner_w_cnt);
    fprintf(fp, "LABEL WLABEL%d\n", w_c_cnt-inner_w_cnt);
    break;|
```

```
case W_EQ:
    fprintf(fp, "-\n");
    fprintf(fp, "GOTRUE WLABEL%d\n", ++w_c_cnt);
    break;
case W_NE:
    fprintf(fp, "-\n");
    fprintf(fp, "GOFALSE WLABEL%d\n", ++w_c_cnt);
    break;
case W_LT:
    fprintf(fp, "-\n");
    fprintf(fp, "COPY\n");
    fprintf(fp, "GOPLUS WLABEL%d\n", ++w_c_cnt);
    fprintf(fp, "GOFALSE WLABEL%d\n", w_c_cnt);
    break;
case W_GT:
    fprintf(fp, "-\n");
    fprintf(fp, "COPY\n");
    fprintf(fp, "GOMINUS WLABEL%d\n", ++w_c_cnt);
    fprintf(fp, "GOFALSE WLABEL%d\n", w_c_cnt);
    break;
case W_LE:
    fprintf(fp, "-\n");
    fprintf(fp, "GOPLUS WLABEL%d\n", ++w_c_cnt);
    break;
case W_GE:
    fprintf(fp, "-\n");
    fprintf(fp, "GOMINUS WLABEL%d\n", ++w_c_cnt);
    break;
```

for

for문 문법을 만들면서 어려웠던점은 for문의 문법순서와 DFS에서 생성되는 코드의 순서를 서로 맞추는 부분이 어려웠습니다. 일단 for문에서도 while문과 동일하게 첫 LOOP LABEL을 찍어줄 노드가 필요하기 때문에 `floop_end`라는 노드를 만들었습니다. 설계한 트리는 아래와 같습니다.



DFS순서에 따라 설명드리면

`stmt(i=0) -> floop_end(LOOP LABEL) -> for_start -> condition`
`-> for_con -> stmt_list(for문 내부) -> stmt(i++) -> for_stmt_list`
`-> FOR`

저는 위에서 말했던 문제점을 MakeOPTree의 파라미터 인자로 순서를 바꿔서 넣음으로써 해결했습니다.

아래는 위 트리의 규칙입니다.

```
floop_end          :      { $$=MakeNode(FL00PEND, FL00PEND); };
```

```
for_stmt_list      :      LAST stmt ING LB stmt_list RB LOOP STMTEND      { $$=MakeOPTree(F0RSTLI, $5, $2); }  
;   
for_con            :      for_start condition      { $2->token=f_cond_check($2->token); $$=MakeOPTree(F0RCON, $1, $2); }  
;   
for_start          :      stmt floop_end { $$=MakeOPTree(F0RSTART, $1, $2); }  
;
```

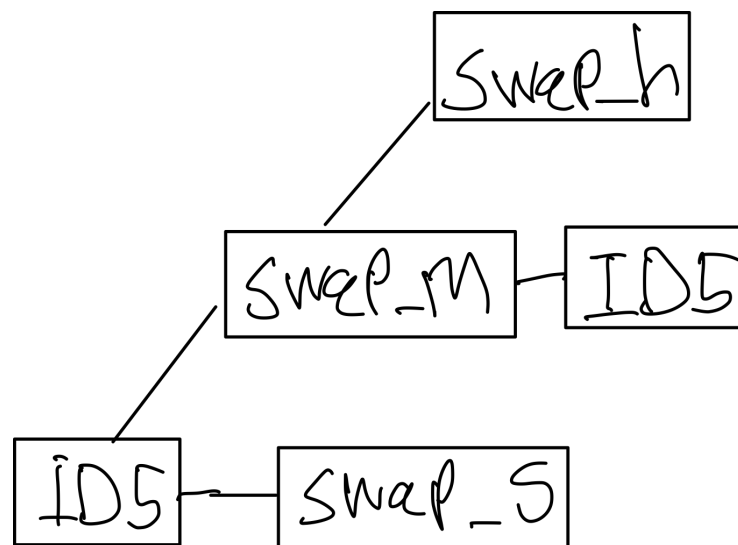
아래는 노드에 따라 생성하는 코드입니다.

```
case F_EQ:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOTRUE FLABEL%d\n", ++f_c_cnt);  
    break;  
case F_NE:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOFALSE FLABEL%d\n", ++f_c_cnt);  
    break;  
case F_LT:  
    fprintf(fp, "-\n");  
    fprintf(fp, "COPY\n");  
    fprintf(fp, "GOPLUS FLABEL%d\n", ++f_c_cnt);  
    fprintf(fp, "GOFALSE FLABEL%d\n", f_c_cnt);  
    break;  
case F_GT:  
    fprintf(fp, "-\n");  
    fprintf(fp, "COPY\n");  
    fprintf(fp, "GOMINUS FLABEL%d\n", ++f_c_cnt);  
    fprintf(fp, "GOFALSE FLABEL%d\n", f_c_cnt);  
    break;  
case F_LE:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOPLUS FLABEL%d\n", ++f_c_cnt);  
    break;  
case F_GE:  
    fprintf(fp, "-\n");  
    fprintf(fp, "GOMINUS FLABEL%d\n", ++f_c_cnt);  
    break;
```

```
case FL00PEND:  
    fprintf(fp, "LABEL FL00P%d\n", ++for_cnt);  
    break;  
  
break;  
case FOR:  
    fprintf(fp, "GOTO FL00P%d\n", for_cnt - inner_f_cnt);  
    fprintf(fp, "LABEL FLABEL%d\n", f_c_cnt - inner_f_cnt);  
    break;
```

swap

swap을 만들때 고민했던 점은 temp의 처리였습니다. 어떤 두 값을 서로 바꾸기 위해선 temp가 필요합니다. 따라서 insertsym함수로 swap문법을 실행할때 TEMP라는 변수를 넣어 DW로 메모리를 할당할수있게 하였습니다. 아래는 swap의 트리입니다.



여기서 ID5란 swap을 위한 새로운 토큰입니다.

먼저 각 노드가 어떤 코드를 생성하는지 설명하기전에 규칙을 보여드리겠습니다.

```
| swap_h SWAP {$$=MakeOPTree(SWAP, $1, NULL); }
```

```
swap_h      :      swap_m ID      {$2->token=ID5; $$=MakeOPTree(SWAPH, $1, $2);}
swap_m      :      ID WA swap_s    {$1->token=ID5; $$=MakeOPTree(SWAPM, $3, $1);}
swap_s      :      {$$=MakeNode(SWAPS,SWAPS); }
```

위와 같은 규칙을 만들었고 각 노드에 따라 아래와 같은 코드를 생성합니다.

```
case SWAPS:
```

```
    insertsym("TEMP");
```

```
    fprintf(fp, "LVALUE TEMP\n");
```

```
    break;
```

```
case SWAPE:
```

```
    fprintf(fp, "RVALUE TEMP\n");
```

```
    fprintf(fp, ":=\n");
```

```
    break;
```

```
case ID5:
```

```
    fprintf(fp, "RVALUE %s\n", symtbl[val]);
```

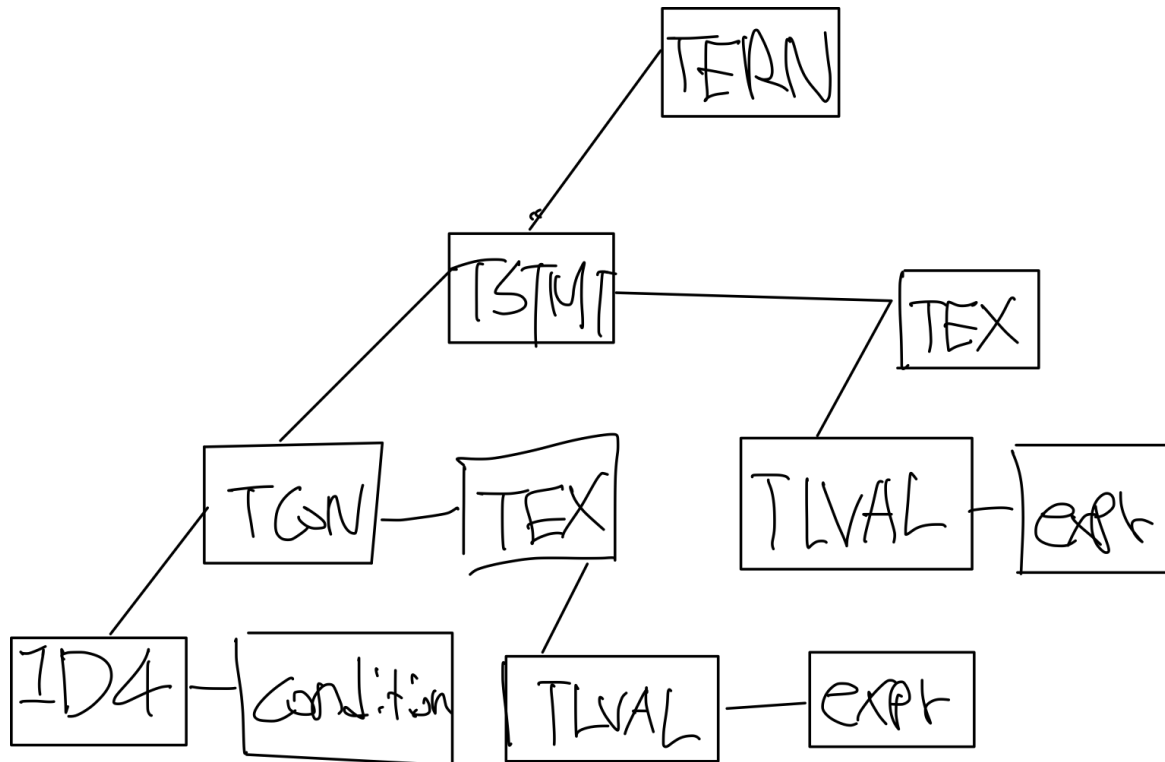
```
    fprintf(fp, ":=\n");
```

```
    fprintf(fp, "LVALUE %s\n", symtbl[val]);
```

```
    break;
```

ternery-operator

3항 연산자에서 고민했던 점은 이제 조건에 따라 어떤 LVALUE에 다른 값을 넣어줘야하는데 제가 생각하기에 이걸 변수명을 저장하는 방법으로 해결할수있을거같았습니다. 우선 아래는 3항 연산자의 트리입니다.



위 트리에서 보는 바와같이 ID4라는 노드를 만들어서 ID의 변수명을 저장할수있도록 했습니다. 그 이후 TLVAL을 사용해서 저장한 변수명을 이용해 ID4의 변수에 저장할수있는 코드를 생성했습니다. 저장하는데 사용한 로직은 char[50] str 이라는 변수를 만든뒤 ID4에서 strcpy로 변수명을 저장하도록했습니다. 아래는 트리를 규칙으로 생성한것입니다.

```
t_stmt      :      t_con t_expr ELSE { $$=MakeOPTree(TSTMT, $1, $2); }
;

t_expr      :      t_lval expr THIS SAVE STMTEND { $$=MakeOPTree(TEX, $1, $2); }
;

t_lval      :      { $$=MakeNode(TLVAL, TLVAL); }
;

t_con       :      ID ASSGN condition { $3->token=t_cond_check($3->token); $1->token=ID4; $$=MakeOPTree(TCON, $1, $3); }
```



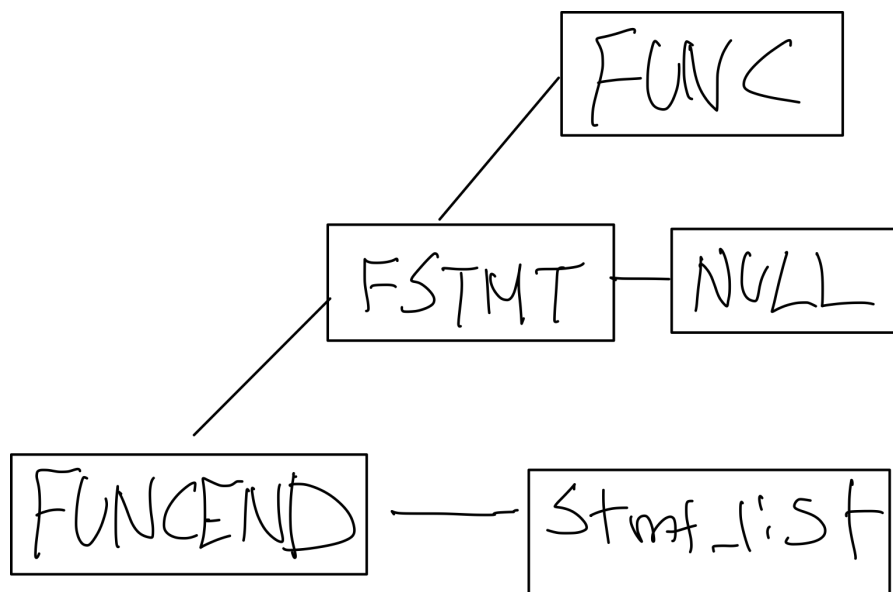
```
t_stmt t_expr { $$=MakeOPTree(TERN, $1, $2); }
```

그리고 노드에 따른 코드 생성입니다.

```
case T_EQ:
    fprintf(fp, "-\n");
    fprintf(fp, "GOTRUE TL%d\n", ++tl_cnt);
    break;
case T_NE:
    fprintf(fp, "-\n");
    fprintf(fp, "GOFALSE TL%d\n", ++tl_cnt);
    break;
case T_LT:
    fprintf(fp, "-\n");
    fprintf(fp, "COPY\n");
    fprintf(fp, "GOPLUS TL%d\n", ++tl_cnt);
    fprintf(fp, "GOFALSE TL%d\n", tl_cnt);
    break;
case T_GT:
    fprintf(fp, "-\n");
    fprintf(fp, "COPY\n");
    fprintf(fp, "GOMINUS TL%d\n", ++tl_cnt);
    fprintf(fp, "GOFALSE TL%d\n", tl_cnt);
    break;
case T_LE:
    fprintf(fp, "-\n");
    fprintf(fp, "GOPLUS TL%d\n", ++tl_cnt);
    break;
case T_GE:
    fprintf(fp, "-\n");
    fprintf(fp, "GOMINUS TL%d\n", ++tl_cnt);
    break;
case TEX:
    fprintf(fp, ":\n");
    break;
case TLVAL:
    fprintf(fp, "LVALUE %s\n", str);
    break;
case TSTMT:
    fprintf(fp, "GOTO TOUT%d\n", tl_cnt);
    fprintf(fp, "LABEL TL%d\n", tl_cnt);
    break;
case TERN:
    fprintf(fp, "LABEL TOUT%d\n", tl_cnt);
    break;
case ID4:
    strcpy(str, sytbl[val]);
    break;
```

function

우리가 흔히 C에서 쓰는 함수는 함수의 선언 및 호출로 이루어져있습니다. 이게 Stacksim에서 이루어지기 위해서는 함수 선언에서는 함수의 내용을 실행하면 안되고 함수를 호출할때 실행되어 다시 함수를 호출한 위치로 돌아와야합니다. 따라서 Entreaty 언어에서의 함수를 설계할때 LABEL을 사용하도록 했습니다. 우선 트리의 설계는 아래와 같습니다.



위의 트리에서 FUNC에서 함수선언에 대한 코드를 만나면 바로 함수 끝으로 점프할 GOTO문을 생성하는 FUNCEND노드와 함수의 내용인 stmt_list 순서로 코드가 생성되고 마지막으로 FUNC노드에서 함수의 끝을 의미하는 LABEL과 나중에 함수가 호출되었을때 함수의 내용을 모두 실행하고 다시 호출한 자리로 돌아갈 GOTO문을 생성합니다. 아래는 트리를 따라 만든 규칙입니다.

```
f_stmt_list      :      stmt_list f_end  { $$=MakeOPTree(FSTMT, $2, $1); }
:
f_end            :      { $$=MakeNode(FUNCEND, FUNCEND); }
FUNC FNAME AND CONTENT LB f_stmt_list RB IN FN THIS INIT STMTEND{ $$=MakeOPTree(FUNC, $6, NULL); }
```

다음으로 설명드릴건 함수를 호출하는 CALLBACK 노드입니다. CALLBACK노드의 규칙은 아래와 같이 간단합니다.

```
call_back      :      FNAME THIS CALL STMTEND {$$=MakeNode(CALLBACK, CALLBACK);}
; 
```

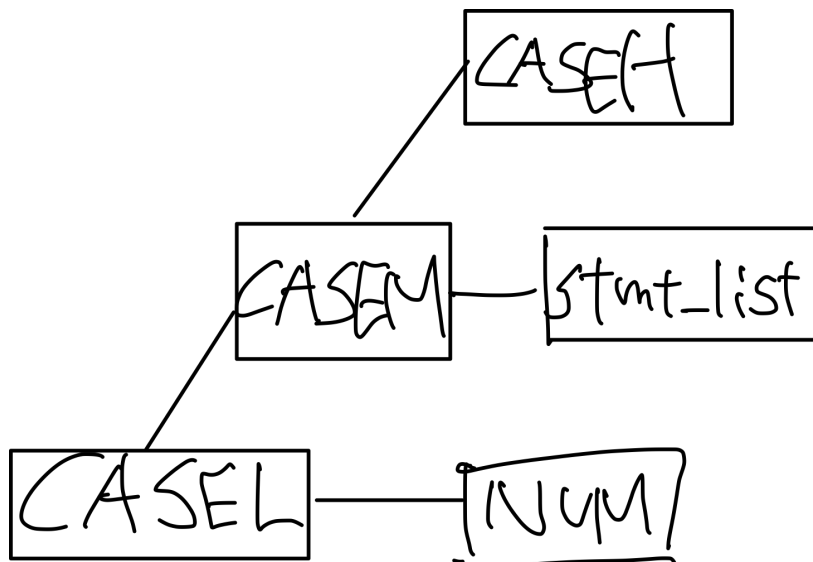
그리고 아래는 각 노드에 따라 생성하는 코드입니다.

```
case FUNCEND:
    fprintf(fp, "GOTO FLABEL%d\n", ++f_cnt);
    fprintf(fp, "LABEL CALLBACK%d\n", cb_cnt);
    cb_cnt--;
    break;
case FUNC:
    fprintf(fp, "GOTO FUNCOUT%d\n", ++fe_cnt);
    fprintf(fp, "LABEL FLABEL%d\n", f_cnt--);
    f_cnt--;
    break;
case CALLBACK:
    fprintf(fp, "GOTO CALLBACK%d\n", cb_cnt+1);
    fprintf(fp, "LABEL FUNCOUT%d\n", fe_cnt);
    fe_cnt--;
    break;
```

위와같이 만들면 함수가 선언된 코드를 만나면 함수 끝으로 점프하고 다시 콜백함수를 만났을때 함수 내용의 시작 부분으로 점프합니다. 그이후 함수의 내용을 전부 실행하고 난뒤에 다시 호출한 자리로 점프함으로써 C에서의 함수 선언 및 호출과 같이 작동하도록 만들었습니다.

switch

switch문은 내부에서 case문이 반복되어 나타나는 형태입니다. 이를 구현하기 위해 MakeListTree함수를 사용하여 case문이 반복되어 나타날수있는 case_list 규칙을 만들고 이를 switch문에서 사용했습니다. 트리는 아래와 같습니다.



트리를 설명하자면 CASEL에서 switch문에서 입력받은 변수를 RVALUE로 찍어줍니다. 그 후 숫자와 변수의 값이 같은 경우 stmt_list를 실행하고 아닐경우 다음 case로 넘어갑니다. 위의 트리를 통해 아래와 같은 규칙을 만들었습니다.

```
case_h      :      case_m COLON stmt_list BREAK      {$$=MakeOPTree(CASEH, $1, $3);}
;

case_m      :      NUM CASE case_l      {$$=MakeOPTree(CASEM, $3, $1);} //todo
;

case_l      :      {$$=MakeNode(CASEL, CASEL);}
;
```

그리고 이런 규칙을 통해 만든 case_h를 다음과 같이 list로 만들어 주었습니다.

```
case_list   :      case_list case_h      {$$=MakeListTree($1, $2); }
|      case_h      {$$=MakeListTree(NULL, $1); }
;
```

그리고 다음과 같이 SWITCHH노드의 자식으로 넣어주었습니다. 또한 default도 구현하기 위해서 따로 규칙을 만든뒤에 가장 상위노드 SWITCH노드의 자식으로 넣어주었습니다.

```
switch_stmt      :      SWITCH ID IS case_list {$2->token=ID4; $$=MakeOPTree(SWITCHH, $2, $4);}
```

```
default_stmt     :      DEFAULT COLON stmt_list BREAK  {$$=MakeOPTree(DEFAULT, $3, NULL); }
```

```
switch_stmt default_stmt      { $$=MakeOPTree(SWITCH, $1, $2); }
```

마지막으로 각 노드에 따른 코드 생성입니다.

```
case CASEL:
    fprintf(fp, "RVALUE %s\n", str);
    break;
case CASEM:
    fprintf(fp, "-\n");
    fprintf(fp, "GOTRUE CLABEL%d\n", ++case_cnt);
    break;
case CASEH:
    fprintf(fp, "GOTO SLABEL%d\n", s_cnt);
    fprintf(fp, "LABEL CLABEL%d\n", case_cnt);
    break;
case SWITCH:
    fprintf(fp, "LABEL SLABEL%d\n", s_cnt++);
    break;
```

printf 와 scanf

printf와 scanf는 설명하기도 뭐할만큼 정말 간단하게 구현하였습니다.

```
| ID VAL SCANF {$1->token=ID2; $$=MakeOPTree(SCAN, $1, NULL);}
| ID VAL PRINTF STMTEND { $$=MakeOPTree(PRINT, $1, NULL); }
```

위와 같은 규칙을 만들어 ID값을 INNUM을 통해 받거나 OUTNUM을 통해 출력할수 있도록 하였습니다. 아래는 노드에 따른 코드 생성입니다.

```
case SCAN:
    fprintf(fp, "INNUM\n");
    fprintf(fp, ":=\n");
    break;
case PRINT:
    fprintf(fp, "OUTNUM\n");
    break;
```

lex

```
%}
sp          [ \t]
ws          {sp}+
nl          \n
eletter     [A-Za-z]
letter      ({eletter})
digit       [0-9]
id          {letter}({letter}|{digit})*
%%
{ws}        { /* do nothing */ }
{nl}        { lineno++; }
\+|더하기   {return(ADD); }
\-|빼기     {return(SUB); }
=           {return(ASSGN); }
을|를       {return(THIS); }
"*"|곱하기   {return(MUL); }
"/"|나누기   {return(DIV); }
해주세요    {return(STMTEND); }
여기부터    {return(LB); }
여기까지    {return(RB); }
하면서     {return(ING); }
만약        {return(IF); }
가          {return(IS); }
보다        {return(THEN); }
와|과       {return(WA); }
저장        {return(SAVE); }
실행        {return(RUN); }
마지막에    {return(LAST); }
아니라면    {return(ELSE); }
이름이      {return(FUNC); }
이고        {return(AND); }
인          {return(IN); }
선언        {return(INIT); }
내용은      {return(CONTENT); }
함수        {return(FN); }
반복        {return(LOOP); }
작다면      {return(LT); }
크다면      {return(GT); }
"의 값을"   {return(VAL); }
"작거나 같다면" {return(LE); }
"크거나 같다면" {return(GE); }
같다면      {return(EQ); }
"가지 않다면" {return(NE); }
```

```

return(NUM); }
"+="      {return(AA); }
"-= "     {return(SA); }
"*="      {return(MA); }
"/="      {return(DA); }
일단      {return(FOR); }
"입력 받아주세요" {return(SCANF); }
"++"      {return(INC); }
"--"      {return(DEC); }
호출      {return(CALL); }
출력      {return(PRINTF); }
이제      {return(NOW); }
"의 값을 서로 바꿔주세요" {return(SWAP); }
"일 경우"  {return(CASE); }
"그리고 나가주세요" {return(BREAK); }
"다 아닌경우" {return(DEFAULT); }
"아래 경우 중" {return(SWITCH); }
"프로그램을 시작해주세요" {return(START); }
"프로그램을 끝내주세요" {return(END); }
func_{letter}+ {return(FNAME); }
{id} {temp=insertsym(yytext); yylval=MakeNode(ID, temp); return(ID); }
-?{digit}+ {sscanf(yytext, "%d", &temp); yylval=MakeNode(NUM, temp); return(NUM); }
-?{digit}+.{digit}+ {sscanf(yytext, "%f", &temp); yylval=MakeNode(REAL, temp); return(REAL); }
. {printf("invalid token %s\n", yytext); }
%%

```

lex를 가장 늦게 설명한 이유는 제가 언어를 만들때 lex를 가장 마지막으로 바꿨기 때문입니다. 제가 문법을 설계할때 중점적으로 생각한것은 그 언어가 제대로 동작되게하는 트리 구조였습니다. 그래서 먼저 C언어와 똑같이 다

다음에 코드를 자연스럽게 부탁하는 한국어말로 변경하는것은 어렵지 않았습니다. 코드를 자연스럽게 만들기위해 쓸데없는 토큰을 꽤 많이하였습니다.

4. 결과

if-else

```
프로그램을 시작해주세요
a에 10을 저장해주세요
만약 a가 10과 같다면 여기부터
    a++해주세요
    만약 a가 10보다 크다면 여기부터
        a에 15를 저장해주세요
        만약 a가 16과 같다면 여기부터
            a에 20을 저장해주세요
        여기까지 실행 해주세요 아니라면 여기부터
            만약 a가 15와 같다면 여기부터
                a에 50을 저장해주세요
            여기까지 실행해주세요
        여기까지 실행 해주세요

    만약 a가 20과 같지 않다면 여기부터
        a에 50을 저장해주세요
    여기까지 실행 해주세요
    여기까지 실행 해주세요
a가 40보다 작다면 여기부터
    a++해주세요
    여기까지 반복해주세요
여기까지 실행해주세요

만약 a가 50과 같다면 여기부터
    a에 149를 저장해주세요
여기까지 실행 해주세요
프로그램을 끝내주세요
```

```
Successfully assembled.
Start of execution.
Successfully executed.

[DATA Dump]
Loc#  Symbol      Value
    0   a          149
[End of Dump]
```

위 소스에서 처음에 a에 10을 저장하고 조건문이 참이기때문에 a++를 하면 16이 된다. 이후 a가 10보다 크기때문에 15를 다시 저장하고 else구문인 a에 50을 저장하게된다. 그리고 그이후로 조건문이 참인 if문이 없기때문에 밖으로 나와서 밖의 조건문 50과 같기때문에 최종 적으로 a는 149가 나오게된다.

while

```
프로그램을 시작해주세요
a에 0을 저장해주세요
b에 0을 저장해주세요
c에 0을 저장해주세요
a가 50보다 작다면 여기부터
    b가 50보다 작다면 여기부터
        b++
    여기까지 반복해주세요
    a++
여기까지 반복해주세요

c가 50보다 작다면 여기부터
    a++
    b++
    c++
여기까지 반복해주세요
프로그램을 끝내주세요
```

Successfully assembled.

Start of execution.

Successfully executed.

[DATA Dump]

Loc#	Symbol	Value
0	a	100
1	b	100
2	c	50

[End of Dump]

첫번째 while문 안에서 a와 b가 모두 50이되고 그후 밖으로 나와 다시 while문에서 c가 0에서 50이 될때까지 a와 b모두 1씩 더해지기때문에 결과는 둘다 100 c는 50이 나온다.

for

프로그램을 시작해주세요

B에 0을 저장해주세요

일단 i에 0을 저장해주세요 i가 10보다 작다면 마지막에 i++ 하면서 여기부터

일단 j에 0을 저장해주세요 j가 5보다 작다면 마지막에 j++ 하면서 여기부터

일단 y에 0을 저장해주세요 y가 10보다 작다면 마지막에 y++하면서 여기부터

B++

여기까지 반복해주세요

여기까지 반복해주세요

여기까지 반복해주세요

프로그램을 끝내주세요

Successfully assembled.

Start of execution.

Successfully executed.

[DATA Dump]

Loc#	Symbol	Value
0	B	500
1	i	10
2	j	5
3	y	10

가장 내부의 for문에서 B++이 10번반복되고 이 반복문이 5번 반복되고 또 이반복문이 10번 반복되기때문에 B의 값은 결과적으로 $10 \times 5 \times 10$ 으로 500이 나오면 정상이다.

tern

프로그램을 시작해주세요

a에 10을 저장해주세요

b에 20을 저장해주세요

max에 a가 b보다 크다면 a를 저장해주세요 아니라면 b를 저장해주세요

프로그램을 끝내주세요

Successfully assembled.

Start of execution.

Successfully executed.

[DATA Dump]

Loc#	Symbol	Value
0	a	10
1	b	20
2	max	20

[End of Dump]

a가 10이고 b는 20이기때문에 3항연산자에서 max에는 20이 들어가게된다.

print-scan

|프로그램을 시작해주세요
a의 값을 입력 받아주세요
a가 10보다 작다면 여기부터
 a++해주세요
 a의 값을 출력 해주세요
여기까지 반복해주세요
프로그램을 끝내주세요

```
Successfully assembled.  
Start of execution.  
1  
2345678910  
Successfully executed.  
[DATA Dump]  
Loc#  Symbol      Value  
    0   a          10  
[End of Dump]
```

a의 값을 1로 입력받게되면 a가 10이 될때까지 a++을하고 출력을하는것을 반복하기때
문에 2~10까지 차례대로 출력된다.

swap

프로그램을 시작해주세요
a에 10을 저장해주세요
b에 149를 저장해주세요

a와 b의 값을 서로 바꿔주세요
프로그램을 끝내주세요

Successfully assembled.

Start of execution.

Successfully executed.

[DATA Dump]		
Loc#	Symbol	Value
0	a	149
1	b	10
2	TEMP	10

[End of Dump]

a에 10을 저장하고 b에 149를 저장한뒤에 swap문법을 사용하였다. 두값이 서로 바뀌어 저장되어있는것을 결과를 통해 볼수있고, 두 값을 변경할때 쓴 TEMP도 생성된걸 볼수있다.

switch

```
프로그램을 시작해주세요
a에 2를 저장해주세요
아래 경우 중 a가
    1일 경우: a에 1을 저장해주세요 그리고 나가주세요
    2일 경우: a에 149를 저장해주세요 그리고 나가주세요
    3일 경우: a에 3을 저장해주세요 그리고 나가주세요
    다 아닌경우: a++ 그리고 나가주세요
프로그램을 끝내주세요
```

```
Successfully assembled.
Start of execution.
Successfully executed.

[DATA Dump]
Loc#  Symbol      Value
   0   a          149
[End of Dump]
```

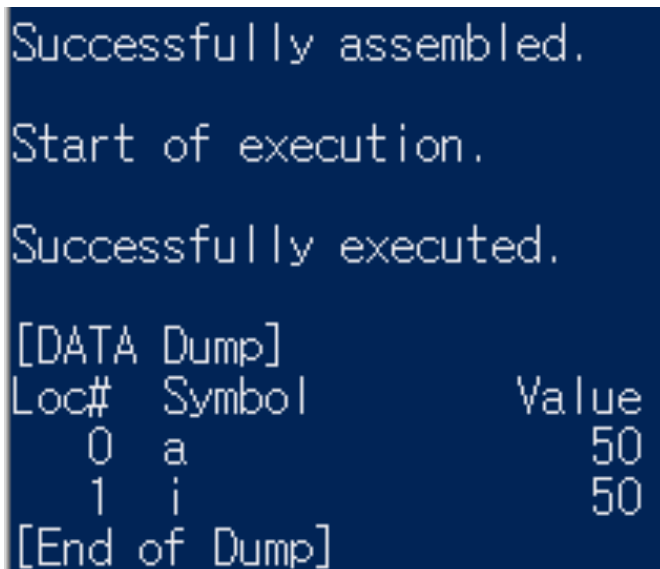
a의 값은 2이기 때문에 switch문에서 case2 구문이 실행되어 a에는 149가 저장되게 된다.

function

```
프로그램을 시작해주세요
이름이 func_a이고 내용은 여기부터
    a에 0을 저장해주세요
    일단 i에 0을 저장해주세요 i가 50보다 작다면 마지막에 i++하면서 여기부터
        a++
    여기까지 반복 해주세요
여기까지인 함수를 선언해주세요

a에 1004를 저장해주세요

func_a를 호출해주세요
프로그램을 끝내주세요
```



```
Successfully assembled.
Start of execution.
Successfully executed.

[DATA Dump]
Loc#  Symbol      Value
  0    a           50
  1    i           50
[End of Dump]
```

소스 시작에서 함수를 선언했는데 함수의 내용은 값이 0인 a를 i를 0부터 50까지 i++하면서 반복하고 반복마다 a의 값을 1증가시킵니다. 따라서 함수를 실행하면 a의 값은 50이 나와야합니다. 그리고 함수 선언부 밑에 a에 1004를 저장시켰습다. 원래의 순서라면 a에는 1004가 나와야겠지만 그 밑에서 func_a함수를 호출했기때문에 a의 값이 정상적으로 50이 나온걸 확인할수 있습니다.

integration-test

마지막으로 모든 문법을 같이 사용해도 문제가 없음을 확인하는 통합 테스트입니다. 일단 함수를 선언합니다. 내용은 a의 값은 3이므로 case문에서 3일 경우의 case구문이 실행되고 while문을 통해 a는 50까지 1씩 증가를 반복합니다. 이후 if문에서 조건문이 참이기때문에 b의 값으로 1004를 입력받고 a와 b의 값을 swap합니다. 그 이후 3항 연산자를 통해 max의 값을 출력하는데 까지입니다. max에 999999를 저장한뒤에 위의 내용의 함수를 호출하여 결과값이 정상적으로 나오는것을 확인할수있습니다.

프로그램을 시작해주세요

이름이 func_a이고 내용은 여기부터

a에 3을 저장해주세요

아래 경우 중 a가

1일 경우: a에 1을 저장해주세요 그리고 나가주세요

2일 경우: a에 149를 저장해주세요 그리고 나가주세요

3일 경우: a가 50보다 작다면 여기부터

a++

여기까지 반복해주세요

그리고 나가주세요

다 아닌경우: a++ 그리고 나가주세요

만약 a가 50과 같다면 여기부터

b의 값을 입력 받아주세요

a와 b의 값을 서로 바꿔주세요

여기까지 실행해주세요 아니라면 여기부터

a에 149를 저장해주세요

여기까지 실행해주세요

Successfully assembled.

Start of execution.

1004

1004

Successfully executed.

[DATA Dump]

Loc#	Symbol	Value
0	a	1004
1	b	50
2	max	1004
3	TEMP	50

[End of Dump]

max에 a가 b보다 크다면 a를 저장해주세요 아니라면 b를 저장해주세요

max의 값을 출력해주세요

여기까지인 함수를 선언해주세요

max에 999999를 저장해주세요

func_a를 호출해주세요

프로그램을 끝내주세요

5. 토론 및 결론

이번 기말 프로젝트는 시작하기전엔 할 생각만 하면 정말 한숨이 나올 정도로 어려워보였다. 처음 교수님의 Tree를 이용한 코드를 이해하기까지는 쉽지 않았던것은 사실이다. 전에 수업에서 배운내용들을 되돌아보며 트리를 통해 어떤식으로 어떤 순서로 코드를 생성하는지 이해하고나니 어느 시점에 어떤 노드가 들어가야되는지 점점 감이 잡히기 시작했다. 하지만 문제는 LABEL을 사용하는 문법들에서 생겼다. 처음 조건에서 GOTO LABEL?을 생성하고 그 내부에서 LABEL을 생성하는 문법들이 여러개 중첩되어 나타난다면 나중에 저 GOTO LABEL에 대응하는 LABEL을 생성해주어야했지만 방법을 생각해내기는 쉽지않았다. 긴 고민끝에 도달한 해결책은 DFS코드 생성에서 자식들의 코드 생성을 끝내고 자기 자신에게 돌아왔을때 DFS 탐색을 통해 자신의 내부에 있는 특정 노드들의 갯수를 셀 수 있었다. 이 갯수를 나중에 대응되는 LABEL을 찍어줄 때 빼주면서 LABEL의 이름을 중첩되어도 중복되지않게 만들었다. 이걸 통해 LABEL을 사용하는 모든 문법에 저 방법을 사용했고 결과는 성공적이었다. 내가 아직 찾지 못한 오류가 있을수도있을수도 있겠지만 지금으로서는 잘 돌아가는 것처럼 보인다. 위와 같이 언어를 만들면서 실제로 잘 동작하게 만드는데 성공할 때마다 정말 재미있었다. 이번 프로젝트를 통해 컴파일러에 대한 이해가 굉장히 높아질수 있는 계기가 되었던거 같다. 또 한편으로는 Python과 같은 정말 대중적이고 편리한 언어를 만든 사람들이 정말 대단한 사람들이라는 생각도 해볼수 있던거같다.