Compiler

Group 6

2015312810 천진우 2015313076 김범주 2015312198 이한범

표 1 < Grammar >

prog	::=	word "(" ")" block
block	::=	"{" slist "}"
	|	";"
slist	::=	slist stat
	|	stat
stat	::=	block
	|	IF cond THEN block ELSE block
	|	WHILE cond block
	|	word "=" expr ";"
	|	";"
cond	::=	expr ">" expr
	|	expr "<" expr
expr	::=	term
	|	term "+" fact
fact	::=	num
	|	word
word	::=	([a-z] | [A-Z])*
num	::=	[0-9]*

주어진 <grammar>를 ;의 모호함 때문에 다음과 같이 바꾸어 진행하였다.

A. Lexer

- Lexer는 주어진 input을 토큰 단위로 끊어서 parser에 전달하는 역할을 한다. 문제에서 주어진 문법을 통해 알 수 있는 terminal은 (,), {, }, +, =, <, >, ;(연산자 및 특수 문자), IF, THEN, ELSE, WHILE, WORD, NUM 이다(WORD, NUM은 숫자와 영문자로 변환 될 수 있지만 연속된 숫자 또는 문자로만 이루어진 문자를 통째로 WORD, NUM으로 치환했다).

- WORD, NUM과 keyword(IF, THEN, ELSE, WHILE을 제외한 다른 모든 terminal들은 영문자와 숫자를 사용하지 않기 때문에 영문자와 숫자는 연속해서 스캔 하고, 스캔 하는 중에 특수문자가 나오게 된다면 특수문자가 나오기 전 까지의 substring을 잘라 keyword, WORD, NUM을 판별한다. 특수문자는 한 글자로 이루어져 있으므로 특수문자가 나올 때마다 개별적으로 저장한다. 공백('₩n', '₩t', ' ')이 있다면 토큰에 아무것도 추가하지 않고 input을 계속 스캔한다. 문제에서 주어진 input([a-z]|[A-Z]|[0-9]과 특수문자, 공백)외에 다른 input이 들어온다면 error를 발생한다.

- Keyword는 대문자로만 이루어진 IF, THEN, ELSE, WHILE만 허용한다. if, then, else, while은 허용하지 않는다.

- 토큰은 vector에 순서대로 저장되며, vector는 pair<tokenType, string>으로 저장된다. tokenType은 토큰의 종류를 나타내는 enum 이다. Enum은 총 16개의 토큰 종류와 2개의 예외 종류가 정의되어 있다.

```
enum tokenType {
        OPENSPAR = 0, CLOSESPAR = 1, OPENMPAR = 2, CLOSEMPAR = 3,
        SEMICOLON = 4,
        KEY_IF = 5, KEY_THEN = 6, KEY_ELSE = 7, KEY_WHILE = 8,
        ASSIGN = 9, LARGER = 10, SMALLER = 11, PLUS = 12,
        WORD = 13, NUM = 14, END = 15,
        ERROR, WSPACE
};
```

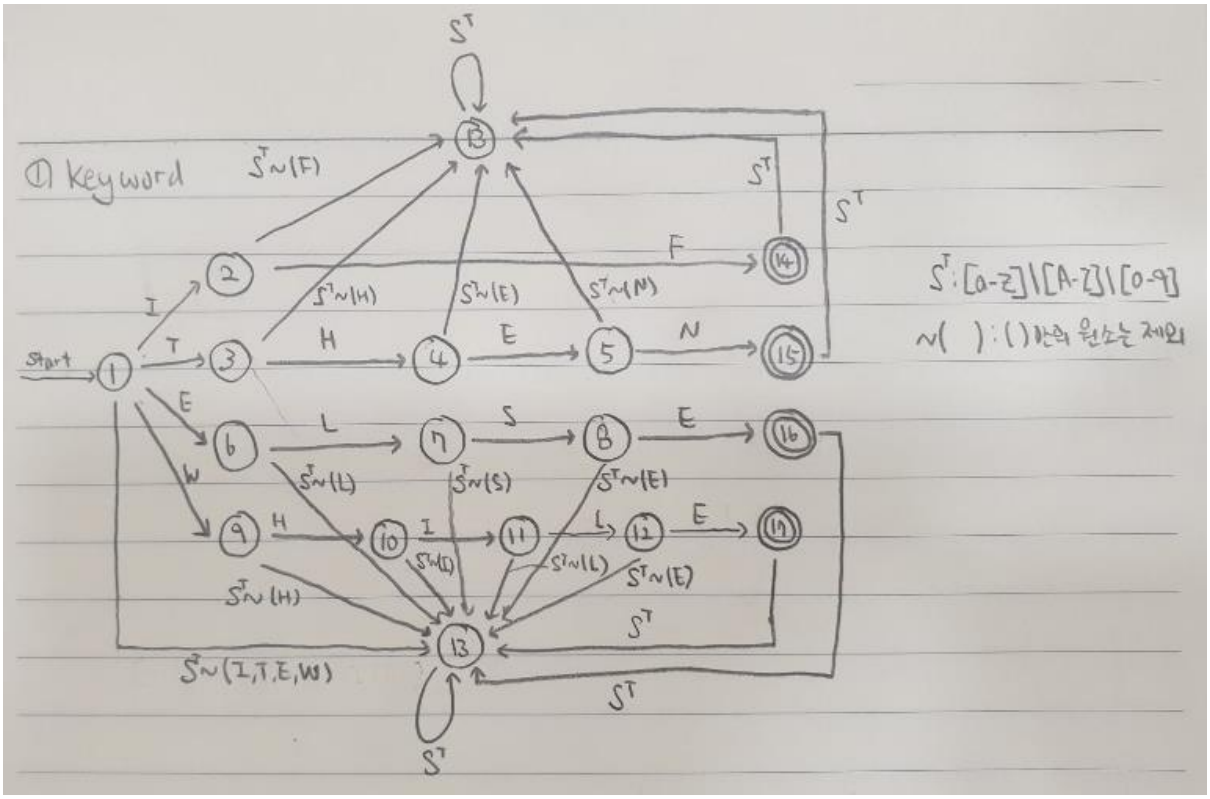
- 특수문자를 스캔 할 경우 즉시 vector에 tokenType과 특수문자에 맞는 string이 저장된다. 예를들어, (가 스캔 되었다면 vector에 < OPENSPAR, "(" > 이 저장된다.

- 영문자와 숫자가 스캔 된다면, 특수문자와 공백이 나올 때 까지 계속 스캔하여 substring에 저장한다. 저장한 substring은 keyword와 WORD, NUM을 판별하기 위해 keyword를 판별하는 DFA, WORD를 판별하는 DFA, NUM을 판별하는 DFA를 순서대로 작동한다(keyword 판별이 우선이기 때문). 각 DFA를 작동하면서 accept 된다면 뒤의 DFA는 작동하지 않고 accept된 토큰을 vector에 저장한다.

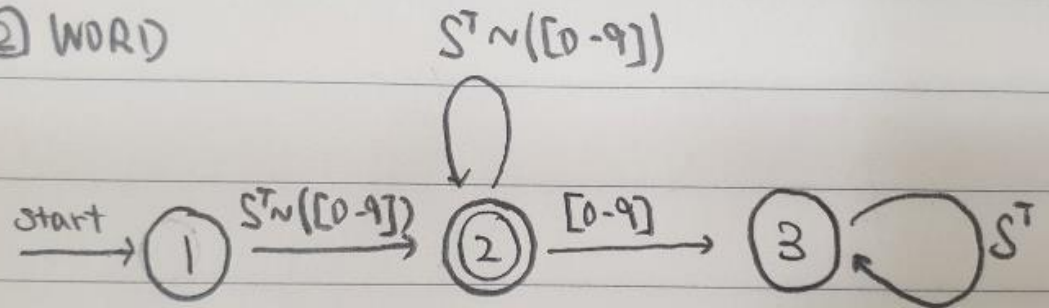
- 문제에 허용된 영문자, 숫자, 특수문자 이외의 input이나 영문자와 숫자로만 이루어진 string 중 WORD와 NUM 어느 곳에 속하지 않는 string이 input 중에 발견될 경우 토큰화를 중지하고 ERROR를 반환하여 잘못된 input이 들어왔음을 알리고, ERROR를 출력함과

동시에 프로그램을 종료한다.

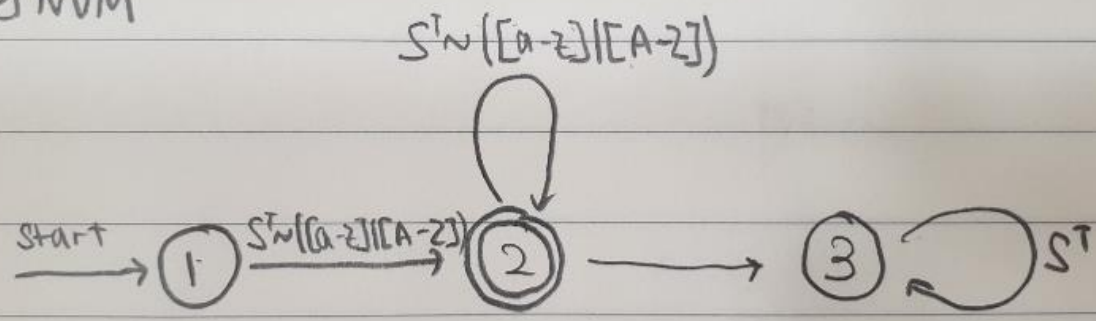
- 토큰 중에 WORD가 있다면, 이를 symbol table에 저장한다. Symbol table은 100개의 entry가 있으며 100개 이상의 WORD가 저장된다면 각 entry에 추가로 entry가 연결되어 확장한다. WORD 토큰을 symbol table에 저장할 때 중복된 WORD가 이미 저장 되어 있는지 확인한 후 symbol table에 저장한다.
- 나중에 code generator를 실행하기 위해 필요한 symbol들과 숫자들을 저장하기 위한 value table을 생성하였다. 토큰 중에 WORD, NUM이 있다면 value table에 저장한다. Value table은 symbol table과 동일한 구조를 가지며, 중복되는 WORD와 NUM의 저장을 허용하지 않는다.
- Keyword, WORD, NUM을 판별하는 DFA는 다음과 같다.
- 사용된 용어 중 S^T 는 [a-z]|[A-Z]|[0-9], 알파벳과 숫자 전부의 집합을 의미한다. A~( )는 A에 있는 원소 중 괄호 안에 표기한 원소를 제외한 나머지를 의미한다. State 13이 중복 표시 된 이유는 다른 state에서 13번 state로 가는 경우가 많아 그림을 알아보기 힘들기 때문에 추가로 동일한 state 13을 그림에 표시하였다.
-



② WORD

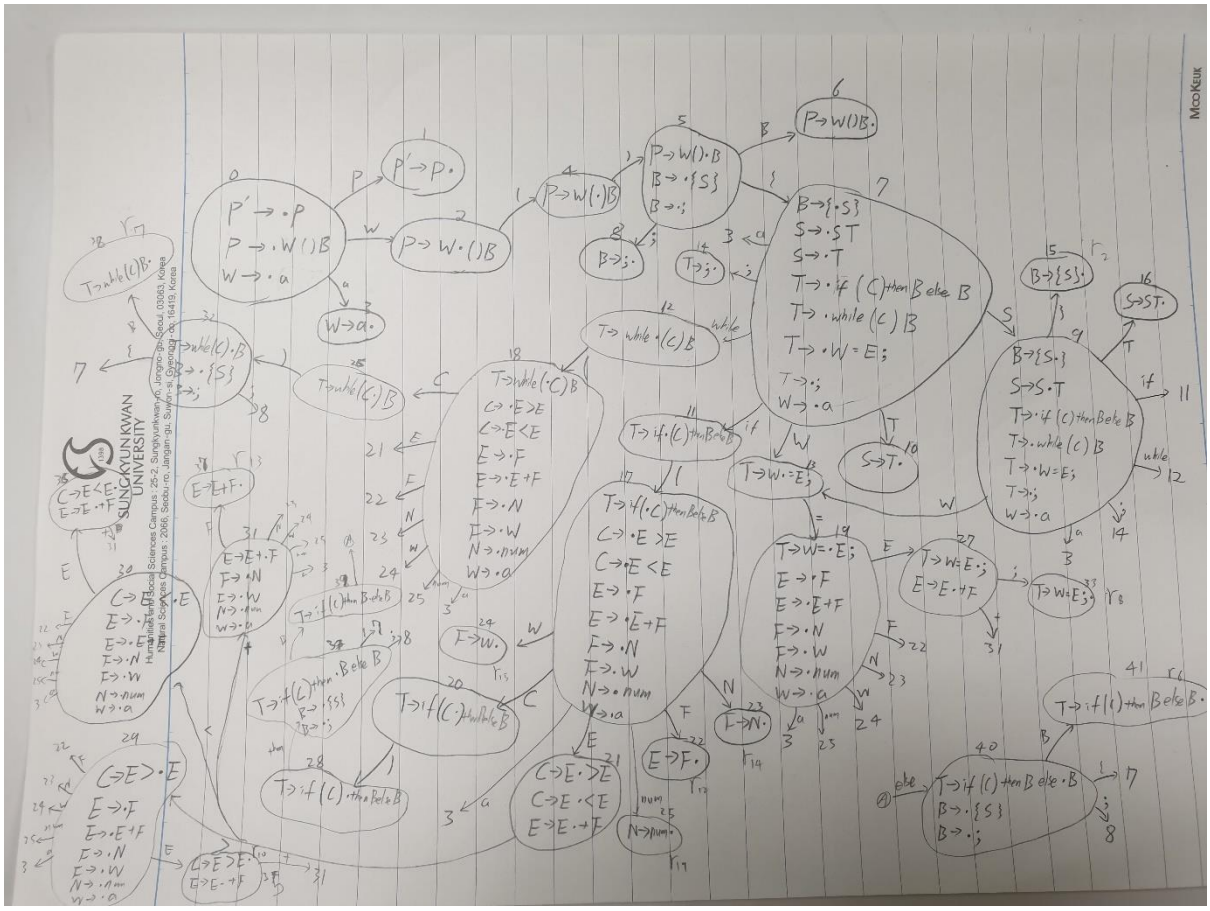


③ NUM

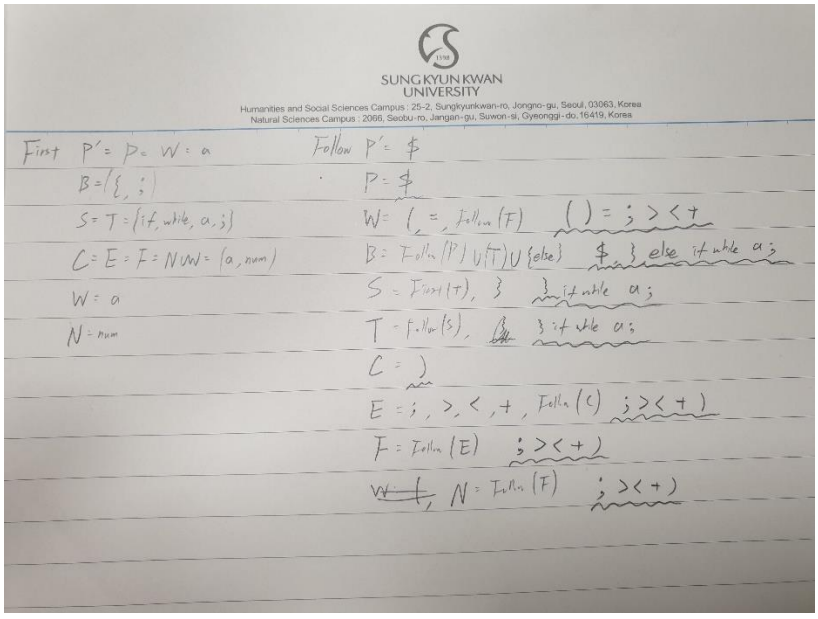


B. Parser

- 주어진 문법으로 SLR을 만들었다.

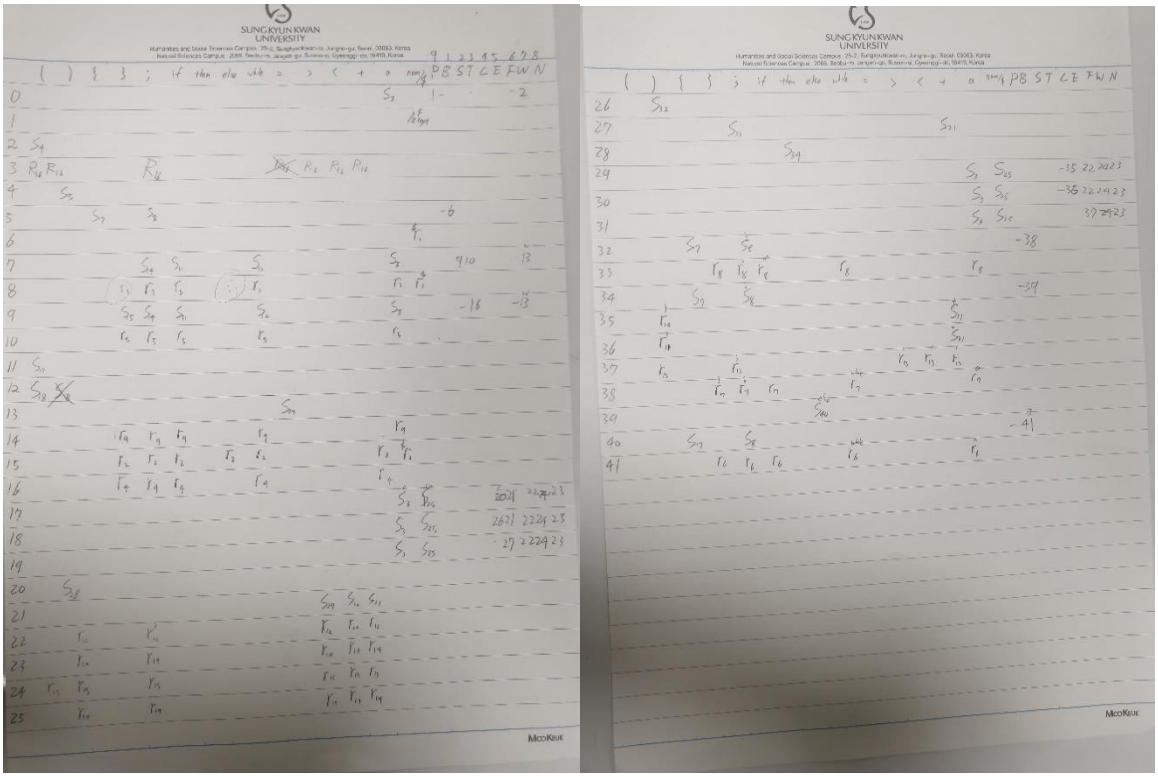


- 주어진 문법에서 각 NON-Terminal의 FIRST와 FOLLOW를 각각 구하여,



- 이를 활용하여 ACTION-GOTO table을 만들었다

-
-



- 이를 코드로 옮기는 작업을 수행하였다.

- Parser는 lexer에서 token단위로 구분된 input을 syntax analysis과정을 거쳐 intermediate expression인 Abstract Syntax Tree로 만드는 역할을 한다. lexer에서 사용한 enum type인 tokenType을 사용하여 ACTION table인 parsingTable[][]을 만들었다. 아래의 그림에서 parsingTable의 row index는 SLR state diagram의 state를 의미하고, tt가 들어가는 column index는 각 terminal symbol을 의미하여 ACTION table을 제작할 수 있었다. 이때 shift의 경우 shift되는 state를 그대로 저장하고, reduce의 경우 어떤 reduce인지 reduce 번호에 100을 더하여 저장하여 둘을 구분하였다.

```
string parser(){
    bool accept = 0;
    int parsingTable[45][20] = {0,};
    //shift 0, reduce 100

    enum tokenType tt;
    tt = WORD;
    parsingTable[0][tt] = 3;

    parsingTable[1][tt = END] = 100;

    parsingTable[2][tt = OPENSPAR] = 4;

    parsingTable[3][tt=OPENSPAR] = 116;
```

-

- PDA를 구현하기 위해 두 개의 Stack을 사용하였다. 각각 symbol, state를 저장하고 초기값으로 '$'와 0을 저장하여 준다.

```
stack<Node*> symbol;
stack<int> state;

Node* endNode = new Node();
endNode->id = make_pair(END,"$");
symbol.push(endNode);

state.push(0);
```

- 이후 lexer에서 만든 token을 읽어 나가며 Shift-Reduce를 진행한다.

- Shift의 경우 terminal을 Stack에 넣을 때마다 이를 새로운 노드에 저장하여 Stack에 넣는다. 이후에 만들 Tree를 쉽게 생성하기 위하여 Node*를 Stack에 넣는 방식을 활용하였다. Shift를 진행하면 다음 input을 읽을 수 있도록 inputIndex를 증가시켜 주었다. input을 끝까지 읽고 accept 되는 과정에서 valid한 input인지 확인하기 위해 shift 0 일 때 1)다음 input이 $가 아니거나 2) stack size가 1이 아니면 이를 invalid한 input으로 처리한다.

```
//shift
if(parsingTable[ state.top()][ token[inputIndex].first ] / 100 == 0){

    if(parsingTable[ state.top()][ token[inputIndex].first] == 0){
        if(token[inputIndex].second != "$") return "NOT_ACCEPT";
        else if(state.size() != 1) return "NOT_ACCEPT";
        else cout << "ACCEPT"<<endl;
    }
    state.push(parsingTable[state.top()][token[inputIndex].first] );

    Node * newNode = new Node();
    newNode->id = token[inputIndex];
    newNode->state = 0;
    symbol.push(newNode);
    inputIndex++;
}
```

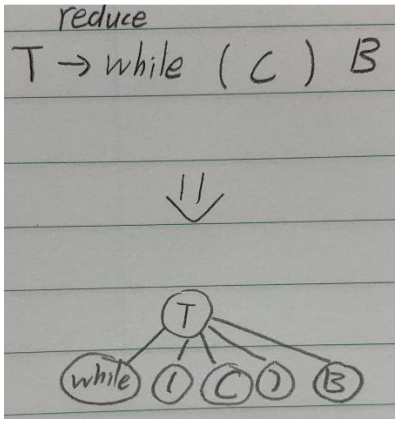
- Reduce의 경우 각 reduce가 어떤 reduce 인지에 따라 switch case문을 사용하여 진행한다.

```
//reduce
else if(parsingTable[ state.top()][ token[inputIndex].first ] / 100 == 1){

    Node * newNode = new Node();
    switch(parsingTable[state.top()][ token[inputIndex].first ] % 100){
        case(1):
```

- 예를 들어 T -> while (C) B 를 reduce 하는 case(7)의 경우, 위에서 생성된 newNode를 사용하여 stack에서 pop시키는 node들을 children으로 연결하여 저장하고, T state를 의미하는 3을 new Node의 state로 저장하여 준다. Reduce이후 GOTO table을 사용하여 state를 저장하여 준다.

```
case(7):
    for(int i = 0; i < 5; i++){
        newNode->children.push_back(symbol.top());
        symbol.top()->parent = newNode;
        symbol.pop();
        state.pop();
    }
    newNode->state = 3;
    symbol.push(newNode);
    if(state.top() == 7) state.push(10);
    else if(state.top() == 9) state.push(16);

    break;
```

- 다음의 그림은 위의 과정을 설명한다.



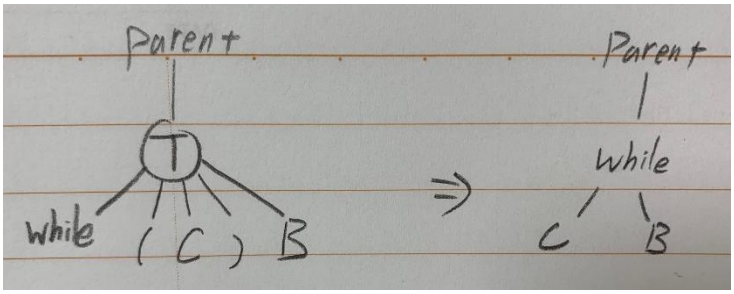
- 이후 Accept를 판별하기 위해 ACTION Table에 만들어 두었던 reduce 0를 활용하여 accept가 정상적으로 되는지 판별한다. Accept가 되었다면 symbol에 남아있는 Node를 root에 저장하여 준다.

```
case(0):
    //accept
    if(token[inputIndex].second == "$"){
        accept = 1;
        root = symbol.top();
        symbol.pop();
        state.pop();
    }else {
        return "NOT_ACCEPT";
    }
    break;
```

- 이 과정을 거쳐 AST가 생성되었다. 이제 이를 코드로 쉽게 변환하기 위하여 non-terminal 들을 없애 주었다. reform()함수를 만들어 진행하였다. DFS기법을 활용하여 현재 노드의 state와 child node의 개수에 따라 reform을 진행하였다. Terminal을 뜻하는 state 0의 경우 아무 것도 하지 않는다. 위에서 예시로 들었던 T -> while (C) B 의 경우를 다시 예시로 들면, T의 state는 3, child node의 개수는 5이다. 현재 노드의 위치에 while을 넣어주고, while의 child로 C와 B를 넣어준다.

```
case 3://T
    if(node->children.size() == 5){ // T -> while ( C ) B
        Node * par = node->parent;
        for(int i = 0; i < par->children.size(); i++){
            if(par->children[i] == node){
                par->children[i] = node->children[4];
                node->children[4]->parent = par;
                break;
            }
        }
        node->children[4]->children.push_back(node->children[0]);
        node->children[4]->children.push_back(node->children[2]);
    }
```

- 그림으로 표현하면 다음과 같다.



- 위와 같은 과정을 거쳐 parsing을 하여 syntax error를 검출하고, AST를 만든다. 만든 AST를 reform함수를 거쳐 code-generate하기 쉽게 바꿔주었다.

C. Code generator

- code generator 는 generation(node *node)함수를 통해 node 의 값에 무엇이 저장되어 있는지 확인하고 그에 따라 <Extended instruction set>에 명시되어 있는 set 들을 통해 assembly code 를 만들어 내는 작업을 수행한다.

- generation()함수는 최초로 저장되어 있는 트리의 root node 를 받고, 그 노드의 자식들을 따라가면서 함수를 수행한다. 그 노드의 값(node->id.second)이 if, while, +, <, >, =, 그리고 상수와 변수 중 어느 것인지에 따라서 자식들을 불러오는 순서가 바뀌며, 출력해주는 assembly code 도 다르다.

- node 의 값이 "if"인 경우 우선 전역변수로 저장되어 있는 labeln 을 통해 이동하게 되는 라벨들을 지정해 준다. labeln 은 다음에 라벨을 지정해 줄 때 같은 라벨이 되지 않도록 labeln = labeln+2 를 해 준다. 그러고 나면 node->child[0]에 연결되어 있는 cond 를 먼저 실행시켜 준 후, 그 결과값이 저장되는 레지스터를 reg1 에 저장한다. 이렇게 얻는 reg1 은 JUMPT 와 JUMPF 를 수행하는데 이용된다. reg1 이 참인 경우 첫 번째 block 으로 이동하고 거짓인 경우에는 두 번째 block 으로 이동한다.

- node 의 값이 "while"인 경우, 마찬가지로 label 을 지정해 주는데, 첫 번째로는 while 문을 시작하는 위치에 지정하여 조건을 만족하면 시작위치로 이동할 수 있도록 하고, 두 번째 라벨은 끝나는 위치에 지정하여 조건을 만족하지 못하면 block 을 실행하지 않고 끝나는 위치로 이동하도록 한다. 마찬가지로 labeln 의 값을 증가시켜 준다.

- node 의 값이 "="이거나 "+", "<", ">"인 경우는 비슷한 과정을 거친다. 우선 아무것도 들어있지 않은 레지스터에 결과값을 지정해 주기 위해 새로운 레지스터를 찾고, 연산을 위한 두 값을 node->child[0], node->child[1]에서 찾아와서 각각을 reg1, reg2 에 저장해 준다. (단, "="의 경우에는 새로운 레지스터가 필요 없다.)

  이렇게 찾은 레지스터들을 각각의 연산에 맞도록 ADD, MV, LT 구문으로 출력한다. 이 때, "<"와 ">"는 두 레지스터의 위치를 바꾸어 주면 같은 연산이 된다.

- node 값이 상수이거나 변수인 경우가 있다. 상수라면 LD Reg#, num 을 통해 레지스터에 값을 저장해 주고, 변수라면 미리 지정해둔 레지스터의 번호를 찾아서 return 해 주어 이전의 generation()에서 사용할 수 있도록 해준다. 추가로 node 값이 ";"인 경우는 아무것도 해주지 않는 블락이므로 아무런 출력도 하지 않고 그 부분을 마친다.

- parsing tree 에는 위의 것들 말고도 중간과정인 prog 와 slilst 이 있다. 이 경우 이들의 자식들이 어떻게 증가하는지를 보여주기 위한 것이고 실제로 이들이 하는 것들은 없으므로 이들의 자식들을 다시 generation()을 통해 실행되도록 하여준다. prog 는 자식들 중 함수이름과 ()sms 자식들이 없으므로 3 번째 자식노드만 generation()시켜주고 함수 이름은 출력시켜 준다.

D. 실행결과

test1.txt

```
1 func(){
2     a = 3;
3     b = 7;
4     testVal = 576;
5
6     IF (testVal < 600) THEN{
7         testVal = testVal + a;
8     }
9     ELSE{
10        a=a+b;
11    }
12
13    c=100;
14
15    WHILE(c>40){
16        c=c+a+b+testVal+3+a+16;
17    }
18
19    WHILE(c>40);
20
21    d=30;
22 }
```

```
BEGIN func
        LD Reg#2, 3
        MV Reg#1, Reg#2
        LD Reg#4, 7
        MV Reg#3, Reg#4
        LD Reg#6, 576
        MV Reg#5, Reg#6
        LD Reg#7, 600
        LT Reg#14, Reg#5, Reg#7
        JUMPT Reg#14 label 0
        JUMPF Reg#14 label 1
label 0:
        ADD Reg#15, Reg#5, Reg#1
        MV Reg#5, Reg#15
label 1:
        ADD Reg#16, Reg#1, Reg#3
        MV Reg#1, Reg#16
        LD Reg#9, 100
        MV Reg#8, Reg#9
label 2:
        LD Reg#10, 40
        LT Reg#17, Reg#10, Reg#8
        JUMPF Reg#17 label 3
        ADD Reg#18, Reg#8, Reg#1
        ADD Reg#19, Reg#18, Reg#3
        ADD Reg#20, Reg#19, Reg#5
        LD Reg#2, 3
        ADD Reg#21, Reg#20, Reg#2
        ADD Reg#22, Reg#21, Reg#1
        LD Reg#11, 16
        ADD Reg#23, Reg#22, Reg#11
        MV Reg#8, Reg#23
        JUMPT Reg#17 label 2
label 3:
label 4:
        LD Reg#10, 40
        LT Reg#24, Reg#10, Reg#8
        JUMPF Reg#24 label 5
        JUMPT Reg#24 label 4
label 5:
        LD Reg#13, 30
        MV Reg#12, Reg#13
END func
Used Register: 24
```

왼쪽의 파일을 컴파일하여 오른쪽과 같은 결과를 얻을 수 있었다.

test2.txt (왼쪽의 코드를 컴파일하여 오른쪽의 pseudo code 를 얻음)

```
1 main() {
2 a = 1;
3 b=2;
4 WHILE(c>a+b){
5     IF(c > 3) THEN{
6         c=c+2;
7     }
8     ELSE{
9         c=c+1;
10 }
11 }
12 c=3;
13 }
```

```
BEGIN main
        LD Reg#2, 1
        MV Reg#1, Reg#2
        LD Reg#4, 2
        MV Reg#3, Reg#4
label 0:
        ADD Reg#7, Reg#1, Reg#3
        LT Reg#8, Reg#7, Reg#5
        JUMPF Reg#8 label 1
        LD Reg#6, 3
        LT Reg#9, Reg#6, Reg#5
        JUMPT Reg#9 label 2
        JUMPF Reg#9 label 3
label 2:
        LD Reg#4, 2
        ADD Reg#10, Reg#5, Reg#4
        MV Reg#5, Reg#10
label 3:
        LD Reg#2, 1
        ADD Reg#11, Reg#5, Reg#2
        MV Reg#5, Reg#11
        JUMPT Reg#8 label 0
label 1:
        LD Reg#6, 3
        MV Reg#5, Reg#6
END main
Used Register: 11
```

test3.txt

```
1  func(){
2      a = 3;
3      b = 7;
4      testVal = 576;
5
6      IF (testVal < 600) THEN{
7          testVal = testVal + a;
8      }
9      ELSE{
10         a=a+b;
11     }
12
13     -1
14
15     c=100;
16
17     WHILE(c>40){
18         c=c+a+b+testVal+3+a+16;
19     }
20
21     WHILE(c>40);
22
23     d=30;
24 }
```

```
czw4653@ubuntu:~/pl$ ./out test3.txt
Error: WRONG INPUT
```

-값은 인풋으로 들어올 수 없기 때문에 WRONG INPUT 이라는 에러를 띄워 주었다.

test4.txt

```
1 func(){
2     a = 3;
3     b = 7;
4     testVal = 576;
5
6     IF (testVal < 600) THEN{
7         testVal = testVal + a;
8     }
9     ELSE{
10        a=a+b
11    }
12
13    c=100;
14
15    WHILE(c>40){
16        c=c+a+b+testVal+3+a+16;
17    }
18
19    WHILE(c>40);
20
21    d=30;
22 }
```

```
czw4653@ubuntu:~/pl$ ./out test4.txt
Syntax error
```

문법에 맞지 않는 파일을 컴파일하면 Syntax error 를 출력해 준다.