|  |
| --- |
| Programming Project #3  <Bottom up parsing을 이용한 Mini-C 인터프리터> |

|  |  |
| --- | --- |
| 과목 | 컴파일러 |
| 교수님 | 류기열 교수님 |
| 학과 | 수학과 |
| 학번 | 201421155 |
| 이름 | 이안규 |

****

1. 서론

* 과제소개

이번과제는 bottom up parsing기법을 이용해서 lexical analysis를 실행한 토큰들의 syntax analysis를 실행한다. 2차 과제에서는 단순한 계산식만을 위한 syntax tree를 구성하고 계산했지만 이번 과제에서는 if문, while문, block statement까지 포함한 syntax tree를 구성하고 인터프리터로 계산결과를 나타내는 것이다.

* 이번과제에서 구현해야 하는 bison을 이용한 bottom up parsing과 bottom up parsing으로 syntax tree를 생성하는 것과 생성된 syntax tree를 이용하여 계산결과를 출력하는 것까지 구현되었다. 선택사항으로 주어진 함수와 recursive한 형태는 구현하지 못했다. 그리고 lexical error가 여러 개 있는 경우 lexical error의 개수만큼 lexical error가 출력된다. 또한 초기화하지 않은 symbol의 값이 쓰레기 값이 나와야 하는데 배열을 사용하여서 0으로 출력되는 것 같다.

1. 문제 분석

* Grammar rule 분석

start :

line

;

line :

line statement

|

;

statement :

‘ ; ‘

| expression ’ ; ‘

| PRINT expression ‘ ; ‘

| ID ‘ = ‘ expression ‘ ; ‘

| WHILE ‘ ( ‘ expression ‘ ) ‘ statement

| IF ‘ ( ‘ expression ‘ ) ‘ statement ELSE statement

| ‘ { ‘ statement\_list ‘ } ‘

;

statement\_list :

statement

| statement\_list statement

;

expression :

INTEGER

| DOUBLE

| ID

| ‘ – ‘ expression %prec UMINUS

| expression ‘ + ‘ expression

| expression ‘ – ‘ expression

| expression ‘ \* ‘ expression

| expression ‘ / ‘ expression

| expression ‘ < ‘ expression

| expression ‘ > ‘ expression

| expression GE expression

| expression LE expression

| expression NE expression

| expression EQ expression

| ‘ ( ‘ expression ‘ ) ‘

;

|  |  |
| --- | --- |
| Rules | tasks |
| start -> line | Bottom up parsing이 끝나고 프로그램이 종료된다. |
| line -> line statement | statement에서 반환된 노드를 인터프리터 함수에 전달해 인터프리터 해준다. statement에서 반환된 노드를 freeNode에서 동적할당을 해제해준다. |
| line -> ε | Line이 ε으로 변환된다. |
| statement -> ‘ ; ‘ | Node의 타입이 ‘ ; ‘인 Node를 반환한다. |
| statement -> expression ‘ ; ‘ | expression에서 반환된 노드를 반환한다. |
| statement -> PRINT expression ‘ ; ‘ | Node의 타입이 PRINT이고 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 Node를 반환한다. |
| statement -> ID ‘=’ expression ‘ ; ‘ | Node의 타입이 ‘=’이고 ID에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| statement->WHILE ‘( ‘expression‘ )‘ statement | Node의 타입이 WHILE이고 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 statement에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| statement -> IF ‘ ( ‘ expression ‘ )’ statement ELSE statement | Node의 타입이 IF이고 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 첫번째 statement에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 두번째 statement에서 반환되는 노드를 Node안의 노드 포인터배열의 세번째에 연결해주고 Node를 반환한다. |
| statement -> ‘ { ‘ statement\_list ‘ } ‘ | statement\_list에서 반환된 노드를 반환한다. |
| statement\_list -> statement | statement에서 반환된 노드를 반환한다. |
| statement\_list -> statement\_list statement | Node의 타입이 ‘ ; ‘이고 statement\_list에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 statement에서 반환 되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| expression -> INTEGER | 노드의 타입이 INTEGER이고 INTEGER 토큰에서 전달받은 int값을 갖는 노드를 반환한다. |
| expression -> DOUBLE | 노드의 타입이 DOUBLE이고 DOUBLE 토큰에서 전달받은 double값을 갖는 노드를 반환한다. |
| expression -> ID | 노드의 타입이 ID이고 ID 토큰에서 전달받은 symbol값을 갖고 symbol table에서 symbol의 index값을 갖는 노드를 반환한다. |
| expression -> ‘-‘ expression %prec UMINUS | Node의 타입이 ‘-‘이고 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 Node를 반환한다. |
| expression -> expression ‘+’ expression | Node의 타입이 UMINUS이고 첫번째expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| expression -> expression ‘ - ‘ expression | Node의 타입이 ‘+‘이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째expression에서 반환되는 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| expression -> expression ‘ \* ‘ expression | Node의 타입이 ‘ \* ‘이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째expression에서 반환되는 Node안의 노드 포인터배열의 두번째에 연결해주고  Node를 반환한다. |
| expression -> expression ‘ / ‘ expression | Node의 타입이 ‘ / ‘이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| expression -> expression ‘<‘ expression | Node의 타입이 ‘ < ‘이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째 expression에서 반환되는 노드를Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| expression -> expression ‘ >‘ expression | Node의 타입이 ‘ > ‘이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| expression -> expression GE expression | Node의 타입이 GE이고 첫번째 expression에서 반환 되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째expression에서 반환 되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고 Node를 반환한다. |
| expression -> expression LE expression | Node의 타입이 LE이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째expression에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고  Node를 반환한다. |
| expression -> expression NE expression | Node의 타입이 NE이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째expression에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고  Node를 반환한다. |
| expression -> expression EQ expression | Node의 타입이 EQ이고 첫번째 expression에서 반환되는 노드를 Node안의 노드 포인터배열의 첫번째에 연결해주고 두번째expression에서 반환되는 노드를 Node안의 노드 포인터배열의 두번째에 연결해주고  Node를 반환한다. |
| expression -> ‘ ( ‘ expression ‘ ) ‘ | ( )안의 expression에서 반환 된 노드를 반환한다. |

* Grammar rule의 LALR(1) parsing 가능 여부(bison 이용)

Grammar rule을 bison을 이용하여 실행했을 때 특별한 에러나 문제점은 발견되지 않았다. Operator 같은 경우에는 우선순위를 정해서 정의해주었더니 parsing중 에러는 발생하지 않았다.

1. 설계

* 주요 자료구조

Bottom up parsing 도중 생성되는 symbol들을 저장하기 위해 symbol\_table을 배열형태로 구현하였다. Symbol\_table의 각 index에는 id를 저장할 수 있는 길이가 20인 배열과 symbol에 해당하는 값을 저장하기 위한 double형 변수 rv가 구조체로 정의 되어있다. syntax tree 노드에는 5가지 종류가 있다. 첫번째 id의 symbol값과 symbol이 symbol\_table에 저장되어 있는 index값을 가지고 있는 nodeId 노드가 있다. 두번째 INTEGER 토큰으로 lexical analysis 된 토큰의 값을 저장하기 위한 int 변수를 가지고 있는 nodeCon 노드가 있다. 세번째 DOUBLE 토큰으로 lexical analysis 된 토큰의 값을 저장하기 위한 double 변수를 가지고 있는 nodeReal 노드가 있다. 네번째 OPERATOR로 구분한 operator들을 넣어주는 int optn과 네가지 토큰들을 union하여 정의 해준 Node를 포인터 배열로 가지고 있는 nodeOpr 노드가 있다.(if-else문의 경우 최대 3개의 노드가 필요하므로 포인터배열의 길이를 3으로 해주었다. -> expression, statement, statement) 마지막으로 위의 4가지 nodeId, nodeCon, nodeReal, nodeOpr을 union하여 정의해주고 각각 토큰의 타입을 enum으로 정의해준 변수를 가지고 있는 Node를 정의해주었다.

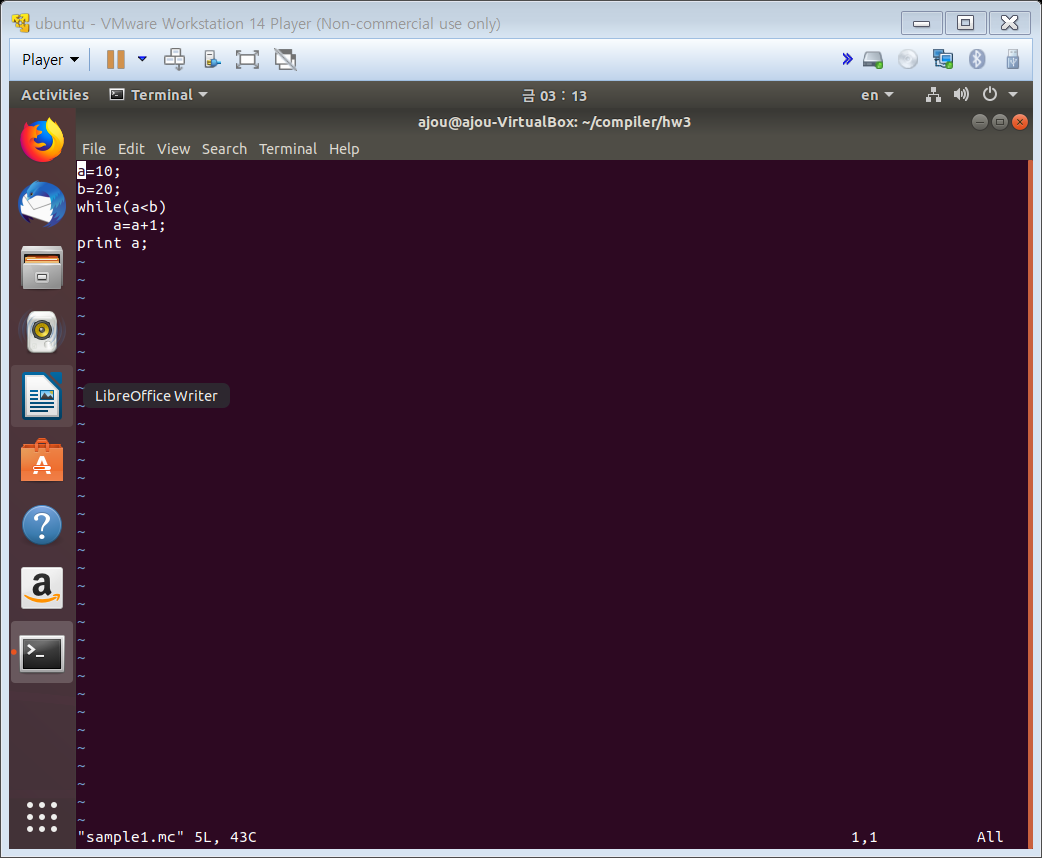
* 모듈 설명

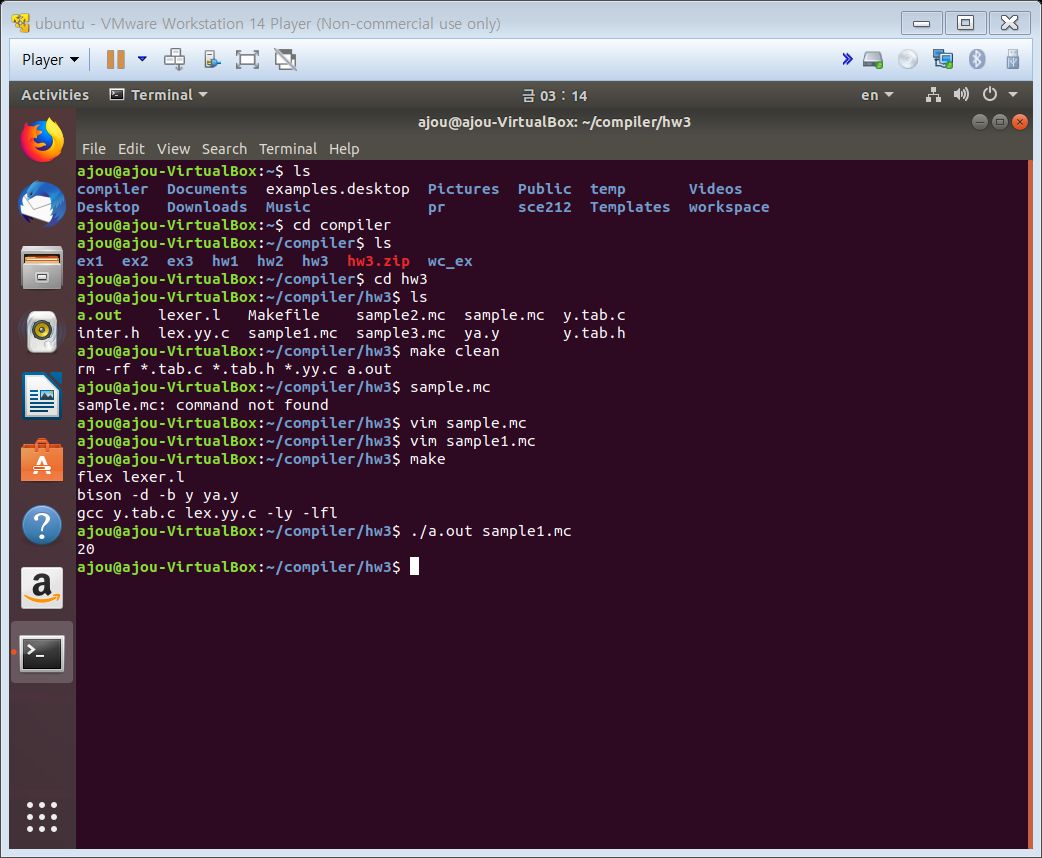
|  |  |
| --- | --- |
| **Variable defined** | |
| typedef enum {typeId, typeCon, typeReal, typeOpr} typeEnum; | Node의 타입을 값으로 넣어주기 위해 정의했다. |
| typedef struct{  char str[20];  int index;  }nodeId; | ID를 인식한 토큰에서 토큰에 해당되는 symbol 값과 symbol table에서 symbol의 index값을 갖는 노드이다. |
| typedef struct{  int conValue;  }nodeCon; | INTEGER를 인식한 토큰에서 토큰에 해당되는 integer값을 갖는 노드이다. |
| typedef struct{  double realValue;  }nodeReal; | DOUBLE을 인식한 토큰에서 토큰에 해당되는 real value를 갖는 노드이다. |
| typedef struct{  int optn;  struct nodeType \* list[2];  }nodeOpr; | Opreator로 구분된 토큰에 해당되는 값을 저장하는 변수(int optn)와 opreator에 필요한 operand들을 포인팅 할 수 있는 포인터배열로 이루어진 노드이다. |
| typedef struct nodeType{  typeEnum type;  union{  nodeId id;  nodeCon con;  nodeReal real;  nodeOpr opr;  };  }Node; | 인터프리터에서 각각 다른 타입의 노드를 계산할 수 없으므로 4개의 노드를 하나의 노드로 묶어주기 위한 구조체 선언이다. 메모리 사용을 줄이기 위해 4개의 노드 타입을 union해주었다. |
| typedef struct{  char symbol[20];  double rv;  }SYMBOL\_TABLE; | Symbol을 저장하기 위한 변수와 symbol의 값을 저장하기 위한 변수를 갖는 구조체이다. |

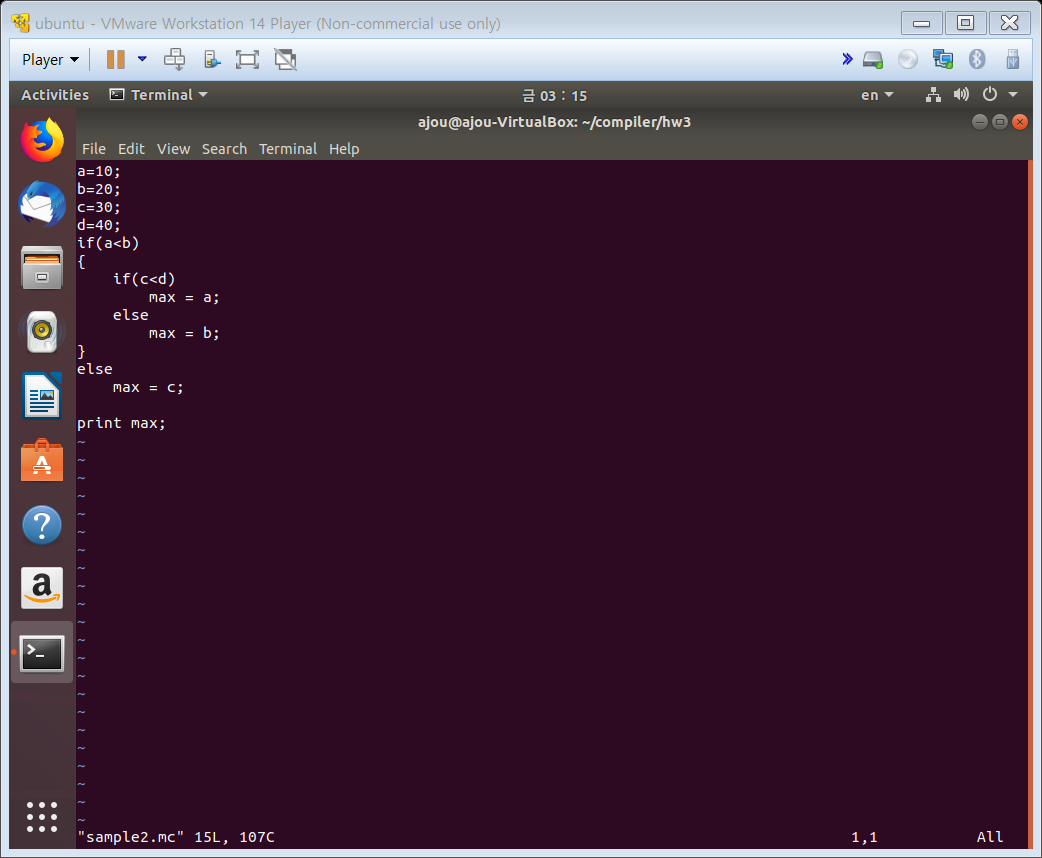
|  |  |
| --- | --- |
| **Syntax analysis** | |
| Node \* con ( int value1 ); | Syntax analysis중 INTEGER토큰이 analysis 됐을 때 INTEGER에 해당되는 값을 갖는 노드를 생성하는 함수이다. |
| Node \* real ( double value2 ); | Syntax analysis중 DOUBLE 토큰이 analysis 됐을 때 DOUBLE에 해당되는 값을 갖는 노드를 생성하는 함수이다. |
| Node \* id ( char \* value3 ); | Syntax analysis중 ID 토큰이 analysis 됐을 때 ID에 해당되는 값과 ID에 해당되는 symbol table의 index의 값을 갖는 노드를 생성하는 함수이다. |
| Node \* opr ( int opr\_type, Node \* pa1, Node \* pa2, Node \* pa3 ); | Operator의 타입 넘버를 갖고 최대 3개의 operand의 노드를 포인팅하는 list를 갖는 노드를 생성하는 함수이다. |
| void freeNode ( Node \* temp1 ); | 동적할당 된 syntax tree 노드를 free해주는 함수이다. |
| void yyerror( char \* msg ); | Error를 출력해주는 함수이다. |
| int find\_symbol\_index( char \* s ); | Symbol의 symbol table에서의 index를 찾는 함수이다. |
| void save\_symbol ( char \* s1 ); | Symbol이 symbol table에 존재하지 않을 때 symbol을 저장하는 함수이다. |
| %union{  int int\_value;  double real\_value;  char id[20];  Node \* ptr;  }; | Multiple value type을 사용하기 위해 정의해준다. |
| %token <int\_value> INTEGER  %token <real\_value> DOUBLE  %token <id> ID  %token WHILE IF PRINT ELSE  %type <ptr> statement expression  statement\_list | 각 토큰의 타입을 정의해주고 non-terminal의 타입 역시 정의해준다. |

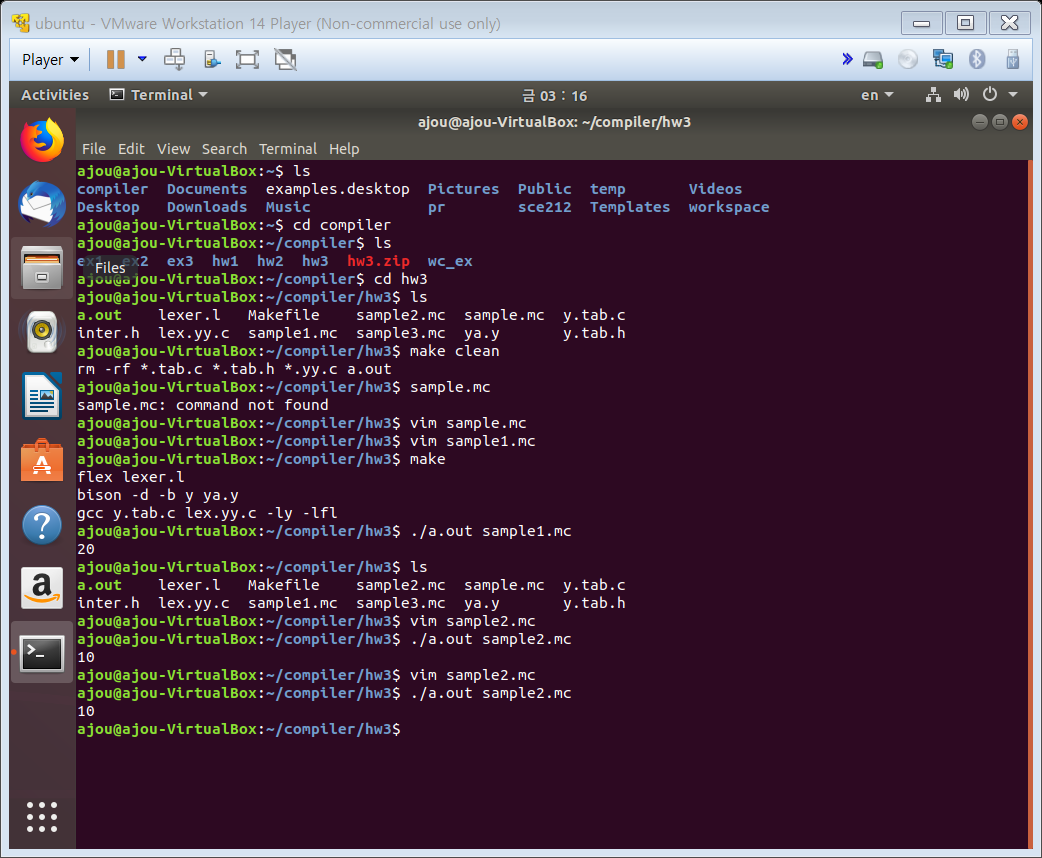
|  |  |
| --- | --- |
| **Syntax tree interpreter** | |
| double cal ( Node \* p ) | 생성된 syntax tree를 전달받아 interpreter 하여 값을 반환해준다. 함수 형태는 recursive한 형태로 구성한다.   1. Node의 타입이 typeId(심볼 테이블에 해당하는 값), typeCon, typeReal인 경우 값을 그대로 반환한다. 2. Node의 타입이 typeOpr인 경우 operator의 종류를 int optn을 통해 확인하다. 3. optn이 WHILE이면 while문을 실행시킨다. while문의 expression에는 포인터배열의 첫번째에 포인팅 된 노드가 실행되고 while문 안의 statement는 포인터배열의 두번째에 포인팅 된 노드가 실행된다. 4. optn이 IF이면 if문을 실행시킨다. if문의 expression에는 포인터배열의 첫번째에 포인팅 된 노드가 실행되고 if의 실행 statement는 포인터배열의 두번째 포인팅 된 노드, else의 실행 statement는 포인터배열의 세번째 포인팅 된 노드가 실행된다. 5. optn이 PRINT이면 포인터배열의 첫번째에 포인팅 된 노드가 실행된다. 여기서 print될 값의 타입을 판단하여 print해준다. 6. optn이 ‘;’이면 포인터배열의 첫번째에 포인팅 된 노드가 실행되고 두번째에 포인팅된 노드를 반환한다. 그 이유는 statement의 grammar룰은 더욱 더 확장될 수 있기 때문이다. 7. optn이 ‘=’이면 포인터배열의 첫번째에 포인팅 된 노드가 가리키는 symbol\_table에 두번째에 포인팅 된 노드의 값을 넣어주고 반환한다. 8. optn이 UMINUS이면 포인터배열의 첫번째에 포인팅 된 노드의 계산값에 ‘–‘된 값을 반환한다. 9. optn이 ‘+’이면 포인터배열의 첫번째에 포인팅 된 노드와 두번째에 포인팅 된 노드의 합을 반환한다. 10. optn이 ‘-‘이면 포인터배열의 첫번째에 포인팅 된 노드의 값에서 두번째에 포인팅 된 노드의 값의 차이를 반환한다. 11. optn이 ‘\*’이면 포인터배열의 첫번째에 포인팅 된 노드와 두번째에 포인팅 된 노드의 곱을 반환한다. 12. optn이 ‘/’이면 포인터배열의 첫번째에 포인팅 된 노드의 값에서 두번째에 포인팅 된 노드의 값으로 나눈 값을 반환한다. 13. optn이 ‘<’이면 포인터배열의 첫번째에 포인팅 된 노드의 값과 두번째에 포인팅 된 노드의 값을 ‘<’ 비교한 결과를 반환한다. 14. optn이 ‘>’이면 포인터배열의 첫번째에 포인팅 된 노드의 값과 두번째에 포인팅 된 노드의 값을 ‘>’ 비교한 결과를 반환한다. 15. optn이 GE이면 포인터배열의 첫번째에 포인팅 된 노드의 값과 두번째에 포인팅 된 노드의 값을 ‘>=’ 비교한 결과를 반환한다. 16. optn이 LE이면 포인터배열의 첫번째에 포인팅 된 노드의 값과 두번째에 포인팅 된 노드의 값을 ‘<=’ 비교한 결과를 반환한다. 17. optn이 NE이면 포인터배열의 첫번째에 포인팅 된 노드의 값과 두번째에 포인팅 된 노드의 값을 ‘!=’ 비교한 결과를 반환한다. 18. optn이 EQ이면 포인터배열의 첫번째에 포인팅 된 노드의 값과 두번째에 포인팅 된 노드의 값을 ‘==’ 비교한 결과를 반환한다. |

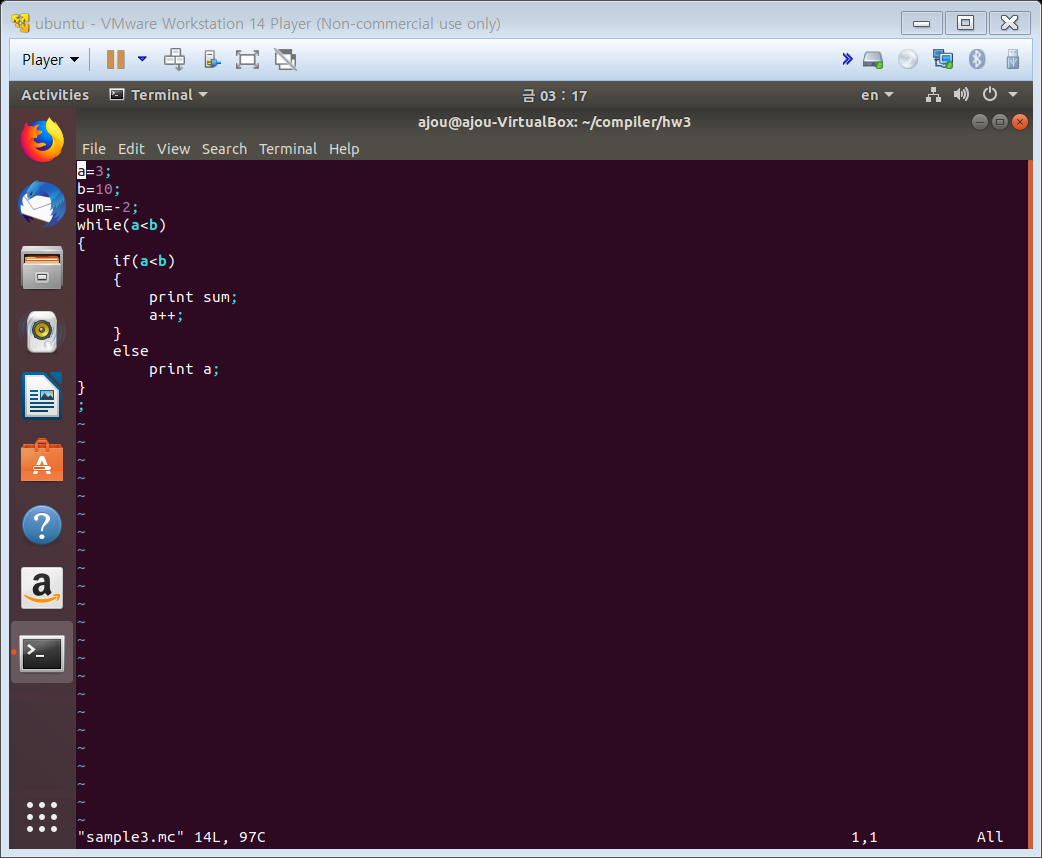
1. 수행결과

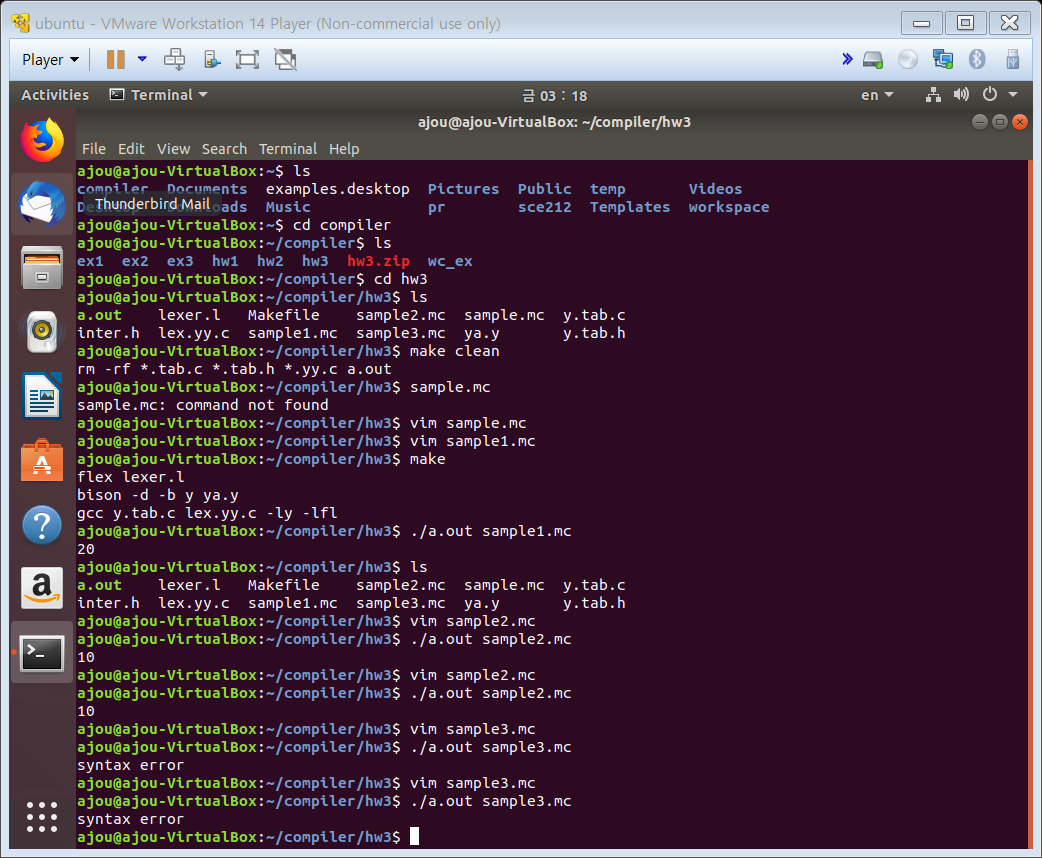
 예시 : sample1.mc

 실행결과

 예시 : sample2.mc

 실행결과

 예시 : sample3.mc

 실행결과