컴파일러 텀 최종 보고서

Oh조

200924413 권성철

201224540 조용래

목차

1. 개요

이번 컴파일러 텀 프로젝트는 3단계로 구성되었다. 첫 단계에서는 컴퓨터가 인식할 수 있도록 언어를 설계하는 과정이었다. 일상생활에서 사람들이 사용하는 언어는 컴퓨터가 처리할 수 있는 능력을 초월하기 때문에 언어를 단순화시키는 작업이 필요하다. 이 작업은 언어를 형식화시키는 과정이 뒤따른다. 이번 텀 프로젝트에서 사용하는 문법은 Unix에서 제공하는 lex와 yacc의 문법에 따라 설계해야 한다. 이런 형식에 맞춰 언어를 설계하면 컴퓨터는 인간이 인위적으로 설계한 언어를 인식할 수 있게 된다. 언어 정의 단계에서는 기존의 잘 설계된 C나 C++을 기반으로 한다. 물론 C나 C++의 기능을 다 쓰겠다는 것은 아니지만 학습의 차원에서 몇 가지 특성들을 뽑아내어 새롭게 설계한 사항들을 포함하여 언어를 정의하였다.

두 번째 단계에서는 우리가 설계한 언어와 이 언어의 조건에 만족하는 사례 사이의 관계를 파악하고 이를 어셈블리 언어로 번역하기 이전에 수행해야 할 작업을 한다. 이 작업에는 Symbol Table과 AST가 필요하다. Symbol Table은 컴파일러 전 단계에서 광범위하게 활용되기 때문에 중요하다. Symbol Table은 프로그램에서 사용되는 변수에 대한 정보를 지속적으로 담고 있다. 우리가 설계한 언어의 문법으로 예제 프로그램에 대한 트리를 생성하였을 경우 실제로 기계가 번역을 할 때 쓰이는 정보보다 훨씬 더 많은 정보를 가지고 있다. 따라서 불필요한 정보들을 트리로부터 잘라내는 과정이 필요하다. 이러한 과정을 통해 생성된 가지가 쳐진 트리를 AST(Abstract Syntax Tree)라고 한다. 실제로 번역에 필요한 정보만 담고 있기 때문에 이 트리가 다음 phase로 전달된다.

세 번째 단계는 앞 단계에서 전달해준 AST를 순회하면서 적절한 기계어 코드를 생성하는 단계이다. 이 때 기계어 코드는 cpu의 종류에 의존하게 되는데 우리는 Intel x86 instruction set으로 번역하였다. 이 코드를 생성하는 도중에 변수와 관련된 트리 노드들을 방문하게 될 때 전 단계에서 보내준 Symbol Table을 참조하게 된다. 그리고 AST에서 ID가 포함된 문법이 타입에 따라 그 구현이 달라지도록 해야 될 때 Symbol Table에서 해당 ID의 타입을 읽어내어 각각에 대해 구현을 다르게 한다. 이 과정을 Semantic Analysis라고 한다. 위의 과정을 거치면 최종적인 Intel x86 코드가 찍히는데 실제 테스트를 해보진 못하였다.

기존에 제출했던 ppt와 보고서에 있는 언어 설계 사항은 상당 부분 생략하였다.

2. Symbol Table

변수는 타입과 이름, 값으로 이루어진다. 컴파일을 할 때 변수에 대한 정보는 앞의 것들과 더불어 다른 정보들도 필요하다. 해당 변수가 메모리상에서 어디에 위치하는지에대한 정보와 초기화 여부, 실행 도중에 바뀌는 값들을 가지고 있어야 한다. Symbol Table은 이러한 정보들을 모두 담고 있어야 한다. 따라서 Symbol Table 설계 시에 타입 이름, 값, 초기화 여부 초기화된 값, 변수의 주소를 포함해야 한다. 여기에 추가적으로 변수가 선언된 라인 수에 대한 정보도 포함시켜 보았다. 다음의 설명들은 Symbol Table을 설계할 때 필요한 자료구조와 Symbol Table에 대한 연산을 정의하는 함수들에 대한 것이다.

Symbol Table의 구조

Symbol Table의 타입을 나타내기 위해서는 문자열이 아니라 정수로 나타내는 것이 효과적이다. 그림 1은 열거형 상수로 타입들을 표현하였다. Symbol Table은 Hash Table로 설계하였다. 이 Table의 최대 크기는 1024이고 그림 1에서 매크로 상수로 정의된 HASHSIZE에서 확인할 수 있다.

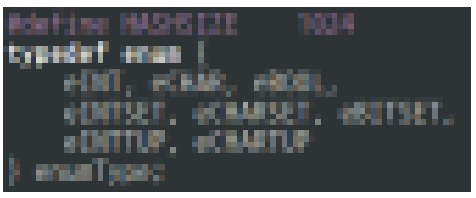


그림 1

그림 2는 flex에서 ID를 만났을 경우 변수이름을 담고 있는 노드를 생성한다. 생성된 노드는 bison으로 전달되고 bison에서 Symbol Table 생성에 이를 활용한다.

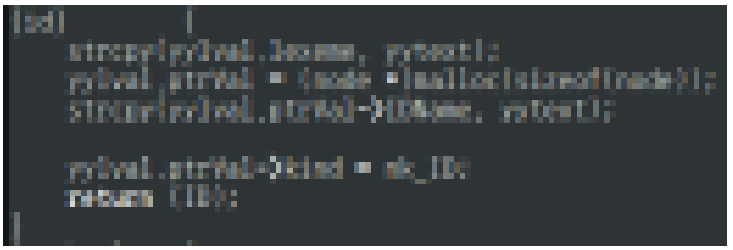


그림 2

그림 3은 Symbol Table을 이루고 있는 구성요소들을 나타낸다. 위에서부터 idName 은 변수의 이름을 저장하고 최대 크기를 20으로 제한하였다. type 은 변수의 타입을 담고 있다. value는 프로그램 실행 도중에 바뀌는 값을 저장한다. lineno 는 변수가 선언된 라인 수이다. inItFlag는 선언된 변수의 초기화 여부를 나타낸다. pIval과 pCval은 변수 선언시 초기화 되었을 때 초기화 값을 담고 있는 곳을 가리키는 포인터이다. size는 배열이 아닐 경우에는 항상 1 이고 배열일 경우 배열의 크기이다. hashNum 은 그림 10의 hash함수의 idName을 넘겨주었을 때 나오는 hash 값이고 이는 Hash Table에서 해당 변수의 위치(행 번호)를 나타낸다. addre ss는 해당 변수의 메모리상의 위치를 나타내고 code gener ation 을 할 때 자주 사용된다.

그림 3

ID 처리

그림 4, 5 , 6, 7은 ID를 처리하는 코드이다. 그림 4 의 idListNode는 id가 한 줄에 여러 개 선언될 경우 해당 ID의 이름들과 크기를 담고 있다. 한번에 최대 10개의 변수를 선언할 수 있으며 각각의 변수의 최대 이름 크기는 20이다. 이 노드는 ID를 인식하는 문법(그림 6의 id_list)에서 ID에 대한 정보들을 담고 상위 문법(그

림 7의 id_dec)으로 올려준다.

```
typedef struct _idListNode {
    // At most 10 variables and 20 characters at once.
    char name[10][20];
    unsigned idNum;
} idListNode;
```

그림 4

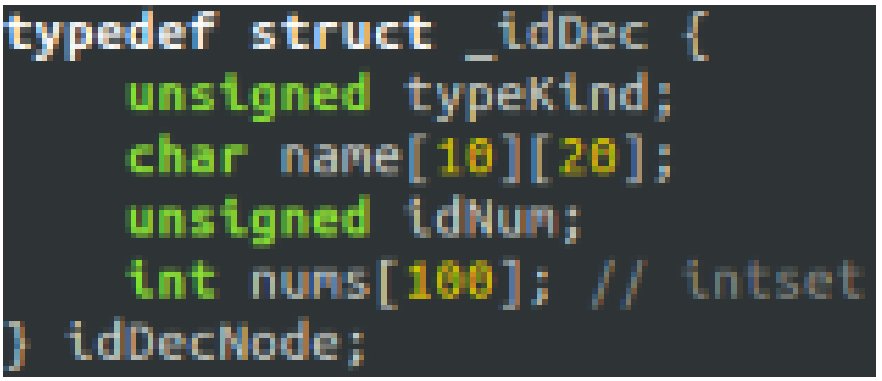


그림 5

그림 6과 7은 bison에서 ID를 처리하는 문법 코드이다. Id_list 에서는 변수의 개수와 변수들의 이름을 담는 노드를 생성하고 id_dec 노드로 올려준다. id_dec 노드는 그림 7에서 보다시피 타입에 대한 정보를 포함한다. 그림 7을 보면 ID를 선언하는 문법은 초기화가 된 경우와 되지 않은 경우로 구분된다. 초기화가 된 경우에는 id_dec_tail 문법에서 초기화에 대한 정보를 포함하게 된다. 초기값이 없는 경우에는 Symbol Table에 삽입하는 InsertSymbol 함수에서 초기화에 관한 인수를 NULL로 넣어주었다. 초기값이 있는 경우에는 id_dec_tail에서 인식한 정보를 InsertSymbol 함수로 넘겨준다.

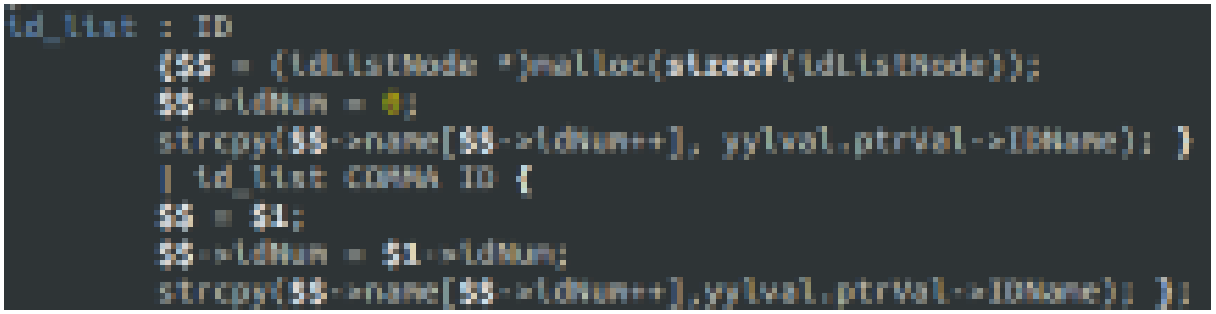


그림 6

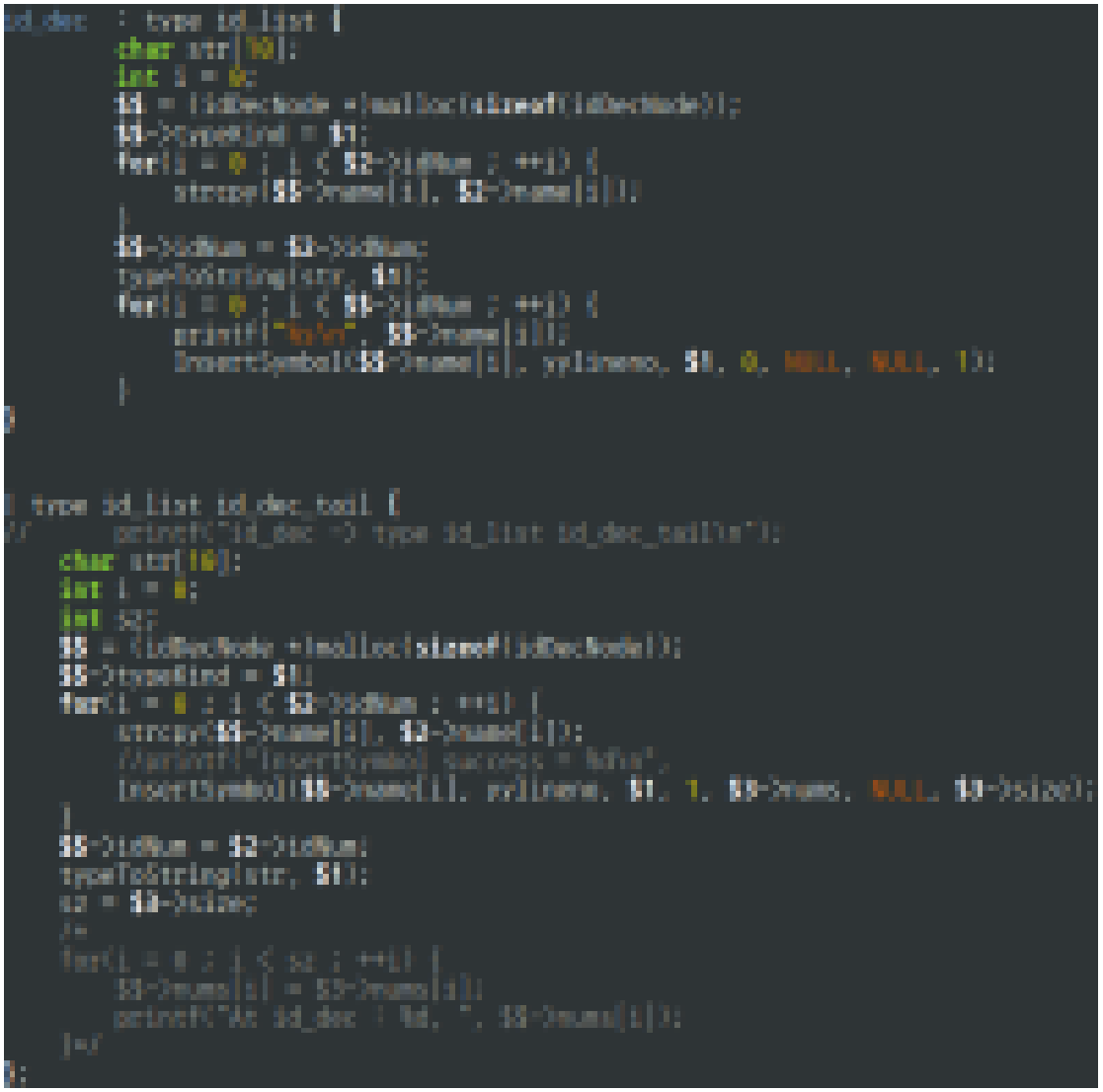

id_dec : type id_list {
char str[10];
int i = 0;
$$ = (idDecNode *)malloc(sizeof(idDecNode));
$$->typeKind = $1;
for(i = 0 ; i < $2->idNum ; ++i) {
strcpy($$->name[i], $2->name[i]);
}
$$->idNum = $2->idNum;
typeToString(str, $1);
for(i = 0 ; i < $$->idNum ; ++i) {
printf("%s\n", $$->name[i]);
InsertSymbol($$->name[i], yylineno, $1, 0, NULL, NULL, 1);
}
}
| type id_list id_dec_tail {
// printf("id_dec -> type id_list id_dec_tail\n");
char str[10];
int i = 0;
int sz;
$$ = (idDecNode *)malloc(sizeof(idDecNode));
$$->typeKind = $1;
for(i = 0 ; i < $2->idNum ; ++i) {
strcpy($$->name[i], $2->name[i]);
//printf("InsertSymbol success = %d\n",
InsertSymbol($$->name[i], yylineno, $1, 1, $$->name, NULL, $$->size);
}
$$->idNum = $2->idNum;
typeToString(str, $1);
sz = $$->size;
/*
for(i = 0 ; i < sz ; ++i) {
$$->name[i] = $$->name[i];
printf("At id_dec : %d, ", $$->name[i]);
}*/
};

그림 7

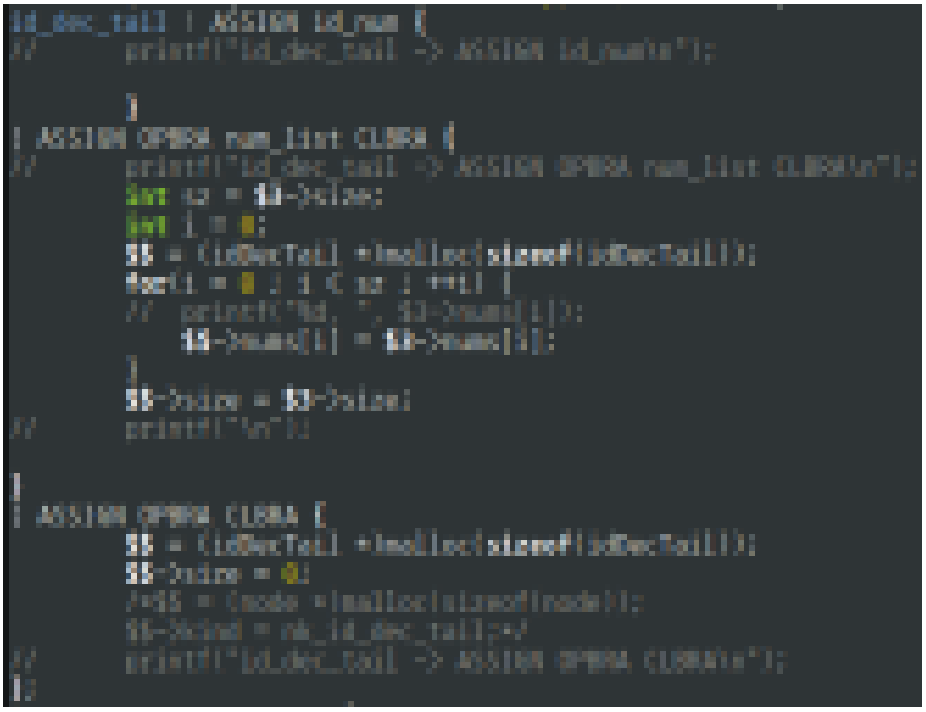


그림 8

Symbol Table에 대한 연산

Symbol Table에 값을 삽입하기 전에 기존에 있던 symbol인지 아닌지 여부를 알아야 할 필요가 있다. 이를 처리해주는 함수가 LookUpSymbol 이다. 그림 12의 LookUpSymbol 함수를 보면 symbol이 이미 존재하는 경우 0을 리턴하고 존재하지 않는 경우에는 1을 리턴한다. LookUpSymbol 함수는 InsertSymbol 함수에서 Symbol Table에 새로운 행을 삽입하기 전에 사용된다. ReadSymbol 함수는 해당 변수가 이미 Symbol Table에 있다면 해당 행을 리턴해준다. ReadSymbol 함수는 code generation 단계에서 변수에 대한 정보가 필요할 때마다 사용된다. 그림 10의 InsertSymbol 함수는 말 그대로 symbol을 Symbol Table에 삽입하는 함수이다.

그림 9

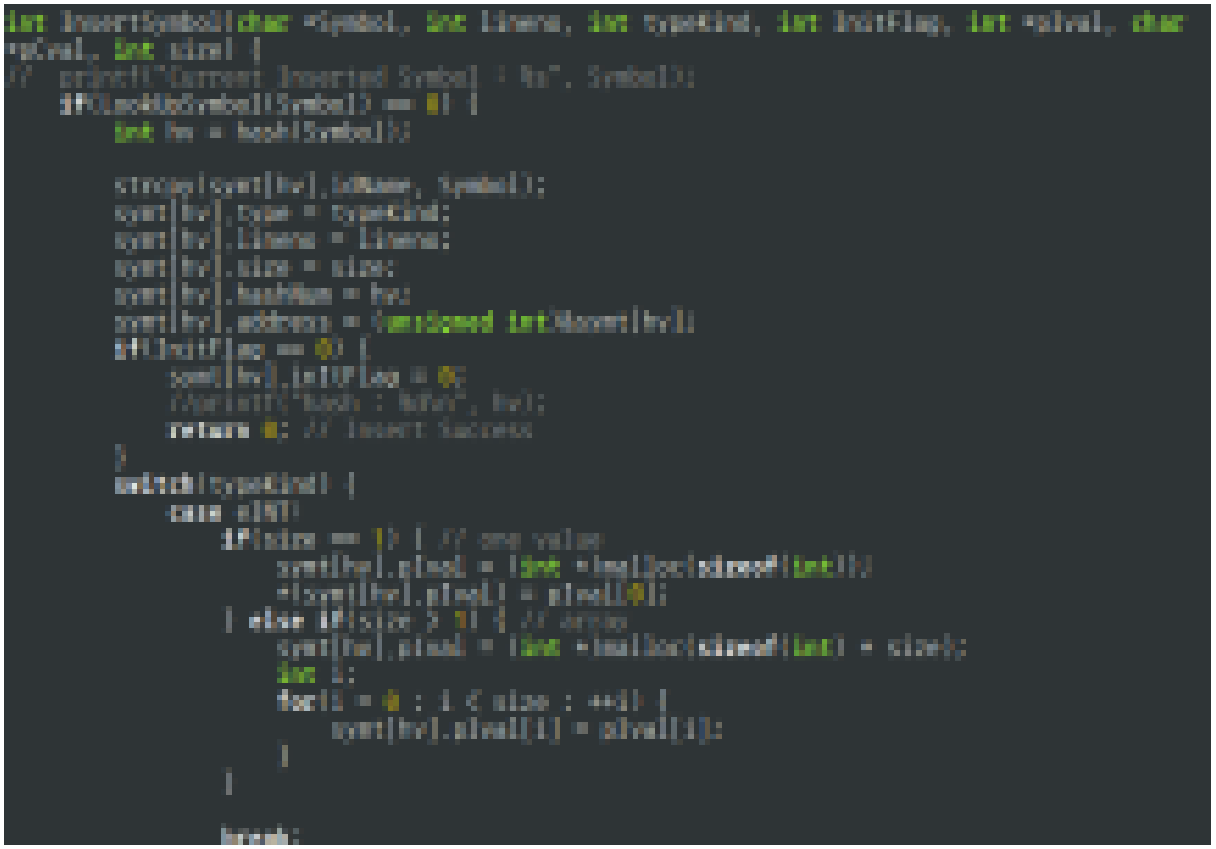


그림 10

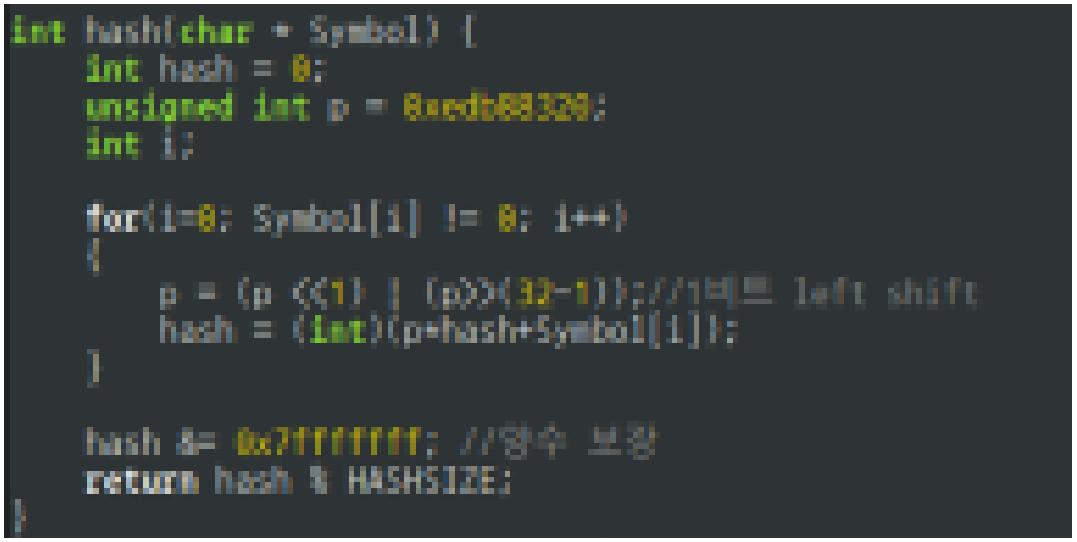


그림 11

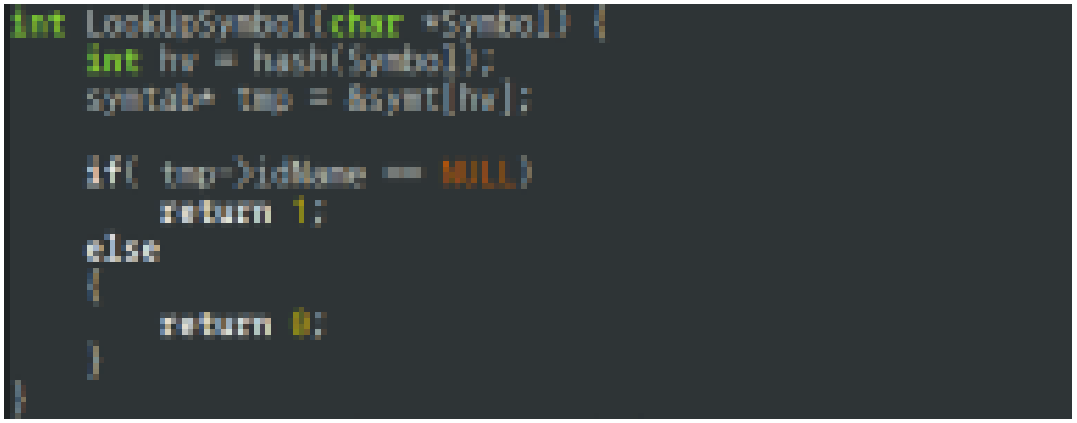


그림 12

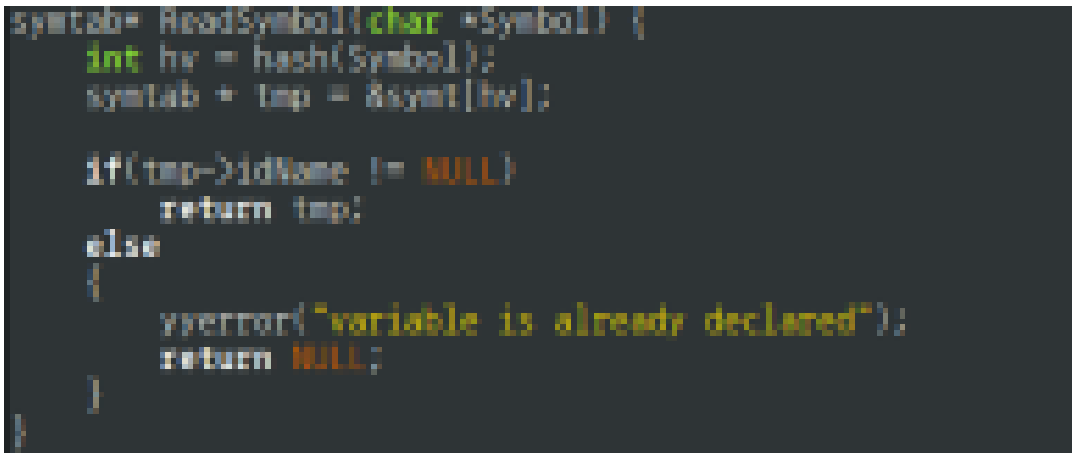
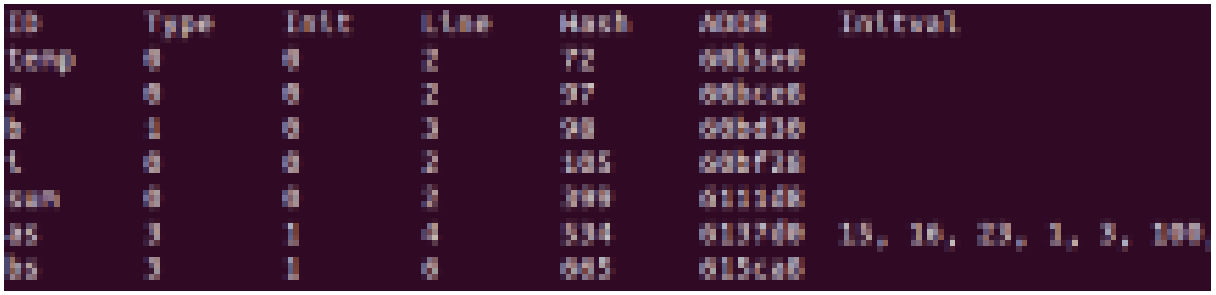


그림 13

그림 14는 위의 함수에서 수행되어 생성된 Symbol Table을 출력하는 함수이다. 그림 15는 샘플 코드에서 생성된 Symbol Table을 print Symt 함수로 출력한 결과이다.

그림 14



ID	Type	Init	LLno	Hash	ADDR	Initval
temp	0	0	2	72	00b5e0	
a	0	0	2	97	00bce0	
b	1	0	3	98	00bd30	
i	0	0	2	105	00bf20	
sum	0	0	2	999	011dd0	
as	3	1	4	534	0137d0	15, 16, 23, 1, 3, 100,
bs	3	1	6	665	015ca0	

그림 15

3. AST

샘플 코드를 제시된 모든 문법으로 트리 형태로 표현하였을 경우 이를 파싱 트리(또는 concrete tree) 라고 한다. 이 트리는 code generation 시 불필요한 정보를 많이 담고 있다. 따라서 꼭 필요한 정보만 담고 있는 트리가 필요한데 이를 추상 구문 트리(Abstract Syntax Tree) 라고 한다.

그림 16 은 추상 구문 트리에서 사용되는 노드 구조를 나타낸다. 노드에 대한 설명을 하기 전에 우리는 C 언어로 작성하였기 때문에 virtual function 같은 객체 지향형 개념을 사용할 수 없다. 따라서 imperative 방식으로 구현하였다. 하나의 노드가 담고 있는 데이터는 AST에 포함되는 문법 모두를 아우르는 구조가 필요하다 보니 해당 문법에서 꼭 필요하지 않은 데이터까지 포함하게 되었다.

노드 구조는 그림 16과 같다. 노드 종류를 나타내는 kind 변수는 nodeKind 라는 열거형 상수로 정의된다. 구조체 _node 타입의 childPointer 는 노드의 자식 노드를 가리킨다. 이 때 자식 노드의 가변적인 경우를 하나씩 다 처리해주면 프로그래밍이 복잡해 지는 경향이 있고 노드 개수가 많아 봐야 한 자리 수라는 경우를 볼 때 최대 자식 노드를 적당한 수로 제한 하는 것이 충분히 합리적인 방향인 것 같았다.

idListPointer 는 id_dec노드에서 선언된 ID에 대한 정보들을 담는 포인터이다. 다른 문법에서는 사용되지 않는다. numOfChild는 자식 포인터의 개수다. 실제로는 자식 포인터를 이어 줄 때 마지막 자식 노드 다음에는 NULL을 삽입해주기 때문에 잘 쓰이지는 않는다. Ival은 정수값을 가질 필요가 있는 노드, cval은 문자값을 가질 필요가 있는 노드, IDName는 ID노드의 이름을 담고 있다.

트리가 실제로 형성되는 곳은 bison에서 각 문법에 대한 액션 코드에서 이다. 예를 들면, 그림 17 을 보면 for_stmt 를 보면 먼저 노드 크기만큼의 메모리를 할당해 주고 자식 노드로 삼을 필요가 있는 assign_stmt, relation _expr, assign_stmt, body를 childPointer 에 대입해 준다. 그 다음은 NULL을 넣어주는데 이는 트리 순회시 필요하다. 마지막으로 자식 노드의 개수를 정해준다.

```
typedef struct _node {
    nodeKind kind;
    // Each node can have 10 children at most
    struct _node *childPointer[10];
    struct _IdListNode *idListPointer;
    unsigned numOfChild;
    int ival;
    char cval;
    char IDName[20];
} node;
```

그림 16

그림 18은 추상 구문 트리를 출력하는 함수이다. 방문 순서는 깊이 우선 탐색에다가 pre-order 방식을 사용하였다. 그림 19는 샘플 코드에 대한 추상 구문 트리를 출력한 결과이다. 보다시피 번역에 불필요한 문법은 다 제거되었다.

```
int depth = 0;
int idx = 0;
char buf[10];
int visitOrder = 1;
void printTree(node* root ) {
    int i = 0;
    while(root->childPointer[i] != NULL) {
        for(idx = 0 ; idx < depth ; ++idx) {
            printf("\t");
        }
        KindToString(root->childPointer[i]->kind, buf);
        printf("%d-%s",visitOrder++, buf);
        if(strcmp(buf, "ID") == 0) {
            printf("(%s)", root->childPointer[i]->IDName);
        } else if(strcmp(buf, "NUM") == 0) {
            printf("(%d)", root->childPointer[i]->ival);
        }
        printf("\n");

for_stmt : FOR OPPAR assign_stmt SEMICOLON relational_expr SEMICOLON assign_stmt CLPAR body
{
        $$ = (node *)malloc(sizeof(node));
        $$->kind = nk_for_stmt;
        $$->childPointer[0] = $3;
        $$->childPointer[1] = $5;
        $$->childPointer[2] = $7;
        $$->childPointer[3] = $9;
        $$->childPointer[4] = NULL;
        $$->numOfChild = 4;
// printf("for_stmt -> FOR OPPAR assign_stmt SEMICOLON relational_expr SEMICOLON
assign_stmt CLPAR body\n");
};
```

그림 17

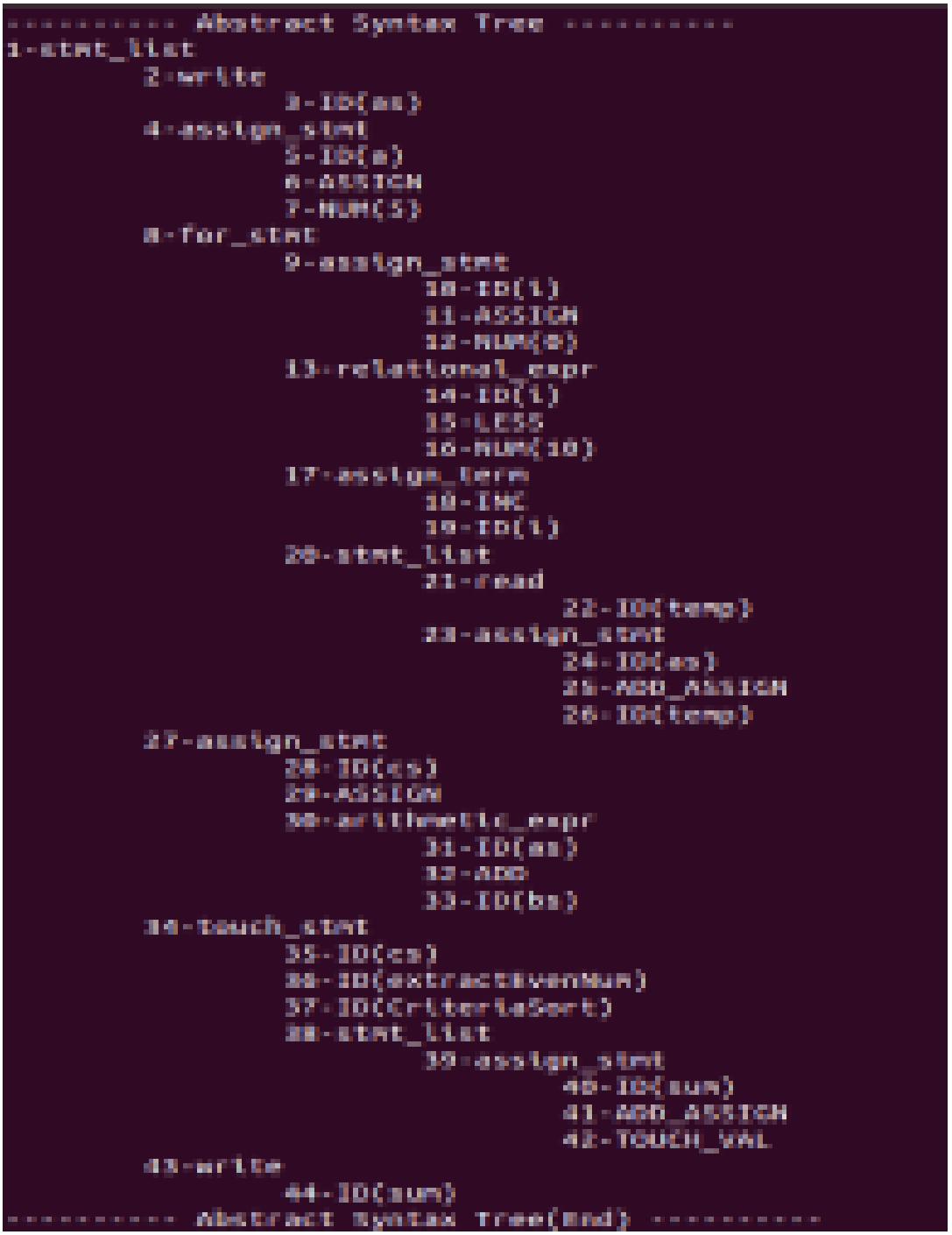

---------- Abstract Syntax Tree ----------
1-stmt_list
2-write
3-ID(as)
4-assign_stmt
5-ID(a)
6-ASSIGN
7-NUM(5)
8-for_stmt
9-assign_stmt
10-ID(i)
11-ASSIGN
12-NUM(0)
13-relational_expr
14-ID(i)
15-LESS
16-NUM(10)
17-assign_term
18-INC
19-ID(i)
20-stmt_list
21-read
22-ID(temp)
23-assign_stmt
24-ID(as)
25-ADD_ASSIGN
26-ID(temp)
27-assign_stmt
28-ID(cs)
29-ASSIGN
30-arithmetic_expr
31-ID(as)
32-ADD
33-ID(bs)
34-touch_stmt
35-ID(cs)
36-ID(extractEvenNum)
37-ID(CriteriaSort)
38-stmt_list
39-assign_stmt
40-ID(sum)
41-ADD_ASSIGN
42-TOUCH_VAL
43-write
44-ID(sum)
---------- Abstract Syntax Tree(End) ----------

그림 19

4. Code Generation

Code Generation 단계는 이전 단계에서 만들어낸 추상 구문 트리를 바탕으로 실제적인 기계어 코드를 만들어 내는 과정이다. 우리 조는 Target CPU는 Intel x86로 정하였다. 코드를 만들어 내는 과정은 트리를 순회하며 적당한 순간 마다 어셈블리 언어를 찍어낸다. 그림 20은 트리를 순회하는 함수이다. 순회 방법은 AST의 순회 방법과 같다. 대신 노드를 방문할 때마다 노드의 종류를 검사하여 종류에 맞는 어셈블리 언어를 출력하는 함수를 호출한다.

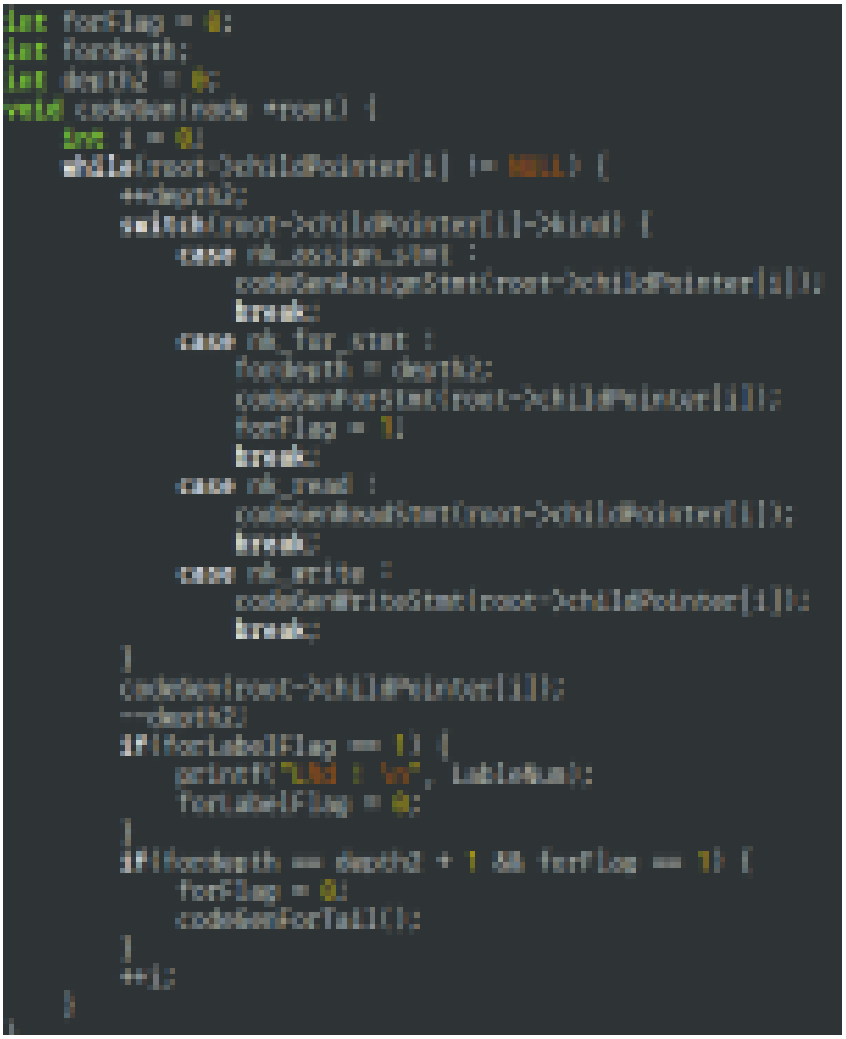


그림 20

그림 21은 노드 종류가 nk_for_stmt 일 때 어셈블리어를 출력하는 함수이다. for 구문에서 사용할 레지스터에 들어있는 값은 미리 스택에 저장해 둔다. 해당 레지스터를 다 쓰고난 후 pop 을 통해 기존의 값으로 복구시킨다.

그림 21

그림 22는 Write함수에 대한 어셈블리어이다. 먼저 인수로 받은 변수의 이름을 심볼 테이블로부터 읽어서 타입을 확인한다. 이후 타입에 따라 정수형일 경우, intset일 경우에 따라 다르게 구현된다.

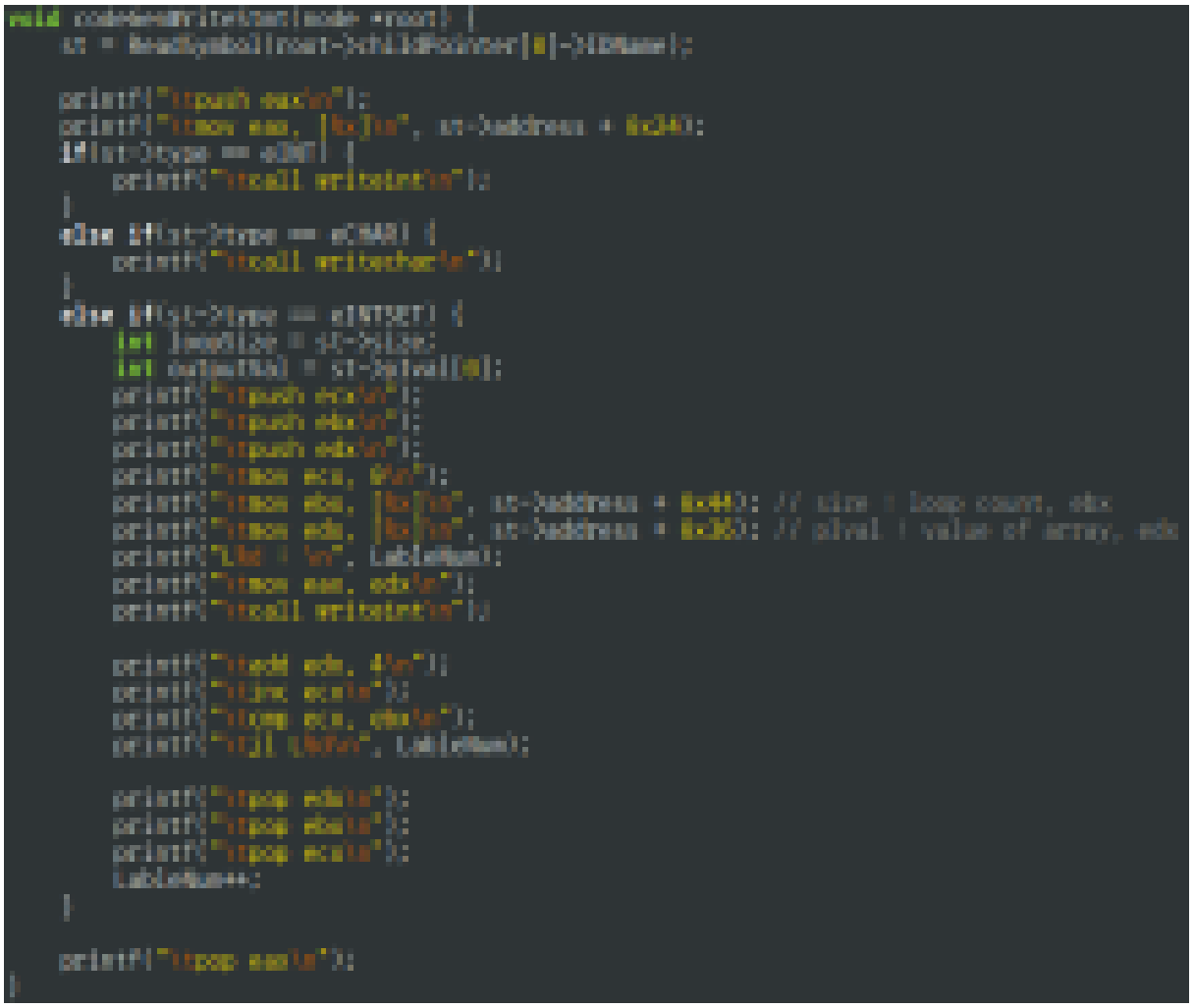


그림 22

아래 그림들은 샘플 코드에 대한 code generation 결과이다.

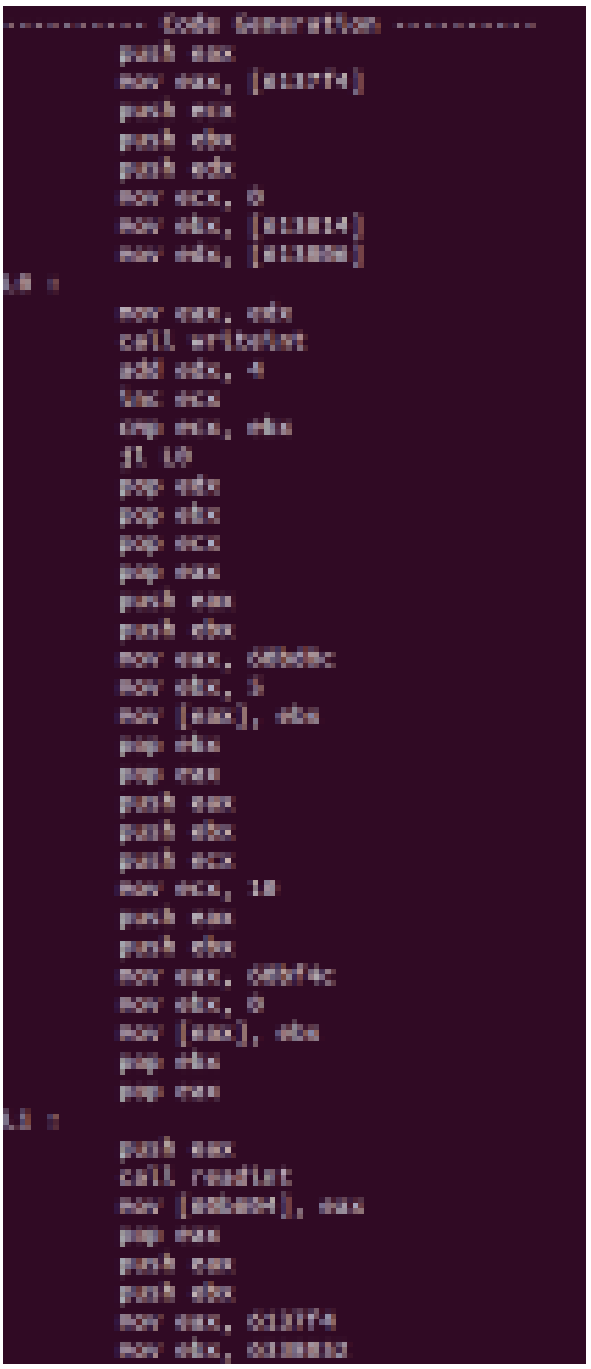


그림 23

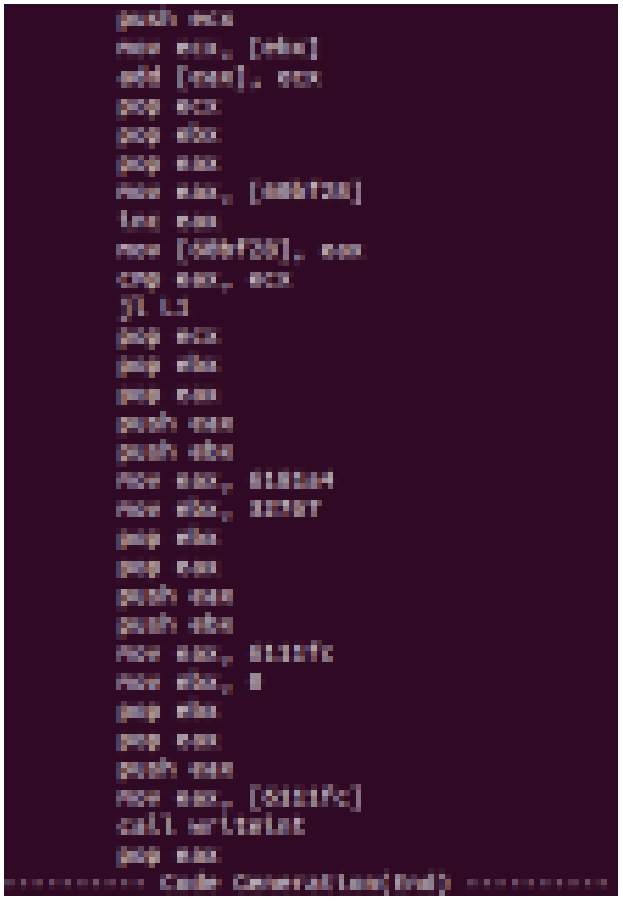


그림 24

5. 일정

10월	11월	12월
언어 정의 설계	AST & Symbol Table	AST & Symbol Table & Code Generation

6. 후기

이번 컴파일러 텀을 수행하면서 많은 난관이 있었다. 실제 강의에서 배운 내용을 토대로 개성 있는 컴파일러를 만들기에는 여러 가지 힘든 점들이 있었다. 코딩 하다 막히는 부분이 생겼을 경우 인터넷을 통해 각종 자료들을 알아보고 제안한 설계에 맞도록 변형하는 과정에서 많은 것을 배웠다. 설계 초기에는 컴파일러를 만든다는 것에 대해 감도 잡히지 않았는데 강의를 들으며 전체적인 흐름을 잡을 수 있었다. 결과적으로 100% 완성된 컴파일러를 만들지는 못

하였으나 제시한 샘플코드를 컴파일 하는 과정에서 방대한 양의 코딩을 하게 되었고 컴파일러 제작의 복잡함을 알 수 있었다. 이런 복잡한 컴파일러를 다루는 과정에 필요한 지식은 이론적인 것과 실천적인 것으로 구분된다. 전자는 샘플 코드를 정의한 문법에 맞게 파싱하는 원리와 실제 컴파일러 구축에 필요한 Symbol Table과 트리 구성 방식 등이 있고, 후자는 이론적 지식을 바탕으로 하여 실질적으로 flex와 bison을 활용하여 이론적 개념들을 손으로 직접 코딩하는 과정이다. 특히 후자의 경우 직접 코딩을 하면서 실제 컴파일러가 어떤 식으로 작성되어 있는지에 대한 감을 갖게 되었고, 실제로 제안한 설계사항을 구현하게 된다면 막대한 양의 코드가 나올 것이라 예상할 수 있었다.

이번 학기 동안 컴파일러 과목을 수강하면서 많은 것들을 배운 것 같아 참 뿌듯했다.