컴파일러 기말 프로젝트 보고서

소프트웨어학과 2017038031 김범철

1. 개요 및 후기

본 프로젝트에서는 음악을 재생시킬 수 있는 컴파일러를 구현해보기로 했다. 사용자가 원하는 음악을 만들어서 재생할 수 있도록 사용자 입장에서 필요한 정보들을 바탕으로 문법을 구성하였으며, C언어에 내장되어있는 라이브러리를 사용해서 비프음을 출력하여 정확한 음정을 낼 수 있도록 했다. 인터넷에 음악 컴파일러를 검색해봐도 잘 나오지 않고 자료도 많이 없었던지라 문법을 구성하고 직접 구현해보는데 많은 시간이 필요했다. 평소에 음악을 취미로 할 때도 건반이나 작곡 프로그램만 사용해봤기 때문에 주파수와 길이를 바탕으로 소리를 생성해내는 저레벨(Low-level) 수준의 음악 생성 프로그램은 만드는 것도 처음이고 직접 써보는 것도 처음이었다.

원래는 외부 음원을 가져와서 재생시키도록 구현하려 했으나 음원을 재생하는 함수가 IDE마다 호환이 잘 안되고, 매끄럽게 재생하기 위한 라이브러리를 따로 설치하면 조교님이 실행하실 때 불편할까봐 비프음으로 했다. 컴파일러를 통해 화음 재생도 해보고 싶어서 쓰레드를 이용하여 동시에 실행하도록 문법도 다 짜고 구현까지 해놨는데 아쉽게도 비프음은 주파수로 연주하는 원리라 처음부터 동시 재생이 불가능했다. 이 점이 가장 아쉬웠으나 사용자가 원하는 음을 문법에 맞게 연주할 수 있는 프로그램을 만들었다는 것에서 보람을 느낀다.

이 컴파일러를 발전시켜서 사용자로부터 입력받는 값을 건반을 누르는 정보와 연동시키면 디지털 피아노를 직접 만들 수도 있을 거라는 생각이 든다. 목재 건반을 만들고 Iot 장비를 활용해서 어떤 음을 어느 세기로, 얼마나 길게 눌렀는지 측정하여 컴파일러의 인자로 받아들이면, 문법 파싱 후 결과 값을 스피커로 출력해주는 방법이다. 디지털 피아노를 직접 만드는 것은 대형 프로젝트일 것 같고, 학부생 수준에서는 저가의 MIDI 건반을 구입해서 컴퓨터와 연결시킨 후 건반을 누른 정보를 받아들이는 것까지 생각해볼 수 있을 것 같다. 그렇게 되면 단순히 소리를 출력하는 것에 그치지 않고 악보를 출력해주는 작곡 프로그램도 제작 가능하다. 기존의 작곡 프로그램은 노드를 일일이 자판으로 쳤어야 했지만 건반을 눌러서 자동으로 음표가 그려지게 되면 상당히 편리할 것 같다.

2. 문법

사용자가 작성하는 프로그램은 start와 end로 끝난다.
start와 end사이에 올 수 있는 명령은 다음과 같다.

- SPEED 값;
  입력 값에 따라 음악의 속도를 조정한다.
- PLAY node_list ---
  node_list에 입력한 노드 정보를 재생한다.
- REP node_list ---
  node_list에 입력한 노드 정보를 한 번 반복하여 재생한다. (도돌이표)

node_list에 입력하는 형태는 다음과 같다.

- 음표 : _길이음정
  해당 길이만큼 음정을 연주한다.
  음정은 도부터 시까지 알파벳 대문자로 작성한다.
  반음 올림은 알파벳 뒤에 s를 붙이고, 쉼표는 R로 작성한다.

  [예시]
  - '도'를 2박자 연주 : _2C
  - '라'를 6박자 연주 : _6A
  - '파#'을 3박자 연주 : _3Fs
  - 2박 쉼표 : _2R

- 옥타브 : OCT 값
  입력 값에 따라 옥타브를 설정한다.
  피아노 건반의 맨 아래부터 위까지 1에서 8까지의 숫자로 조정한다.

  [예시]
  - 4옥타브 도 : OCT 4 _2C
  - 5옥타브 라 : OCT 5 _2A

섬집아기의 첫 부분을 해당 문법으로 구현하면 아래와 같다.

```
start
SPEED 270;
PLAY
OCT 4 _2C _2F _2G _2A _2G _2F _10G _2R
       _4A _2F _2G _2F _2D _10C _2R
---
end
```

위 문법을 악보로 구현하면 아래와 같다.

3. 구현 설명

먼저 Lex에 필요한 토큰을 정의한다.

```
%%
{ws}      {/* do nothing */}
{nl}      {lineno++; }
;                 {return(STMTEND); }
start             {return(START); }
end               {return(END); }
_                 {return(PLAY); }
PLAY      {return(P); }
REP               {return(REP); }
---               {return(REPEND); }
SPEED     {return(SPEED);}
OCT               {return(OCT); }
{id}      {strcpy(s, yytext); temp=insertsym(s); yylval=MakeNode(ID, temp);
return(ID);}
{digit}+          {sscanf(yytext, "%d", &temp); yylval=MakeNode(NUM, temp);
return(NUM);}
.                 {printf("invalid token %s\n", yytext); }
%%
```

PLAY, REP, ---, SPEED 등 사용자로부터 인식해야 할 특별한 단어들은 Yacc에서 쓰일 토큰값으로 리턴한다.

Yacc에서는 문법과 토큰을 정의한다. 처음엔 start와 end를 인식한다.

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

stmt_list에서는 리스트 트리를 만든다.

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

stmt에 핵심 문법을 작성한다.

【PLAY】

PLAY는 Yacc에서 P토큰으로 간주되며, REPEND는 ---로 간주된다.

```
stmt    :      P expr_list REPEND              { $$=MakeOPTree(P, $2, NULL); }
```

P토큰과 REPEND 사이에 오는 값들을 바탕으로 OP트리를 형성한다.
expr_list에는 노드와 옥타브 정보가 올 수 있는데, 리스트 트리를 형성하여 처리한다.

```
expr_list:     expr_list expr   {$$=MakeListTree($1, $2);}
       |       expr             {$$=MakeListTree(NULL, $1);}

expr   :       PLAY term term    { $$=MakeOPTree(PLAY, $2, $3); }
       |       OCT term          { $$=MakeNode(OCT, $2->tokenval);}

term   :       ID               { /* ID node is created in lex */ }
       |       NUM              { /* NUM node is created in lex */ }
       ;
```

위에서 옥타브의 경우 OCT 뒤에 오는 숫자를 tokenval로 설정하는데, 파싱할 때 옥타브 값으로 저장된다.

```
void prtcode(int token, int val)
{
        switch (token) {
        case OCT:
                octave = val;
                break;
        };
}
```

음악을 재생할 때 옥타브만큼 주파수에 2를 곱한다. 옥타브가 커질수록 곱하는 횟수가 증가하여 높은 음이 재생된다.

```
void prtcode(int token, int val)
{
        switch (token) {
        case PLAY:
                for (i=0;i<octave;i++){
                        playNode *= 2;
                }
                break;
        };
}
```

음정에 따른 주파수는 아래와 같으며, 가장 낮은 음정에 2를 계속 곱하여 높은 음을 생성할 수 있다.

(단위 : Hz)

옥타브 / 음계	1	2	3	4	5	6	7	8
C(도)	32.7032	65.4064	130.8128	261.6256	523.2511	1046.502	2093.005	4186.009
C#	34.6478	69.2957	138.5913	277.1826	554.3653	1108.731	2217.461	4434.922
D(레)	36.7081	73.4162	146.8324	293.6648	587.3295	1174.659	2349.318	4698.636
D#	38.8909	77.7817	155.5635	311.1270	622.2540	1244.508	2489.016	4978.032
E(미)	41.2034	82.4069	164.8138	329.6276	659.2551	1318.510	2637.020	5274.041
F(파)	43.6535	87.3071	174.6141	349.2282	698.4565	1396.913	2793.826	5587.652
F#	46.2493	92.4986	184.9972	369.9944	739.9888	1479.978	2959.955	5919.911
G(솔)	48.9994	97.9989	195.9977	391.9954	783.9909	1567.982	3135.963	6271.927
G#	51.9130	103.8262	207.6523	415.3047	830.6094	1661.219	3322.438	6644.875
A(라)	55.0000	110.0000	220.0000	440.0000	880.0000	1760.000	3520.000	7040.000
A#	58.2705	116.5409	233.0819	466.1638	932.3275	1864.655	3729.310	7458.620
B(시)	61.7354	123.4708	246.9417	493.8833	987.7666	1975.533	3951.066	7902.133

playNode에는 연주하게 될 주파수 정보가 저장되어 있는데, 노드에 따른 기본 주파수는 상단에 전역변수로 선언되어 있다.

```
float C_node = 16.4;
float Cs_node = 17.3;
float D_node = 18.4;
float Ds_node = 19.4;
float E_node = 20.6;
float F_node = 21.8;
float G_node = 24.4;
...(이하 생략)
```

노드는 ID토큰으로 인식된다. JudgeNode함수를 통해 연주할 노드 정보를 갱신한다.

```
void prtcode(int token, int val)
{
	switch (token) {
	case ID:
		JudgeNode(val);
	};
}

...

void JudgeNode(int val){
	if(strcmp(symtbl[val],"C")==0)	playNode = C_node;
	else if(strcmp(symtbl[val],"Cs")==0) playNode = Cs_node;
	else if(strcmp(symtbl[val],"D")==0) playNode = D_node;
	else if(strcmp(symtbl[val],"Ds")==0) playNode = Ds_node;
	else if(strcmp(symtbl[val],"E")==0) playNode = E_node;
	else if(strcmp(symtbl[val],"F")==0) playNode = F_node;
	else if(strcmp(symtbl[val],"Fs")==0) playNode = Fs_node;
	else if(strcmp(symtbl[val],"G")==0) playNode = G_node;
	else if(strcmp(symtbl[val],"Gs")==0) playNode = Gs_node;
	else if(strcmp(symtbl[val],"A")==0) playNode = A_node;
	else if(strcmp(symtbl[val],"As")==0) playNode = As_node;
	else if(strcmp(symtbl[val],"B")==0) playNode = B_node;
	else if(strcmp(symtbl[val],"R")==0) playNode = 0.0;
}
```

【SPEED】

SPEED 토큰을 인식하면 뒤에 오는 숫자를 tokenval값으로 지정하여 노드를 만든다.

```
stmt    : SPEED term STMTEND  { $$=MakeNode(SPEED, $2->tokenval);}
```

인식된 NUM 토큰은 length 변수에 저장된다. 기본 speed는 250이다.

```
int speed = 250; //전역변수
...
void prtcode(int token, int val)
{
        switch (token) {
        case NUM:
              length = val*speed;
              break;
        };
}
```

【Beep】

설정된 주파수와 길이를 바탕으로 C언어의 Beep 함수를 호출하여 비프음을 출력한다.

```
void prtcode(int token, int val)
{
        switch (token) {
        case PLAY:
                Beep(playNode,length);
                break;
}
```

【REP】

반복 기능은 REP토큰을 인식하여 똑같은 트리를 brother에 하나 더 만들어 구현했다.

```
|  REP expr_list REPEND {$$=MakeOPTree(REP, $2, NULL);
 $$->brother=MakeOPTree(REP, $2, NULL);} //brother에 같은 트리 추가
```

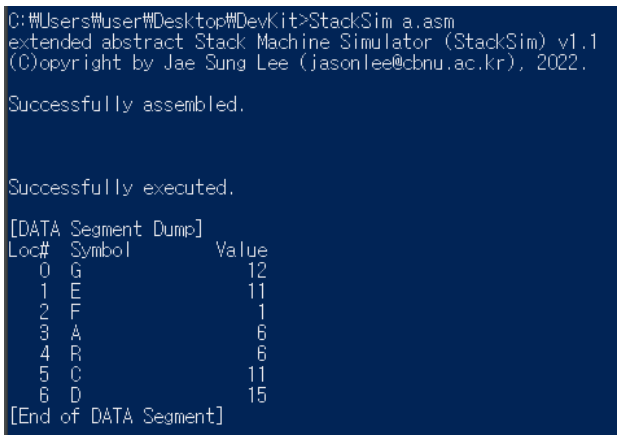
4. 실행결과

오디오 프로그램이라 실행 후 소리를 직접 들어보지 않으면 결과를 알 수 없다.
보고서에는 만든 샘플 몇 개와 생성되는 스택 화면을 작성하고, 자세한 실행결과는 프로젝트 데모 영상에 녹화했다.

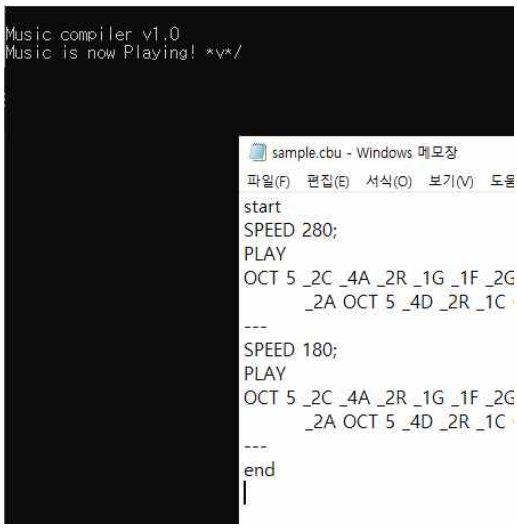
> 직접 제작한 샘플악보

고향의 봄을 악보로 제작해서 구동시켜보았다. 이 곡은 처음부터 끝까지 반복없이 일정한 속도로 재생한다.

a.sim을 StackSim으로 구동시키면 해당 곡에 쓰인 음표의 종류와 빈도를 출력한다.

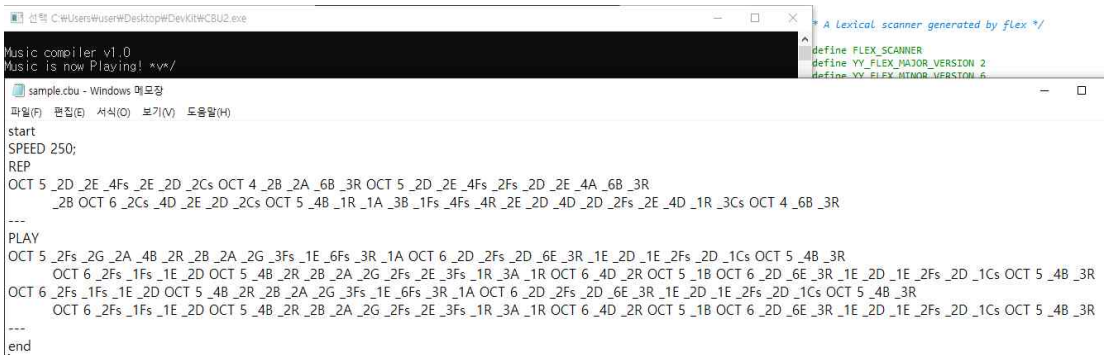


응원가를 악보로 제작해서 구동시켜보았다. 중간에 속도 변주를 했는데, 속도가 빨라졌다.



> 속도가 280에서 180으로 변경되어 빨라진다.

가로수 그늘 아래 서면 악보를 제작해서 구동시켜보았다. 이 곡은 첫 부분에 반복이 있고 반복이 끝난 후에 후렴구로 넘어간다.



※ music 폴더에 있는 악보를 sample.cbu에 붙여넣은 뒤 컴파일 후 실행하면 재생할 수 있습니다.