CS3480 Project

In this directory you will find the files needed to use the language. In this report you will find the explanation of the language and its features. The language is a Domain Specific Language aimed at music creation, using the java midi as its backbone.

List of Domain Specific Features:

Programs:

It is one or more statements

Statements are separated by a '`;`', there is not statement-terminator.

A sequence of whitespace characters can be used wherever one whitespace is valid

Arithmetic and expressions:

Operations available are subtraction, addition, multiplication, division, modular and to the power of such as `x-1`, `x+1`, `x*y`, `x/4`, `x%2`, `x**3`.

Expressions, such as strings, doubles are available, these are needed to allow the user to input strings for the musical notes and doubles for adjusting beat ratios.

Integers are available, these are used for changing instrument, repeating notes.

Predicates

Boolean expressions available are greater than, less than, equal to, not, less than or equal to and greater than or equal to, such as:

`x > y`; x greater than y

`x < y`; x less than y

`x == y`; x equal to y

`x != y`; x not y

`x <= y`; x less than or equal to y

`x >= y`; x greater than or equal to y

Selection statements

```
if pred then statement else statement
```

```
if pred then statement
```

Where `pred` is a predicate as defined above and statement is any statement.

I did not add anymore selection statements as for the domain I was creating the language for I did not see any reason for anymore.

Iteration statement

```
while pred do statement
```

Where `pred` is a predicate as defined above, and statement is any statement.

Backend statements

This provides several DSL-type statements:

`play a`; where a is a string expression or ID

`melody x { String }` where x is the ID to the melody and string is a string expression of notes

`instrument x` xor `y` where x is an integer, between 0 and 127, or a string and y is an integer, between 0 and 16.

`volume x` where x is an integer

`mix(x,y)` where x and y are melodies.

`scale(x,y)` where x and y are strings, x is a note and y is a scale type.

`arpeggio(x,y)` where x and y are strings, x is a note and y is a scale type.

`print(x)` or `print(x,y)` where x and y are an integer, ID, or string.

`beatRatio x` where x is a double, between 0 and 1.

`bpm x` where x is an integer for the bpm to be set to

`octave x` where x is an integer between 0 and 10.

`repeateNote(x,y)` where x is a string and y is an integer, x is a note and y is how many times the user wants the note to be repeated instead of writing out a melody of the same note.

Internal syntax constructors and arities:

- seq 2 sequence _1 then _2
- sub 2 compute _1 − _2
- gt 2 compute _1 > _2
- ne 2 compute _1 != _2
- assign 2 bind _2 to _1 in variables map
- deref 1 retrieve binding for _1 in variables map
- if 3 if _1 then _2 else _3
- while 2 while _1 do _2
- lt 2 compute _1 < _2
- eq 2 compute _1 == _2
- lteq 2 compute _1 <= _2
- gteq 2 compute _1 >= _2
- add 2 compute _1 + _2
- div 2 compute _1 / _2
- mul 2 compute _1 * _2
- mod 2 compute _1 % _2
- pow 2 compute _1 ** _2
- repeateNote 2 while _2 > 0 do _1
- bpm 1 bind _1 to bpm in MiniMusicPlayer

- instrument 1 bind _1 to instrument in MiniMusicPlayer
- instrument 2 bind _1 to instrument in MiniMusicPlayer for channel _2
- volume 1 bind _1 to volume in MiniMusicPlayer
- beatRatio 1 bind _1 to beatRatio in MiniMusicPlayer

eSOS rules

Source Form

-sequenceDone

---

seq(__done, _C), _sig -> _C, _sig

-sequence

_C1, _sig -> _C1P, _sigP

---

seq(_C1, _C2), _sig -> seq(_C1P, _C2), _sigP

-ifTrue

---

if(True, _C1, _C2),_sig -> _C1, _sig

-ifFalse

---

if(False, _C1, _C2),_sig -> _C2,_sig

-ifResolve

_E, _sig ->_EP, _sigP

---

if(_E,_C1,_C2),_sig -> if(_EP, _C1, _C2), _sigP

-while

---

while(_E, _C),_sig -> if(_E, seq(_C, while(_E,_C)), __done), _sig

-assign

_n |> __int32(_)

---

assign(_X, _n), _sig -> __done, __put(_sig, _X, _n)

-assignResolve

_E, _sig -> _I, _sigP

---

assign(_X,_E), _sig -> assign(_X, _I), _sigP

-gt

_n1 |> __int32(_) _n2 |> __int32(_)

---

gt(_n1, _n2),_sig -> __gt(_n1, _n2),_sig

-gtRight

_n |> __int32(_) _E2, _sig -> _I2,_sigP

---

gt(_n, _E2),_sig -> gt(_n, _I2), _sigP

-gtLeft

_E1, _sig -> _I1, _sigP

---

gt(_E1, _E2),_sig -> gt(_I1, _E2), _sigP

-ne

_n1 |> __int32(_) _n2 |> __int32(_)

---

ne(_n1, _n2),_sig -> __ne(_n1, _n2),_sig

-neRight

_n |> __int32(_) _E2, _sig -> _I2,_sigP

---

ne(_n, _E2),_sig -> ne(_n, _I2), _sigP

-neLeft

_E1, _sig -> _I1, _sigP

---

ne(_E1, _E2),_sig -> ne(_I1, _E2), _sigP

-sub

_n1 |> __int32(_) _n2 |> __int32(_)

---

sub(_n1, _n2),_sig -> __sub(_n1, _n2),_sig

-subRight

_n |> __int32(_) _E2,_sig -> _I2,_sigP

---

sub(_n, _E2),_sig -> sub(_n, _I2), _sigP

-subLeft

_E1,_sig -> _I1,_sigP

---

sub(_E1, _E2),_sig -> sub(_I1, _E2), _sigP

-variable

__get(_sig, _R) |> _Z

---

deref(_R),_sig -> _Z, _sig

-add

_n1 |> __int32(_) _n2 |> __int32(_)

---

add(_n1, _n2),_sig -> __add(_n1, _n2),_sig

-addRight

_n |> __int32(_) _E2,_sig -> _I2,_sigP

---

add(_n, _E2),_sig -> add(_n, _I2), _sigP

-addLeft

_E1,_sig -> _I1,_sigP

---

add(_E1, _E2),_sig -> add(_I1, _E2), _sigP

-mul

_n1 |> __int32(_) _n2 |> __int32(_)

---

mul(_n1, _n2),_sig -> __mul(_n1, _n2),_sig

-mulRight

_n |> __int32(_) _E2,_sig -> _I2,_sigP

---

nul(_n, _E2),_sig -> mul(_n, _I2), _sigP

-mulLeft

_E1,_sig -> _I1,_sigP

---

mul(_E1, _E2),_sig -> mul(_I1, _E2), _sigP

-div

_n1 |> __int32(_) _n2 |> __int32(_)

---

div(_n1, _n2),_sig -> __div(_n1, _n2),_sig

-divRight

_n |> __int32(_) _E2,_sig -> _I2,_sigP

---

div(_n, _E2),_sig -> div(_n, _I2), _sigP

-divLeft

_E1,_sig -> _I1,_sigP

---

div(_E1, _E2),_sig -> div(_I1, _E2), _sigP

-mod

_n1 |> __int32(_) _n2 |> __int32(_)

---

mod(_n1, _n2),_sig -> __mod(_n1, _n2),_sig

-modRight

_n |> __int32(_) _E2,_sig -> _I2,_sigP

---

mod(_n, _E2),_sig -> mod(_n, _I2), _sigP

-modLeft

_E1,_sig -> _I1,_sigP

---

mod(_E1, _E2),_sig -> mod(_I1, _E2), _sigP

-lt

_n1 |> __int32(_) _n2 |> __int32(_)

---

lt(_n1, _n2),_sig -> __lt(_n1, _n2),_sig

-ltRight

_n |> __int32(_) _E2, _sig -> _I2,_sigP

---

lt(_n, _E2),_sig -> lt(_n, _I2), _sigP

-ltLeft

_E1, _sig -> _I1, _sigP

---

lt(_E1, _E2),_sig -> lt(_I1, _E2), _sigP

-eq

_n1 |> __int32(_) _n2 |> __int32(_)

---

eq(_n1, _n2),_sig -> __eq(_n1, _n2),_sig

-eqRight

_n |> __int32(_) _E2, _sig -> _I2,_sigP

---

eq(_n, _E2),_sig -> eq(_n, _I2), _sigP

-eqLeft

_E1, _sig -> _I1, _sigP

---

eq(_E1, _E2),_sig -> eq(_I1, _E2), _sigP

-ge

_n1 |> __int32(_) _n2 |> __int32(_)

---

ge(_n1, _n2),_sig -> __ge(_n1, _n2),_sig

-geRight

_n |> __int32(_) _E2, _sig -> _I2,_sigP

---

ge(_n, _E2),_sig -> ge(_n, _I2), _sigP

-geLeft

_E1, _sig -> _I1, _sigP

---

ge(_E1, _E2),_sig -> ge(_I1, _E2), _sigP

-le

_n1 |> __int32(_) _n2 |> __int32(_)

---

le(_n1, _n2),_sig -> __le(_n1, _n2),_sig

-leRight

_n |> __int32(_) _E2, _sig -> _I2,_sigP

---

le(_n, _E2),_sig -> le(_n, _I2), _sigP

-leLeft

_E1, _sig -> _I1, _sigP

---

le(_E1, _E2),_sig -> le(_I1, _E2), _sigP

-play

_opCode |> __int32(_) _n1 |> __string(_)

---

play(_opCode, _n1),_sig -> __user(_opCode, _n1)

-repeatNote

_opCode |> __int32(_) _n1 |> __string(_) _n2 |> __int32(_)

---

repeatNote(_opCode, _n1, _n2),_sig -> __user(_opCode, _n1, _n2)

-setBpm

_opCode |> __int32(_) _n1 |> __int32(_)

---

setBpm(_opCode, _n1),_sig -> __user(_opCode, _n1)

-setInstrument

_opCode |> __int32(_) _n1 |> __int32(_)

---

setInstrument(_opCode, _n1),_sig -> __user(_opCode, _n1)

-setInstrument

_opCode |> __int32(_) _n1 |> __string(_)

---

setInstrument(_opCode, _n1),_sig -> __user(_opCode, _n1)

-mix

_opCode |> __int32(_) _n1 |> __array(_) _n2 |> __array(_)

---

mix(_opCode, _n1, _n2),_sig -> __user(_opCode, _n1, _n2)

-setVolume

_opCode |> __int32(_) _n1 |> __int32(_)

---

setVolume(_opCode, _n1),_sig -> __user(_opCode, _n1)

-scale

_opCode |> __int32(_) _n1 |> __string(_) _n2 |> __string(_)

---

scale(_opCode, _n1, _n2),_sig -> __user(_opCode, _n1, _n2)

-arpeggio

_opCode |> __int32(_) _n1 |> __string(_) _n2 |> __string(_)

---

arpeggio(_opCode, _n1, _n2),_sig -> __user(_opCode, _n1, _n2)

-octave

_opCode |> __int32(_) _n1 |> __int32(_)

---

octave(_opCode, _n1),_sig -> __user(_opCode, _n1)

-beatRatio

_opCode |> __int32(_) _n1 |> __real64(_)

---

beatRatio(_opCode, _n1),_sig -> __user(_opCode, _n1)

Typeset Form

Please see eSOS_rules_typeset.pdf

Internal to External Syntax Translator

statement ::= seq^^ | assign^^ | if^^ | while^^ | backend^^ | repeatNote^^ | bpm^^ | instrument^^ | volume^^ | beatRatio^^ | octave^^

seq ::= statement statement

assign ::= ID ':='^ subExpr ';'^

if ::= 'if'^ relExpr statement 'else'^ statement

while ::= 'while'^ relExpr statement

repeatNote ::= 'repeatNote'^ '('^ subExpr ','^ subExpr ')'^

bpm ::= 'bpm'^ INTEGER

instrument ::= 'instrument'^ INTEGER | STRING_DQ

beatRatio ::= 'beatRatio'^ REAL

volume ::= 'volume'^ INTEGER

octave ::= 'octave'^ INTEGER

relExpr ::= subExpr^^ | gt^^ | ne^^ | lt^^ | eq^^ | lteq^^ | gteq^^

gt ::= relExpr '>'^ subExpr

ne ::= relExpr '!='^ subExpr

eq ::= relExpr '=='^ subExpr

lt ::= relExpr '<'^ subExpr

le ::= relExpr '<='^ subExpr

ge ::= relExpr '>='^ subExpr

subExpr ::= operand^^ | sub^^ | add^^ | mul^^ | div^^ | mod^^

sub ::= subExpr '-'^ operand

add ::= subExpr '+'^ operand

mul ::= subExpr '*'^ operand

div ::= subExpr '/'^ operand

mod ::= subExpr '%'^ operand

operand ::= deref^^ | INTEGER^^ | '('^ subExpr^^ ')'^

deref ::= ID

Running Instructions

Use the command:

```
javac -cp ../art.jar *.java
```

to compile the java files.

Then to parse the eSOS rules use the command:

```
esos eSOSRules.art
```

To parse the Attribute Grammar:

```
parse AttributeGrammar
```

Examples and Tests

This test will show a majority of the functionality of the language, including all of the DSL functionality.

```
a := 1 + 1;

b := a + 1;

print(a);

print(b);

print("hello");

print("hello","hi");

print(13);

bpm 180;

octave 5;

beatRatio 0.9;

instrument(114,1);

melody t {"C . C B C . C B C . D D . F F . E C D . B G E C D . G C F . E E . D D . C B"}

play t;

instrument 10;

volume 20;

repeatNote("A B", 3);

melody c {"A A A A A A A A"}

melody d {"A B C D E F G F"}

mix(c, d);
```

```
volume 80;

instrument ("Acoustic Grand Piano",0);

play t;

instrument "Banjo";

play t;

scale ("C","MAJOR");

arpeggio ("C", "MAJOR");
```

Output after test:

```
[Ljavax.sound.midi.MidiDevice$Info;@4bf558aaAttached to ValueUserPlugin : Chris' Value User Plugin

[Ljavax.sound.midi.MidiDevice$Info;@78aab498** Accept

2

3

hello

hello

hi

13

Variables at end of program: {a=2, b=3}
```