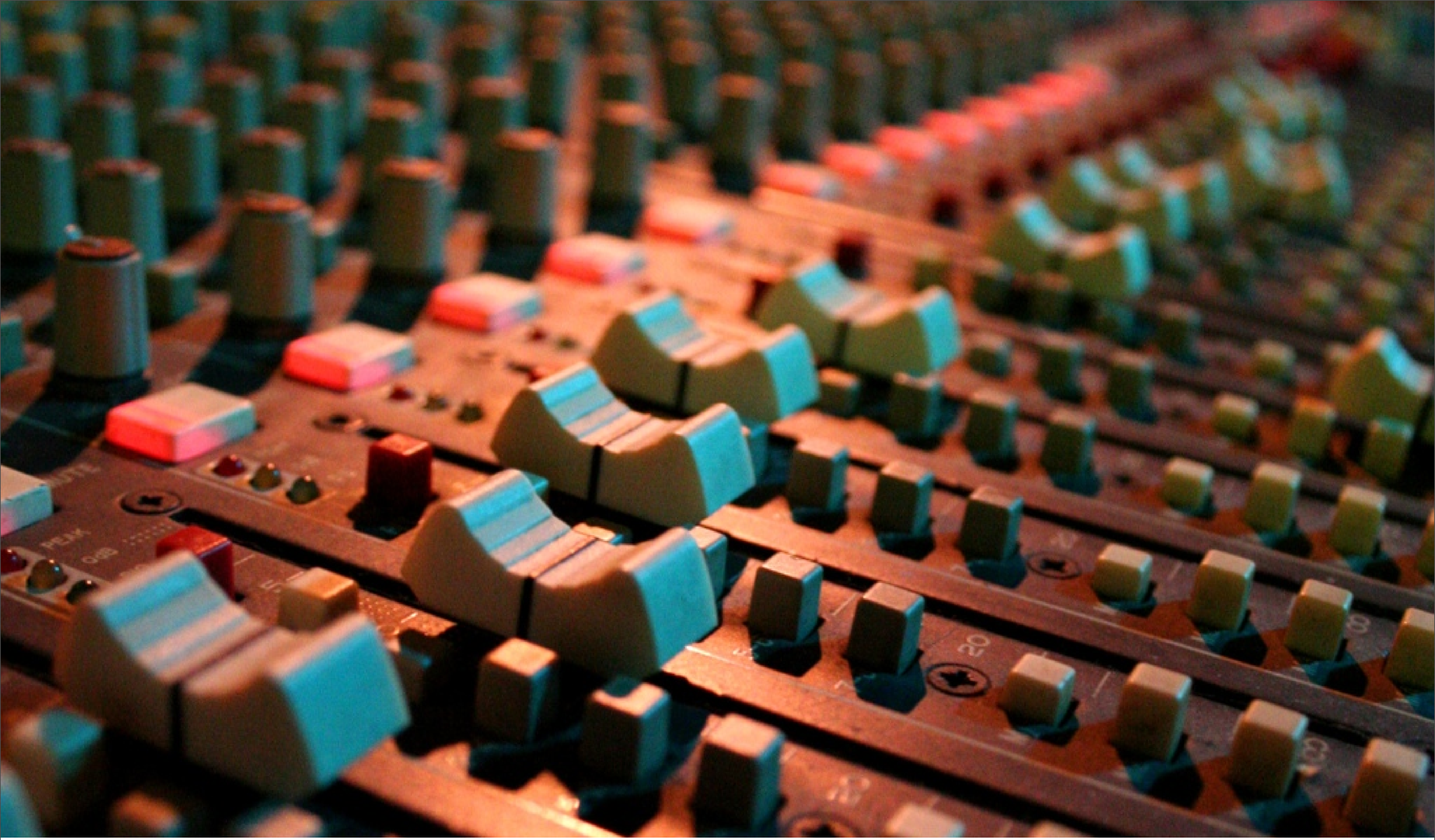


Analog Additive Synthesis Language

Vaishnav Janardhan, Rob Katz,
Carlos René Pérez, Albert Tsai

Motivation

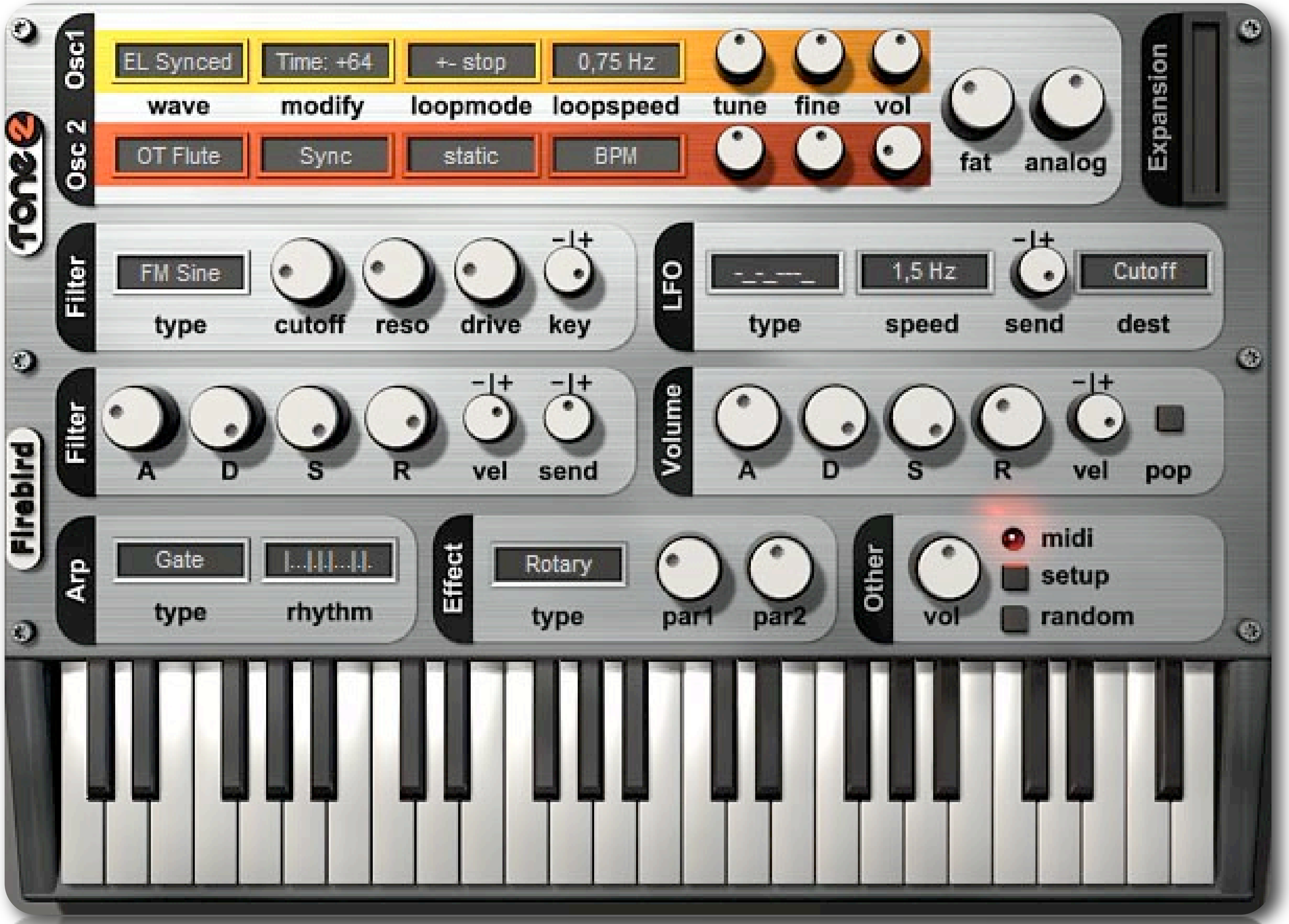
Sound Synthesis

Generating or manipulating electronic signals/audio tones for music creation.

Motivation

Sound Synthesis

Generating or manipulating electronic signals/audio tones for music creation.



Motivation

Sound Synthesis

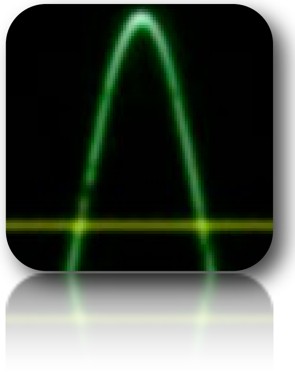
Generating or manipulating electronic signals/audio tones for music creation.

Motivation

Sound Synthesis

Generating or manipulating electronic signals/audio tones for music creation.

Motivation

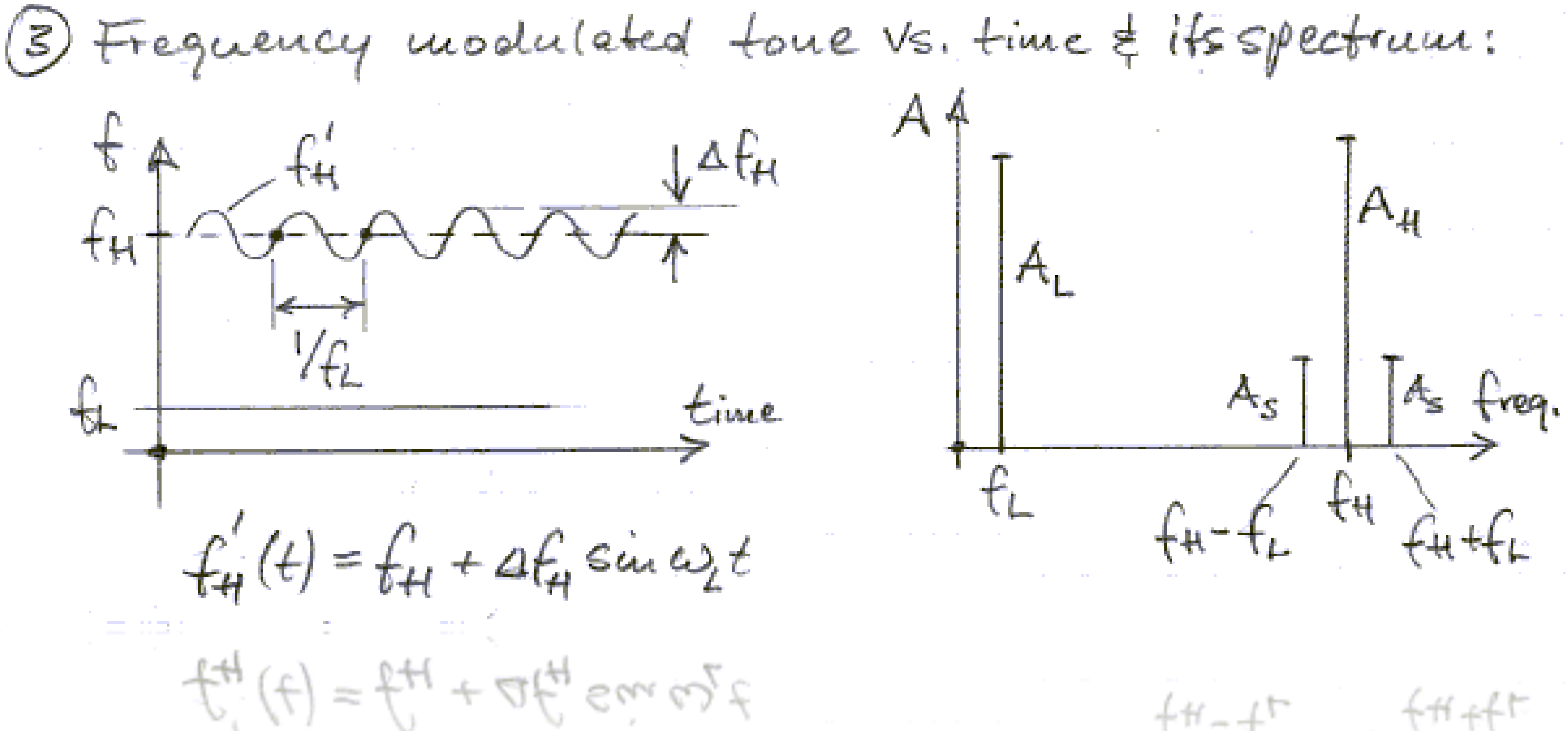


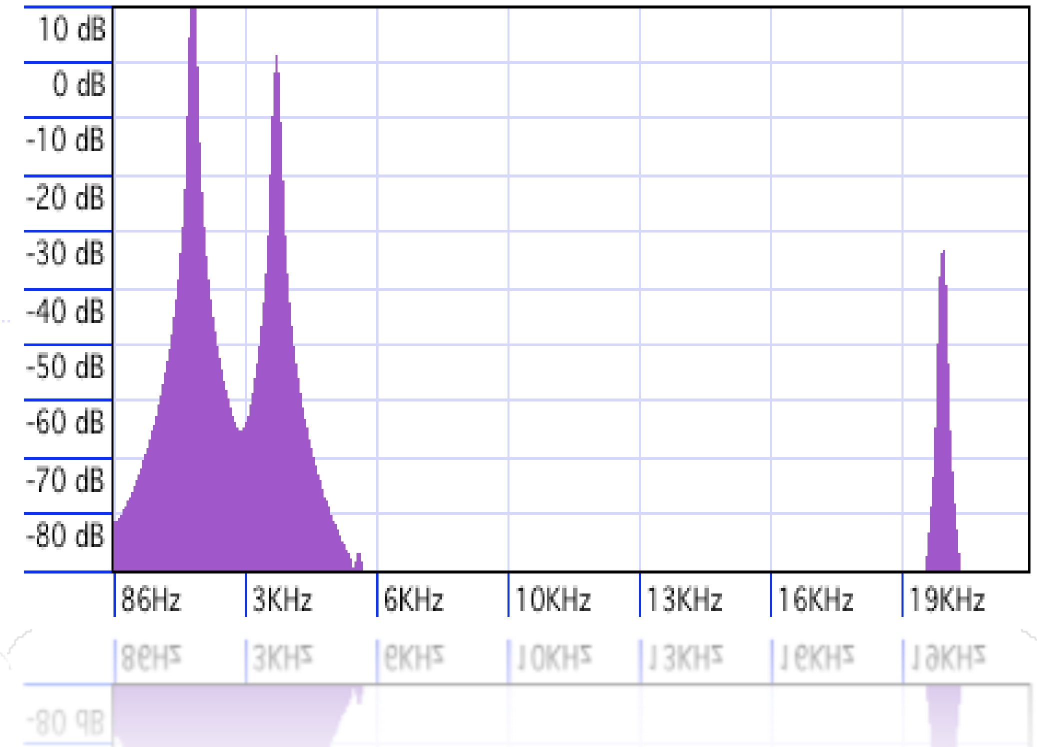
Sound Synthesis

Generating or manipulating electronic signals/audio tones for music creation.

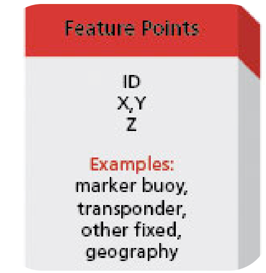
Educational Tool

Study the effects of envelopes, oscillators, mixers on electronic signals.

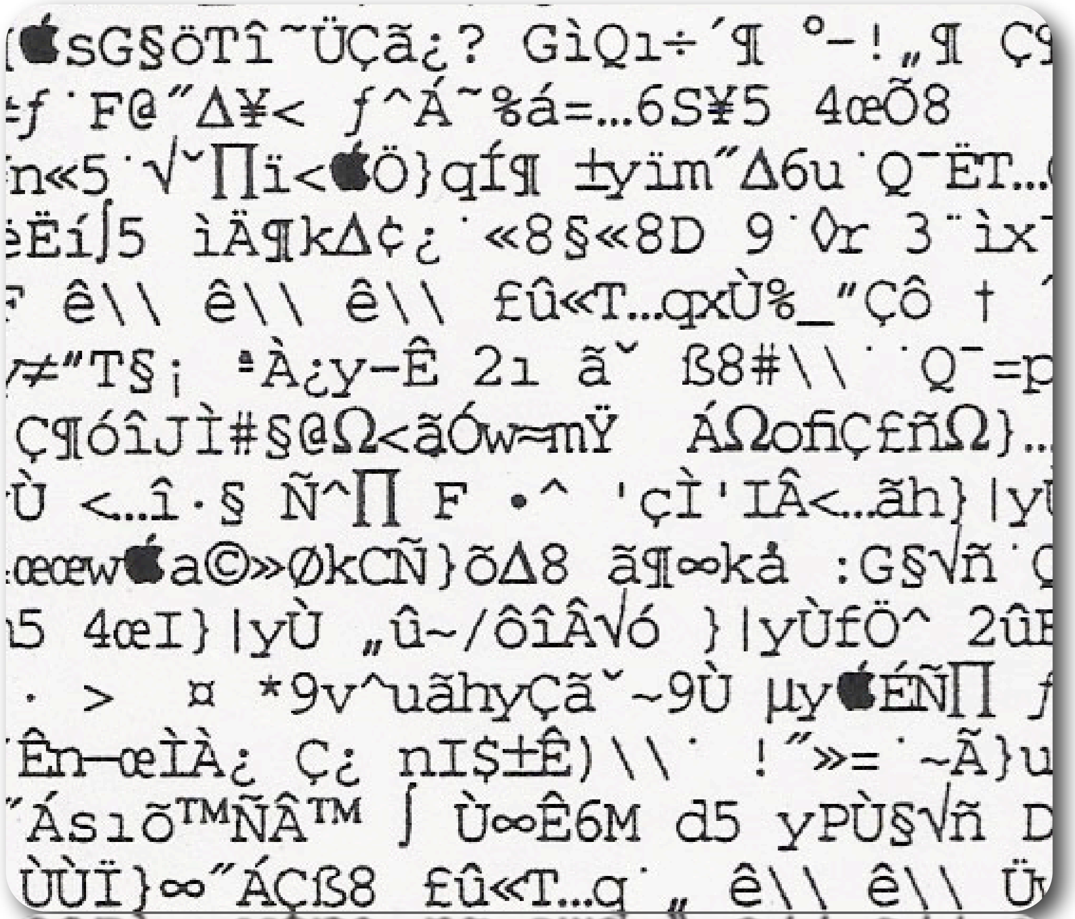




Language Introduction

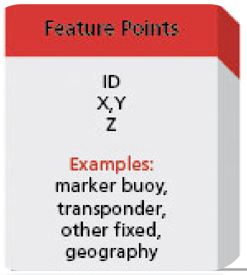


Fundamental Data Types



Program Structure

Language Features



Feature Points

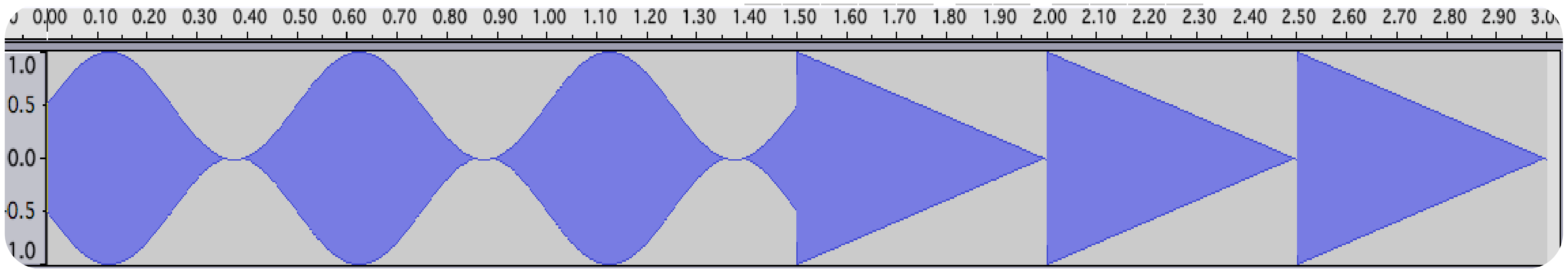
ID
X,Y
Z

Examples:
marker buoy,
transponder,
other fixed,
geography

Fundamental Data Types

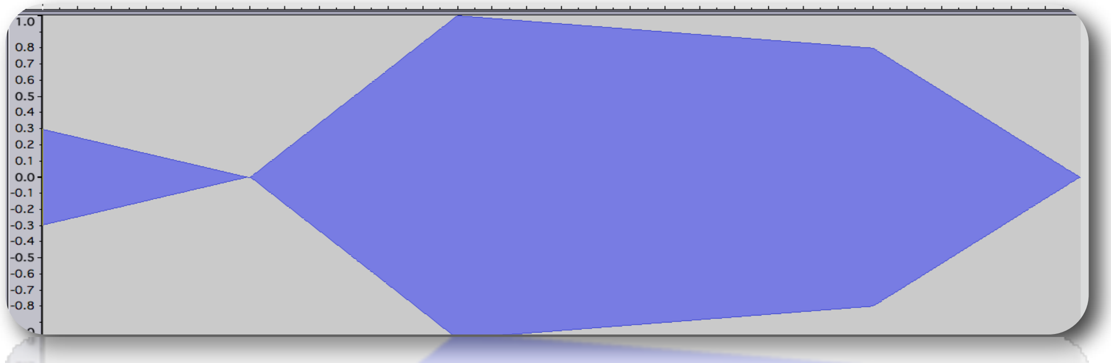
Oscillators

Produces a repetitive electronic signal: sine, saw, revsaw, and square waves.



Envelopes

Produces a repetitive electronic signal: sine, saw, revsaw, and square waves.



Feature Points

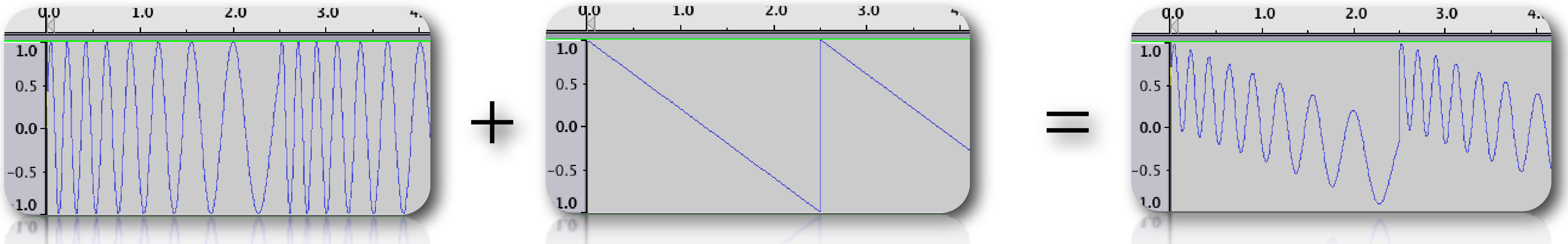
ID
X,Y
Z

Examples:
marker buoy,
transponder,
other fixed,
geography

Fundamental Data Types

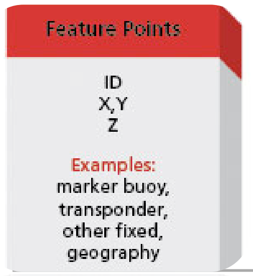
Mixers

Sums up its inputs, from multiple oscillators and generates a new output.



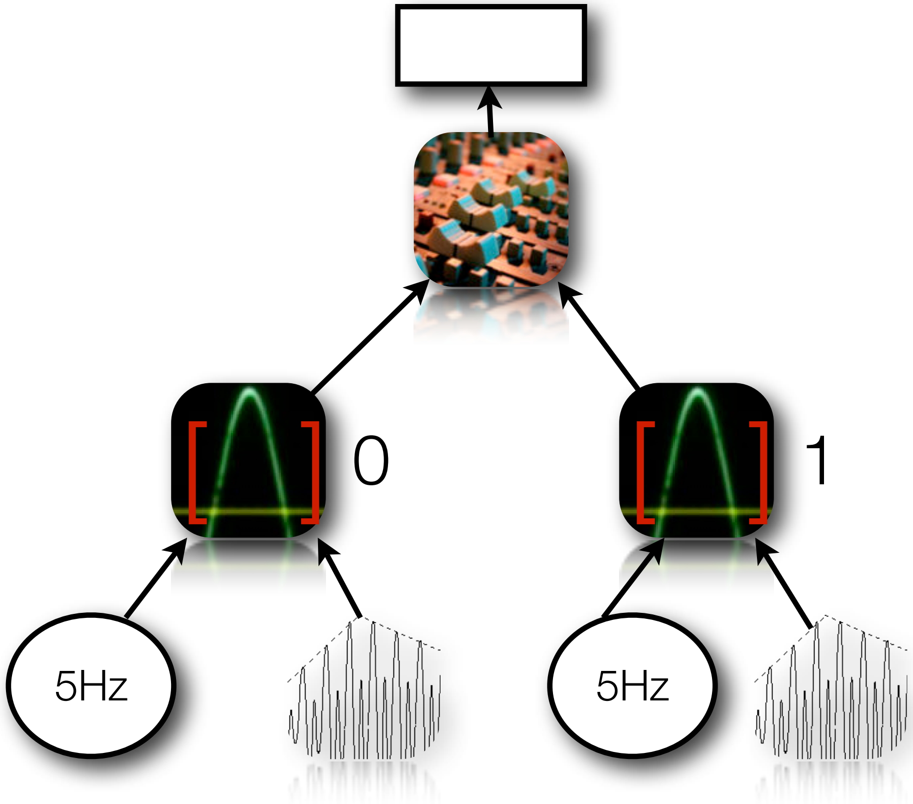
Oscillators Banks

Array of oscillators that share the same name and are indexed.



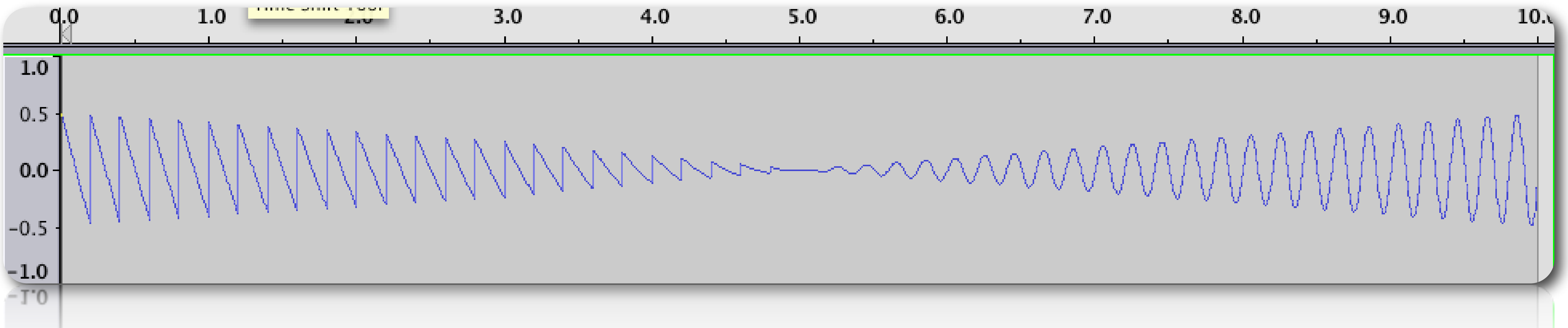
Data Types Relations

- Oscillators
- Envelopes
- Mixers
- Oscillators Banks



```
mixer output;
oscbank ob = 2;
ob[0] = “SINE”;
ob[1] = “SAW”;
env e1 = { (0.0,0.0) (0.5,0.0) (1.0,1.0) };
env e2 = { (0.0,1.0) (0.3,0.5) (0.5,0.0) (1.0,0.0) };

ob[0](5,e1);
ob[1](5,e2);
output = ob[0] + ob[1]
```



Program Structure

- Header Section

  Assign the name of the output file, length and optional assignment of the segments.

```
start header
    OUTPUT = “test”
    TONELENGTH = 10
    SEGMENTS = 5
end header
```

- User Defined Functions Section

  Similar to Main Function, always returns a mixer.

```
start func of(int x)
    def...
    con..
    OUTPUT = ..
end func of
```

- Main Function

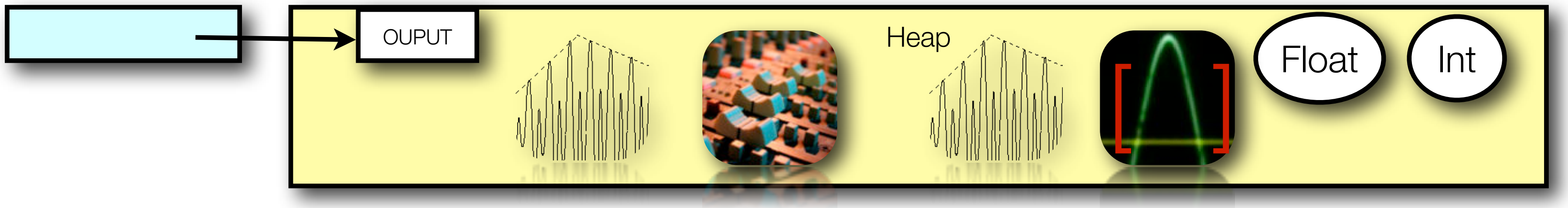
  Must be defined, follows the form of all functions except that it has the “main” identifier.

```
start func main
    def...
    con..
    OUTPUT = ..
end func main
```

Program Structure

Definition Section

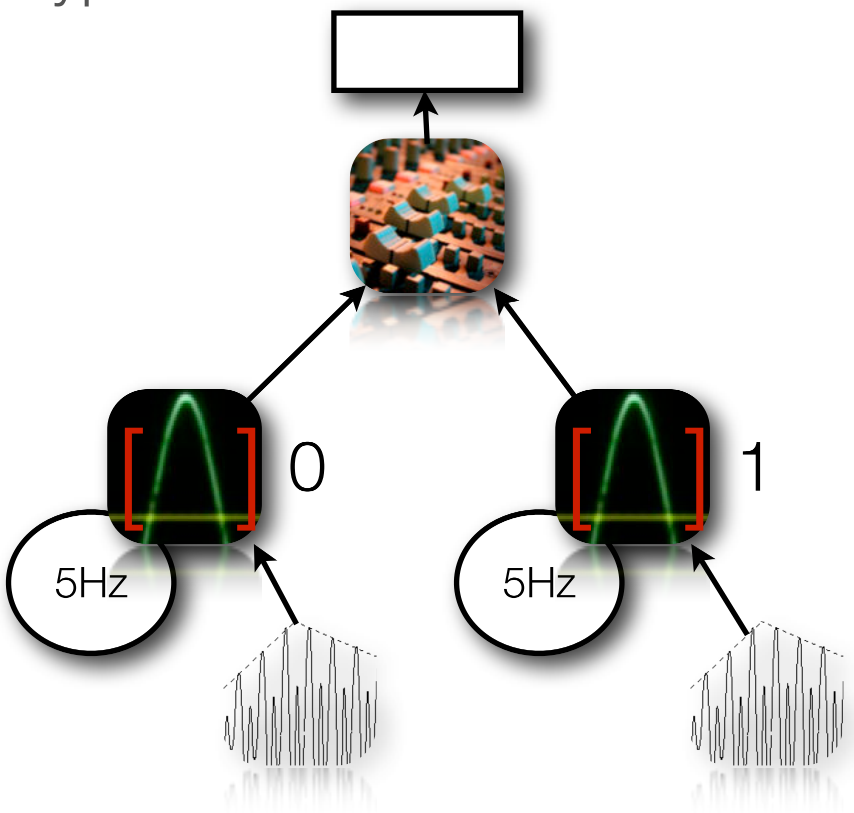
All identifiers must be declared or defined before there use.



```
start func main
    start def
        osc o1, o2;
        o1="SINE";
        o2="SAW";
        mixer s1;
    end def
    ...
end func main
```

Connection Section

Tree Walker associates data types with other data types.



OUTPUT Equation

The most general mixer, always defined.
Also the output of user defined functions.

Language Features

- Function Overloading

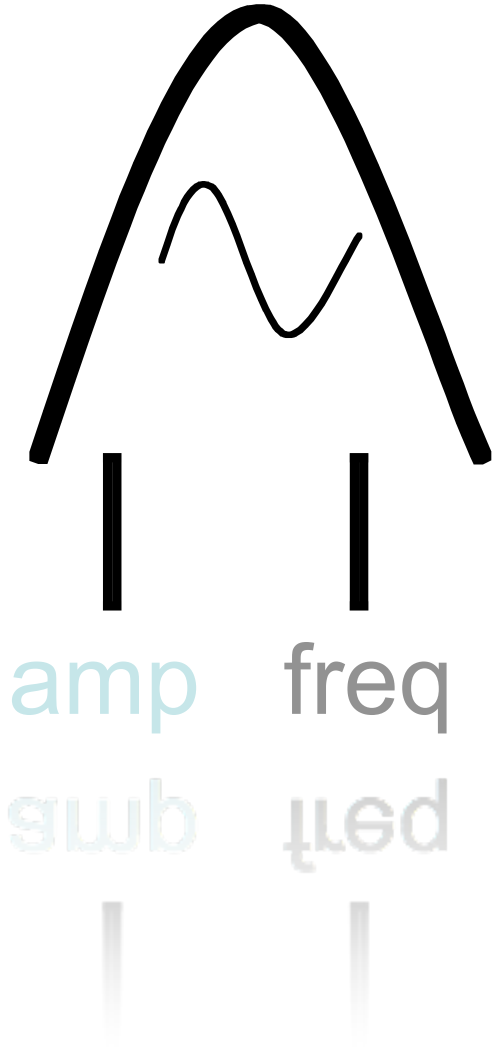
  Functions will be matched according to name and argument types.

- Dynamic Scoping

  Variables in the symbol table

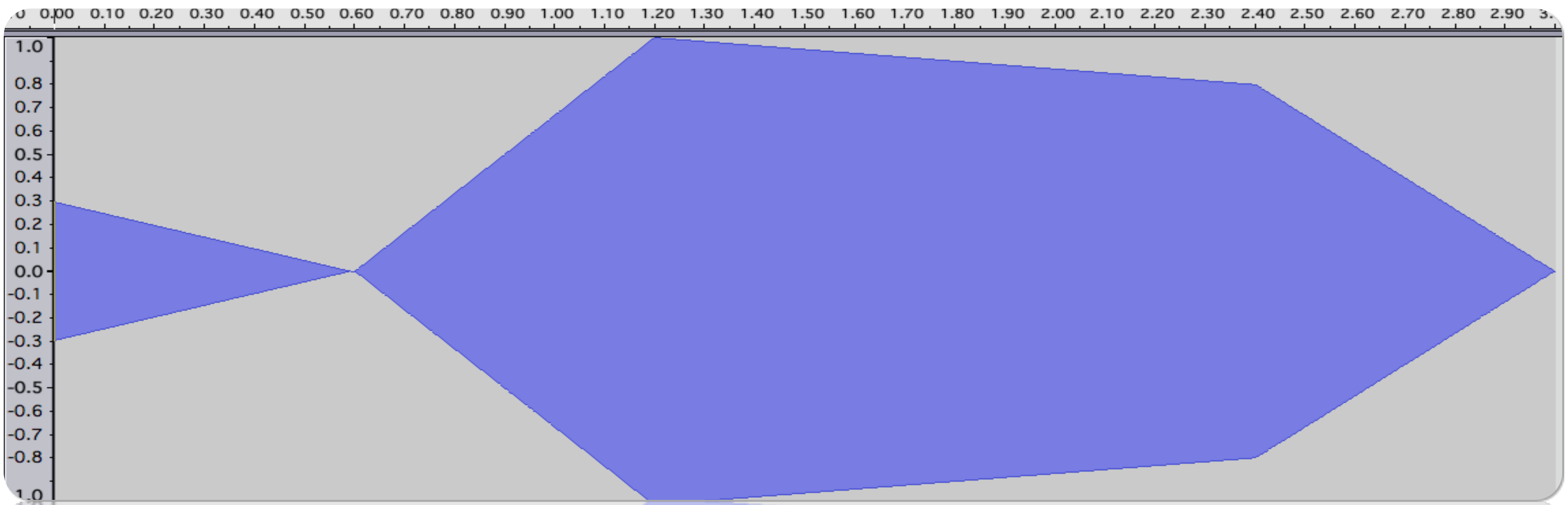
In Depth: Oscillators and Oscillator Banks

- Oscillators are the basic tone generators
- Each tone has a frequency and amplitude input associated to it.
- Frequency and amplitude can be constant or controlled by another synthesis element.
- Oscillator banks are array of oscillators, and are stored as individual elements.



Input to Oscillators
Int/Float
Oscillators
Envelopes

In Depth: Envelopes



- Envelope is composed of (time,value) pairs.

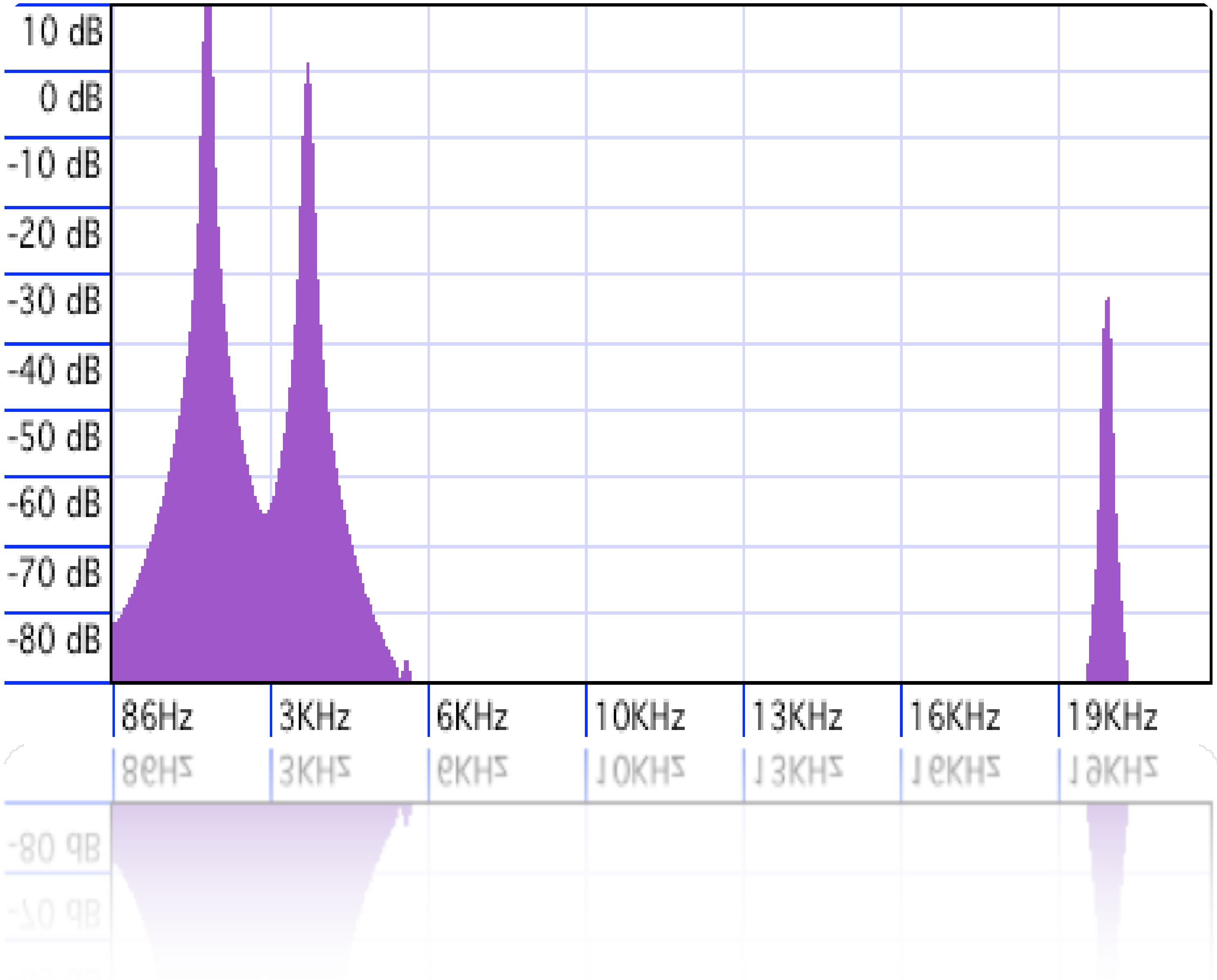
```
env e1 = { (0.0,0.3)
               (0.2,0.0) (0.4,1.0)
               (0.8,0.8) (1.0,0.0) };
...
oscb1[1](e1@440,e1);
```

- It is controlling the amplitude of an oscillator, in the example provided

- If frequency is not constant, a central frequency will be set using '@' symbol.

In Depth: Mixer and Oscillator Connection

Mixer: Mixer takes oscillators or functions as inputs and adds them according to given proposition.



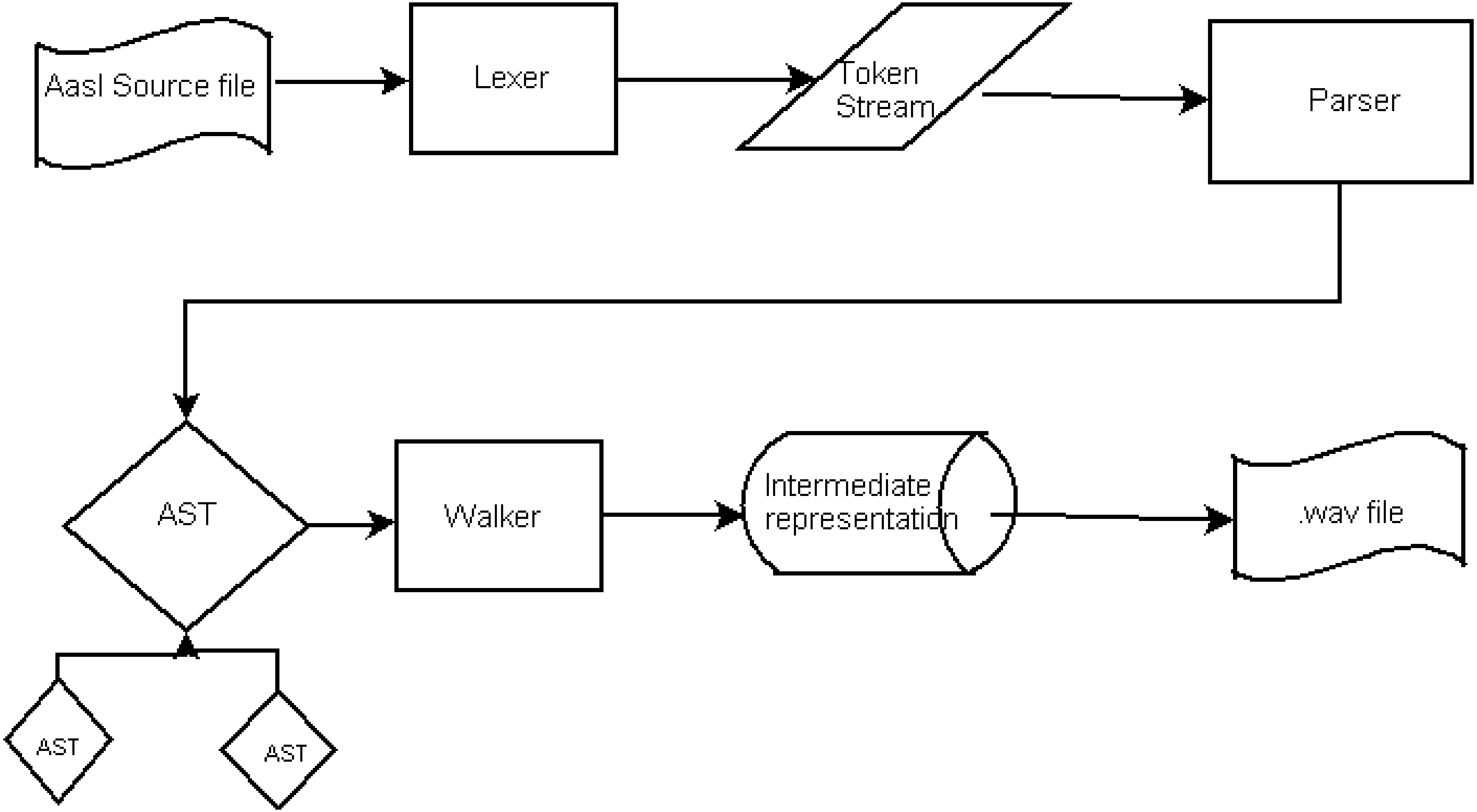
```
in the function osc3(int x) …
        o3(x,1.0);
in main
        x=2000
        mixer m1;

        start connect
        o1(x,1.0);
        o2(2*x,1.0);
        end connect

m1 = 200*o1 + 50*o2 + osc3{20000};
```

All oscillators, with increasing frequency (2kHz, 4kHz, 20kHz) and decreasing multiplicative factor are proportional in output amplitude length (200, 50, 1).

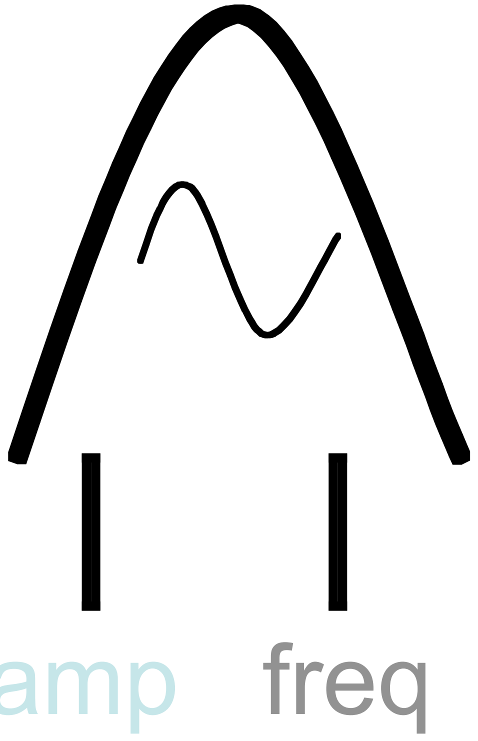
Top-Level Design



Walking the Tree

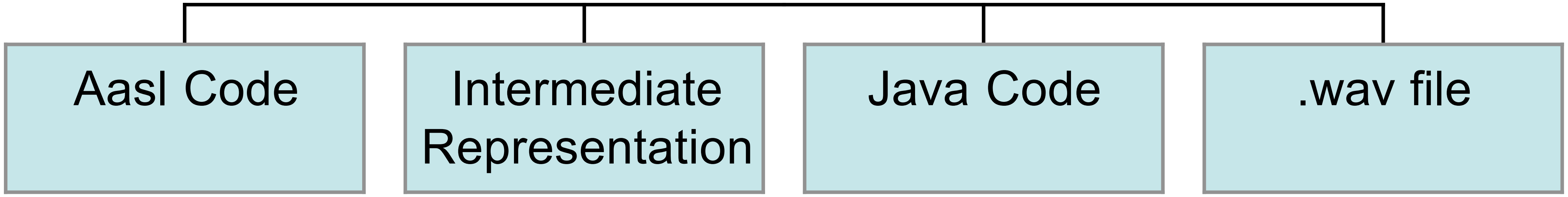
Asides from doing the static semantic checking, the tree walker is where each element is set up and the intermediate representation is built.

From each trait of a synthesis element, a single value is not enough, as it changes across multiple segments.



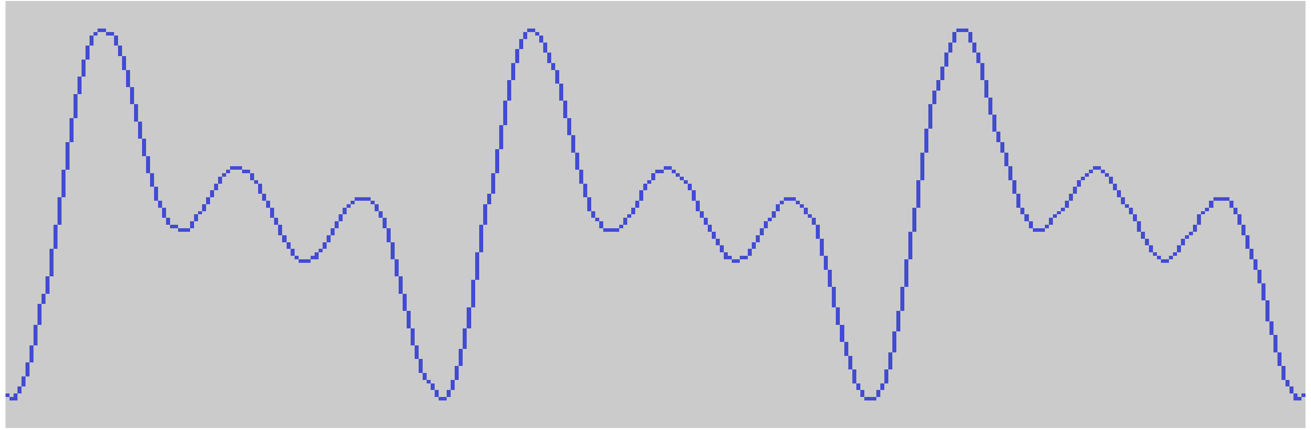
Amp array	1.0	0.5	e1	osc2	1.0
Freq array	440	440	e2	1.0	osc3
Wavetype array	SINE	SAW	SQUARE	SINE	SINE

AASL Architecture



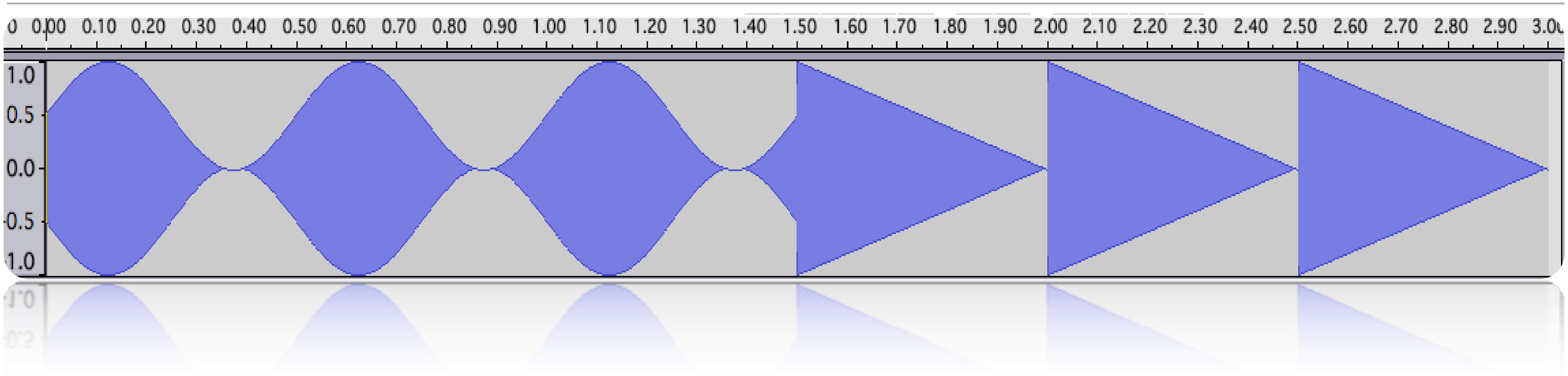
```
if(SEG==0)
         {osc1="SINE";}
else
         {osc1="SAW";}


//in the connect section…
osc2(osc1@440,osc1);
```



```
Envelope e1 = new Envelope(e1array, 5, tone_length);
Oscillator osc1 = new Oscillator("SINE",
          tone_length/segments, 1000.0, 0.75);
```

Segments and Control Structures



```
...head...
SEGMENTS = 2

 ..def section...
if (SEG == 0) {
     osc1 = “SINE”;
} else {
     osc1 = “SAW”;
}

..connection section...
osc2(osc1@440,osc1);
```

In the head, we split the output into 2 segments.

In segment 0, osc1 is a sine wave.
In segment 1, osc1 is a saw wave.

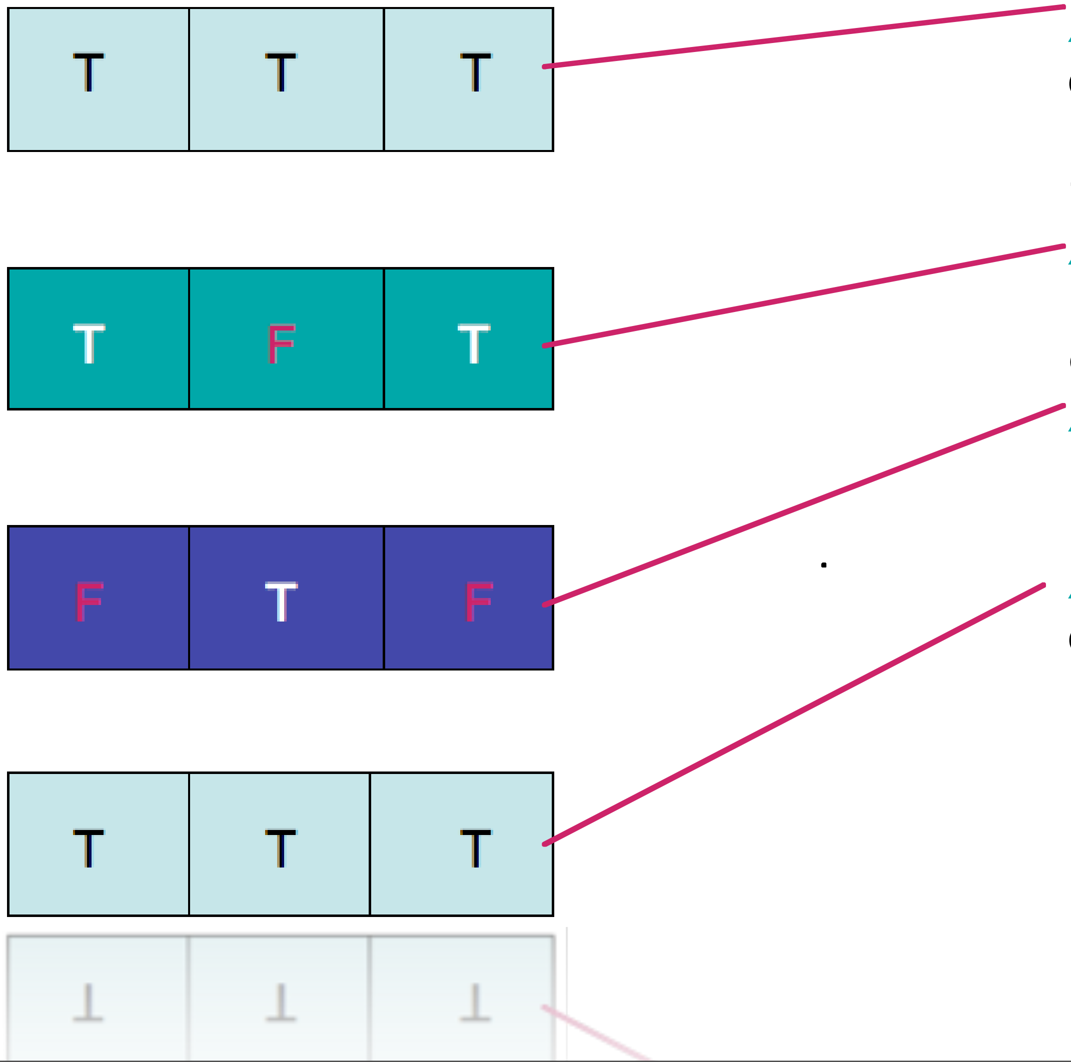
We use osc1 to control osc2’s amplitude, as shown above.
The pitch moves up and down at the same time, centering around 440Hz.

Segment Arrays

Segment Arrays keep track of which parts of the tone are currently being modified.

Every function begins with a global segment array in which all values are true and the code applies to all segments.

We may select only certain segments by using “if (SEG==x)” where ‘x’ is the segment to activate.



```
//global segment array
osc2=“REVSAW”;

if(SEG==0 || SEG==2)
//now we have this seg array
         {osc1="SINE";}
else
//for the else we invert the array
         {osc1="SAW";}

//return to global segment array
osc2=“SINE”;
```

Intermediate Representation

output	length	# segs	osc	mixer	env	func

One vector contains output name, tone length and number of segments. This is followed by a list of the elements and functions. (AaslExecutedFunction and AaslType).

They are split into individual vectors for each element type, a vector for functions, and a seperate vector for elements that attach directly to the output for at least one segment.

All oscillator banks have been split into their component oscillators during the tree walker

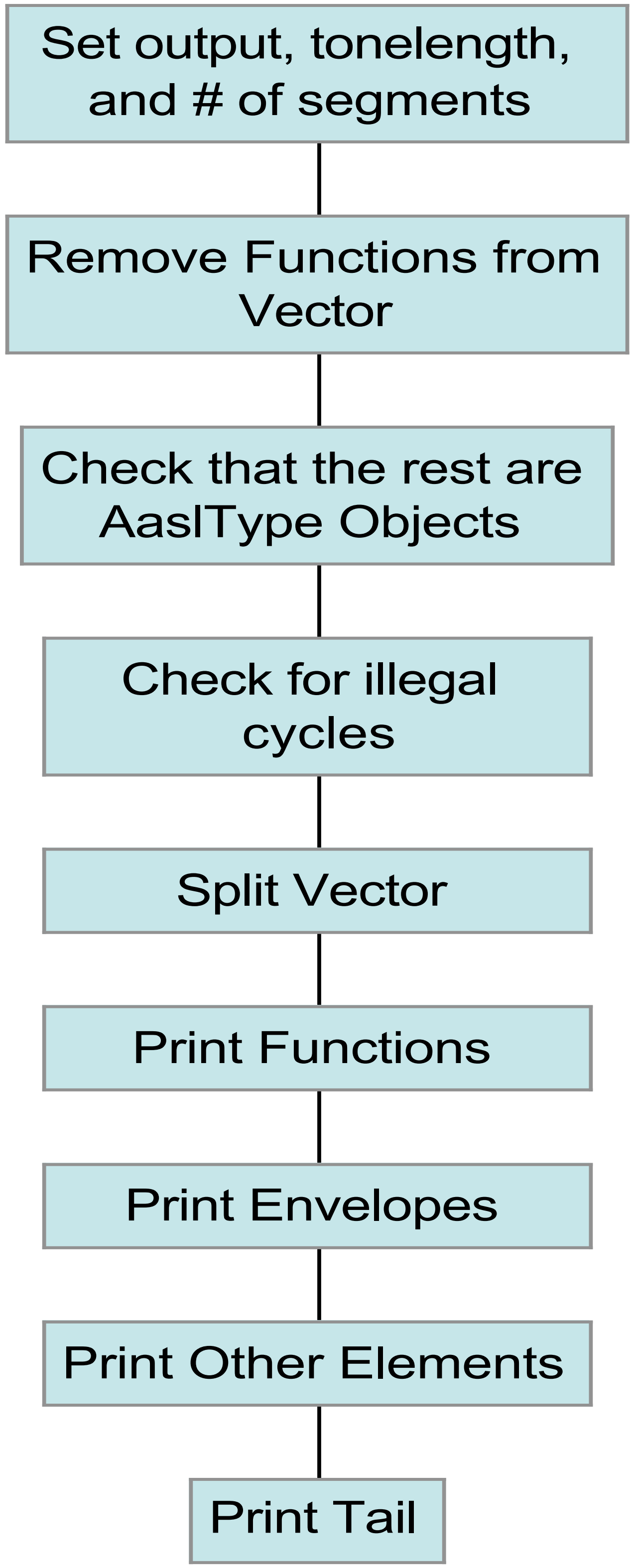
Intermediate Representation

length	# segs	osc	mixer	env	osc	osc

Each AaslExecutedFunction object had a similar vector for the elements defined with it.

Code is generated for each of the functions first, and then for the main function.

Code Generation



Set output, tonelength, and # of segments

Remove Functions from Vector

Check that the rest are AaslType Objects

Check for illegal cycles

Split Vector

Print Functions

Print Envelopes

Print Other Elements

Print Tail

Illegal cycles: an oscillator indirectly end up attached to itself.

Printing functions consists of nearly identical code generation process on their individual vectors.

Envelopes are printed first, since they are constant.

Oscillators with constant amplitude and frequency inputs are printed first.

Other oscillators may be printed once the elements attached to their inputs have been evaluated and printed.

Context

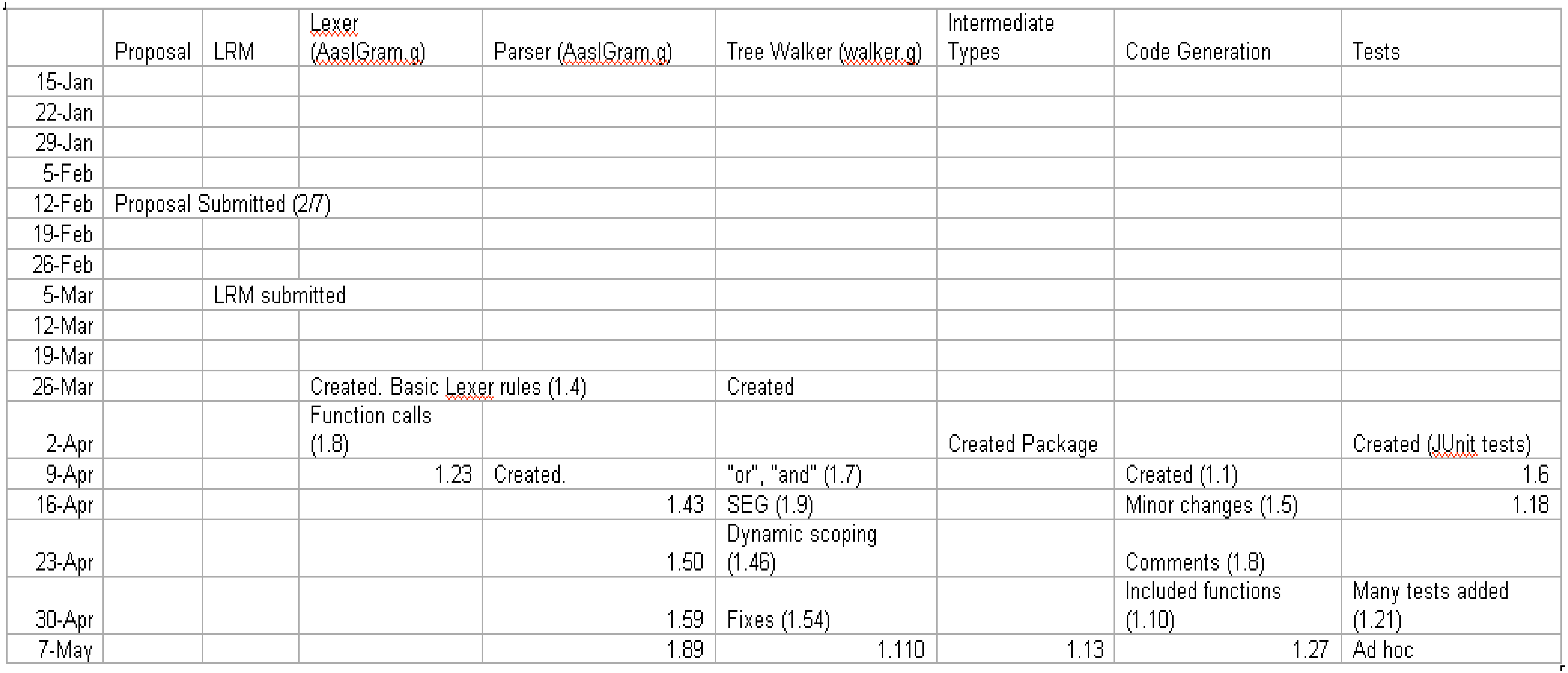
- Functionality
- Historical Devices
- Possibilities

Testing

- Unit Testing
  - gUnit
  - String Comparison
  - Jeremy D. Fren’s Test harness
- Integration Testing
  - Code Generation
  - Sample Programs

Reality

“No plan survives contact with the enemy.”



	Proposal	LRM	Lexer (AaslGram.g)	Parser (AaslGram.g)	Tree Walker (walker.g)	Intermediate Types	Code Generation	Tests
15-Jan								
22-Jan								
29-Jan								
5-Feb								
12-Feb	Proposal Submitted (2/7)							
19-Feb								
26-Feb								
5-Mar		LRM submitted						
12-Mar								
19-Mar								
26-Mar			Created. Basic Lexer rules (1.4)		Created			
2-Apr			Function calls (1.8)			Created Package		Created (JUnit tests)
9-Apr			1.23	Created.	"or", "and" (1.7)		Created (1.1)	1.6
16-Apr				1.43	SEG (1.9)		Minor changes (1.5)	1.18
23-Apr				1.50	Dynamic scoping (1.46)		Comments (1.8)	
30-Apr				1.59	Fixes (1.54)		Included functions (1.10)	Many tests added (1.21)
7-May				1.89	1.110	1.13	1.27	Ad hoc

Lessons Learned

- Rob
- Carlos Rene
- Vaishnav
- Albert

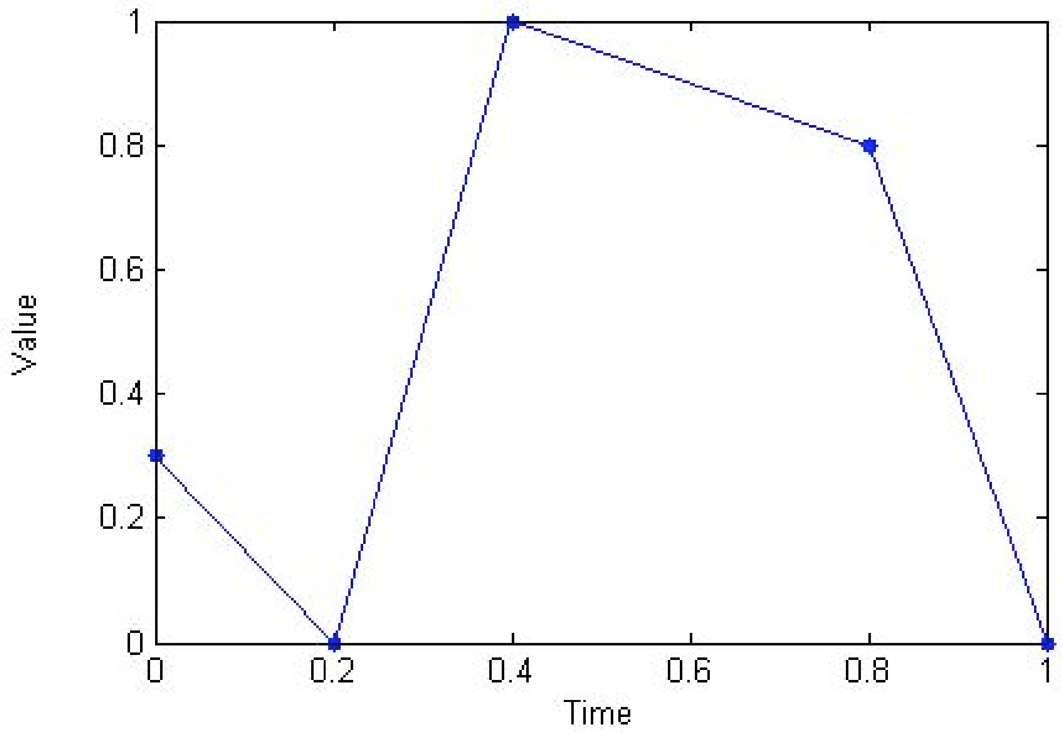
Examples

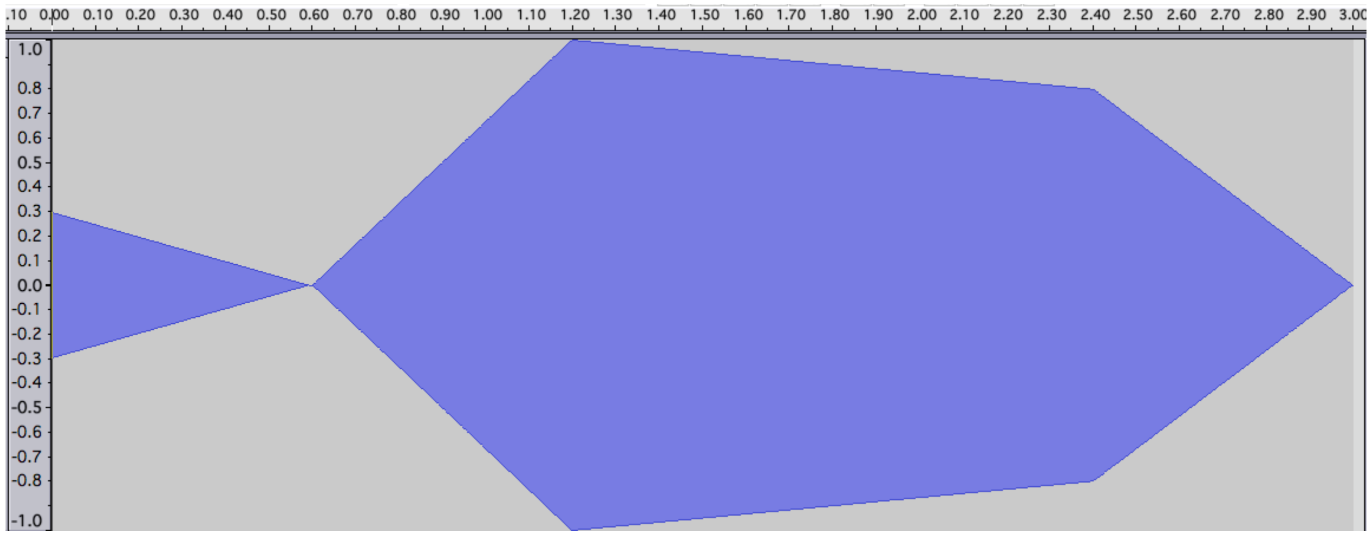
```
start header
    OUTPUT = “envtest1”
    TONELENGTH = 3
    SEGMENTS = 2
end header

start func main
    start def
        oscbank oscb1 = 2;
        if(SEG==2) {
            oscb1[0] = “SINE”;
        } else {
            oscb1[0] = “SAW”;
        }
        oscb1[1]=”SQUARE”;
        env e1 = {(0.0,0.3) (0.2,0.0) (0.4,1.0) (0.8,0.8) (1.0,0.0) };
        mixer s1;
    end def

    start connect
        oscb1[0](2,1.0);
        oscb1[1](oscb1[0]@440,e1);
        s1 = oscb1[1];
    end connect

    OUTPUT = s1;
end func main
```


Value
Time


.10
0.00
0.10
0.20
0.30
0.40
0.50
0.60
0.70
0.80
0.90
1.00
1.10
1.20
1.30
1.40
1.50
1.60
1.70
1.80
1.90
2.00
2.10
2.20
2.30
2.40
2.50
2.60
2.70
2.80
2.90
3.00
1.0
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-1.0