Reading 26: Little Languages

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

In this reading we will begin to explore the design of a **little language** for constructing and manipulating music. Here's the bottom line: when you need to solve a problem, instead of writing a *program* to solve just that one problem, build a *language* that can solve a range of related problems.

The goal for this reading is to introduce the idea of **representing code as data** and familiarize you with an initial version of the **music language** .

Representing code as data

Recall the `Formula` datatype from *Recursive Data Types* (../16-recursive-data-types/recursive/#another_example_boolean_formulas) :

```
Formula = Variable(name:String)
          + Not(formula:Formula)
          + And(left:Formula, right:Formula)
          + Or(left:Formula, right:Formula)
```

We used instances of `Formula` to take propositional logic formulas, e.g. $(p \lor q) \land (\neg p \lor r)$, and represent them in a data structure, e.g.:

```
And(Or(Variable("p"), Variable("q")),
    Or(Not(Variable("p")), Variable("r")))
```

In the parlance of grammars and parsers, formulas are a *language* , and `Formula` is an *abstract syntax tree* (../18-parser-generators/) .

But why did we define a `Formula` type? Java already has a way to represent expressions of Boolean variables with *logical and* , *or* , and *not* . For example, given `boolean` variables `p` , `q` , and `r` :

```
(p || q) && ((!p) || r)
```

Done!

The answer is that the Java code expression `(p || q) && ((!p) || r)` is evaluated as soon as we encounter it in our running program. The `Formula` value `And(Or(...), Or(...))` is a **first-class value** that can be stored, passed and returned from one method to another, manipulated, and evaluated now or later (or more than once) as needed.

The `Formula` type is an example of **representing code as data** , and we've seen many more.

Consider this functional object (../25-map-filter-reduce/#first-class_functions_in_java) :

```
class VariableNameComparator implements Comparator<Variable> {
    public int compare(Variable v1, Variable v2) {
        return v1.name().compareTo(v2.name());
    }
}
```

An instance of `VariableNameComparator` is a value that can be passed around, returned, and stored. But at any time, the function that it represents can be invoked by calling its `compare` method with a couple of `Variable` arguments:

```
Variable v1, v2;
Comparator<Variable> c = new VariableNameComparator();
...
int a = c.compare(v1, v2);
int b = c.compare(v2, v1);
SortedSet<Variable> vars = new TreeSet<>(c); // vars is sorted by name
```

Lambda expressions allow us to create functional objects with a compact syntax:

```
Comparator<Variable> c = (v1, v2) -> v1.name().compareTo(v2.name());
```

Building languages to solve problems

When we define an abstract data type (../12-abstract-data-types/) , we're extending the universe of built-in types provided by Java to include a new type, with new operations, appropriate to our problem domain. This new type is like a new language: a new set of nouns (values) and verbs (operations) we can manipulate. Of course, those nouns and verbs are abstractions built on top the existing nouns and verbs which were themselves already abstractions.

A *language* has greater flexibility than a mere *program* , because we can use a language to solve a large class of related problems, instead of just a single problem.

- That's the difference between writing `(p || q) && ((!p) || r)` and devising a `Formula` type to represent the semantically-equivalent Boolean formula.
- And it's the difference between writing a matrix multiplication function and devising a `MatrixExpression` type (../16-recursive-data-types/matexpr/) to represent matrix multiplications — and store them, manipulate them, optimize them, evaluate them, and so on.

First-class functions and functional objects enable us to create particularly powerful languages because we can capture patterns of computation as reusable abstractions.

Music language

In class, we will design and implement a language for generating and playing music. To prepare, let's first understand the Java APIs for playing music with the MIDI (//en.wikipedia.org/wiki/MIDI) synthesizer. We'll see how to write a *program* to play MIDI music. Then we'll begin to develop our music *language* by writing a recursive abstract data type for simple musical tunes. We'll choose a notation for writing music in strings, and we'll implement a parser to create instances of our `Music` type.

The **full source code for the basic music language (https://github.com/mit6005/sp16-ex26-music-starting)** is on GitHub.

Clone the **sp16-ex26-music-starting (https://github.com/mit6005/sp16-ex26-music-starting)** repo so you can run the code and follow the discussion below.

Playing MIDI music

`music.midi.MidiSequencePlayer` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/midi/MidiSequencePlayer.java) uses the Java MIDI APIs to play sequences of notes. It's quite a bit of code, and you don't need to understand how it works.

`MidiSequencePlayer` implements the **`music.SequencePlayer`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java) interface, allowing clients to use it without depending on the particular MIDI implementation. We *do* need to understand this interface and the types it depends on:

`addNote : SequencePlayer × Instrument × Pitch × double × double → void` (SequencePlayer.java:15 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15)) is the workhorse of our music player. Calling this method schedules a musical pitch to be played at some time during the piece of music.

`play : SequencePlayer → void` (SequencePlayer.java:20 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L20)) actually plays the music. Until we call this method, we're just scheduling music that will, eventually, be played.

The `addNote` operation depends on two more types:

`Instrument` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Instrument.java) is an enumeration of all the available MIDI instruments.

`Pitch` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java) is an abstract data type for musical pitches (think keys on the piano keyboard).

Read and understand the **`Pitch` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java)** documentation and the specifications for its **public constructor (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java#L57-68)** and all its **public methods (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Pitch.java#L70-122)** .

Our music data type will rely on `Pitch` in its rep, so be sure to understand the `Pitch` spec as well as its rep and abstraction function.

Using the MIDI sequence player and `Pitch` , we're ready to write code for our first bit of music!

Read and understand the **`music.examples.ScaleSequence` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleSequence.java)** code.

Run the main method in `ScaleSequence` . You should hear a one-octave scale!

Music data type

The `Pitch` datatype is useful, but if we want to represent a whole piece of music using `Pitch` objects, we should create an abstract data type to encapsulate that representation.

To start, we'll define the **`Music`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java) type with a few operations:

`notes : String × Instrument → Music` (MusicLanguage.java:51 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L51)) makes a new Music from a string of simplified abc notation, described below.

`duration : Music → double` (Music.java:11 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L11)) returns the duration, in beats, of the piece of music.

`play : Music × SequencePlayer × double → void` (Music.java:18 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L18)) plays the piece of music using the given sequence player.

We'll implement `duration` and `play` as instance methods of `Music` , so we declare them in the `Music` interface.

`notes` will be a static factory method; rather than put it in `Music` (which we could do), we'll put it in a separate class: **`MusicLanguage`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java) will be our place for all the static methods we write to operate on `Music` .

Now that we've chosen some operations in the spec of `Music` , let's choose a representation.

- Looking at `ScaleSequence` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleSequence.java) , the first concrete variant that might jump out at us is one to capture the information in each call to `addNote` : a particular pitch on a particular instrument played for some amount of time. We'll call this a **`Note`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java) .
- The other basic element of music is the silence between notes: **`Rest`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java) .
- Finally, we need a way to glue these basic elements together into larger pieces of music. We'll choose a tree-like structure: **`Concat(m1,m2:Music)`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java) represents `m1` followed by `m2` , where `m1` and `m2` are any music.

  This tree structure turns out to be an elegant decision as we further develop our `Music` type later on. In a real design process, we might iterate on the recursive structure of `Music` before we find the best implementation.

Here's the datatype definition:

```
Music = Note(duration:double, pitch:Pitch, instrument:Instrument)
        + Rest(duration:double)
        + Concat(m1:Music, m2:Music)
```

Composite

`Music` is an example of the **composite pattern** , in which we treat both single objects (*primitives* , e.g. `Note` and `Rest`) and groups of objects (*composites* , e.g. `Concat`) the same way.

- `Formula` is also an example of the composite pattern.
- The GUI view tree relies heavily on the composite pattern: there are *primitive views* like `JLabel` and `JTextField` that don't have children, and *composite views* like `JPanel` and `JScollPage` that do contain other views as children. Both implement the common `JComponent` interface.

The composite pattern gives rise to a tree data structure, with primitives at the leaves and composites at the internal nodes.

Emptiness

One last design consideration: how do we represent the empty music? It's always good to have a representation for *nothing* , and we're certainly not going to use `null` .

We could introduce an `Empty` variant, but instead we'll use a `Rest` of duration `0` to represent emptiness.

Implementing basic operations

First we need to create the **`Note`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java) , **`Rest`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java) , and **`Concat`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java) variants. All three are straightforward to implement, starting with constructors, `checkRep` , some observers, `toString` , and the equality methods.

- Since the `duration` operation is an instance method, each variant implements `duration` appropriately.
- The `play` operation is also an instance method; we'll discuss it below under *implementing the player* .

And we'll discuss the `notes` operation in *implementing the parser* .

Read and understand the **`Note` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java)** , **`Rest` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java)** , and **`Concat` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java)** classes.

To avoid representation exposure, let's add some additional static factory methods to the `Music` interface:

`note : double × Pitch × Instrument → Music` (MusicLanguage.java:92 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L92))

`rest : double → Music` (MusicLanguage.java:100 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L100))

`concat : Music × Music → Music` (MusicLanguage.java:113 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L113)) is our first producer operation.

All three of them are easy to implement by constructing the appropriate variant.

Music notation

We will write pieces of music using a simplified version of **abc notation** (//en.wikipedia.org/wiki/ABC_notation) , a text-based music format.

We've already been representing pitches using their familiar letters. Our simplified abc notation represents sequences of **notes** and **rests** with syntax for indicating their **duration** , **accidental** (sharp or flat), and **octave** .

For example:

`C D E F G A B C' B A G F E D C` represents the one-octave ascending and descending C major scale we played in `ScaleSequence` . `C` is middle C, and `C'` is C one octave above middle C. Each note is a quarter note.

`C/2 D/2 _E/2 F/2 G/2 _A/2 _B/2 C'` is the ascending scale in C minor, played twice as fast. The E, A, and B are flat. Each note is an eighth note.

Read and understand the specification of **`notes` in `MusicLanguage` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L20-50)** .

You don't need to understand the parser implementation yet, but you should understand the simplified abc notation enough to make sense of the examples.

If you're not familiar with music theory — why is an octave 8 notes but only 12 semitones? — don't worry. You might not be able to look at the abc strings and guess what they sound like, but you can understand the point of choosing a convenient textual syntax.

Implementing the parser

The `notes` method parses strings of simplified abc notation into `Music` .

`notes : String × Instrument → Music` (MusicLanguage.java:51 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L51)) splits the input into individual symbols (e.g. `A,,/2` , `.1/2`). We start with the empty `Music` , `rest(0)` , symbols are parsed individually, and we build up the `Music` using `concat` .

`parseSymbol : String × Instrument → Music` (MusicLanguage.java:62 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L62)) returns a `Rest` or a `Note` for a single abc symbol (`symbol` in the grammar). It only parses the type (rest or note) and duration; it relies on `parsePitch` to handle pitch letters, accidentals, and octaves.

`parsePitch : String → Pitch` (MusicLanguage.java:77 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/MusicLanguage.java#L77)) returns a `Pitch` by parsing a `pitch` grammar production. You should be able to understand the recursion — what's the base case? What are the recursive cases?

Implementing the player

Recall our operation for playing music:

`play : Music × SequencePlayer × double → void` (Music.java:18 (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Music.java#L18)) plays the piece of music using the given sequence player after the given number of beats delay.

Why does this operation take `atBeat` ? Why not simply play the music *now* ?

If we define `play` in that way, we won't be able to play sequences of notes over time unless we actually *pause* during the `play` operation, for example with `Thread.sleep` . Our sequence player's `addNote` operation (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/SequencePlayer.java#L15) is already designed to schedule notes in the future — it handles the delay.

With that design decision, it's straightforward to implement `play` in every variant of `Music` .

Read and understand the **`Note.play` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Note.java#L55)** , **`Rest.play` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Rest.java#L33)** , and **`Concat.play` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/Concat.java#L51)** methods.

You should be able to follow their recursive implementations.

Just one more piece of utility code before we're ready to jam: **`music.midi.MusicPlayer`** (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/midi/MusicPlayer.java) plays a `Music` using the `MidiSequencePlayer` . `Music` doesn't know about the concrete type of the sequence player, so we need a bit of code to bring them together.

Bringing this *all* together, let's use the `Music` ADT:

Read and understand the **`music.examples.ScaleMusic` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/ScaleMusic.java)** code.

Run the main method in `ScaleMusic` . You should hear the same one-octave scale again.

That's not very exciting, so **read `music.examples.RowYourBoatInitial` (https://github.com/mit6005/sp16-ex26-music-starting/blob/master/src/music/examples/RowYourBoatInitial.java)** and **run the main method.** You should hear *Row, row, row your boat* !

Can you follow the flow of the code from calling `notes(..)` to having an instance of `Music` to the recursive `play(..)` call to individual `addNote(..)` calls?

To be continued

Playing *Row, row, row your boat* is pretty exciting, but so far the most powerful thing we've done is not so much the *music language* as it is the very basic *music parser* . Writing music using the simplified abc notation is clearly much more **easy to understand** , **safe from bugs** , and **ready for change** than writing page after page of `addNote` `addNote` `addNote` …

In class, we'll expand our music language and turn it into a powerful tool for constructing and manipulating complex musical structures.