January 22snd 2015

Project presentation

Ruled Random Generation

Valérian Pittet

Outline

1. **Model** Presentation

2. **Generation** Procedure

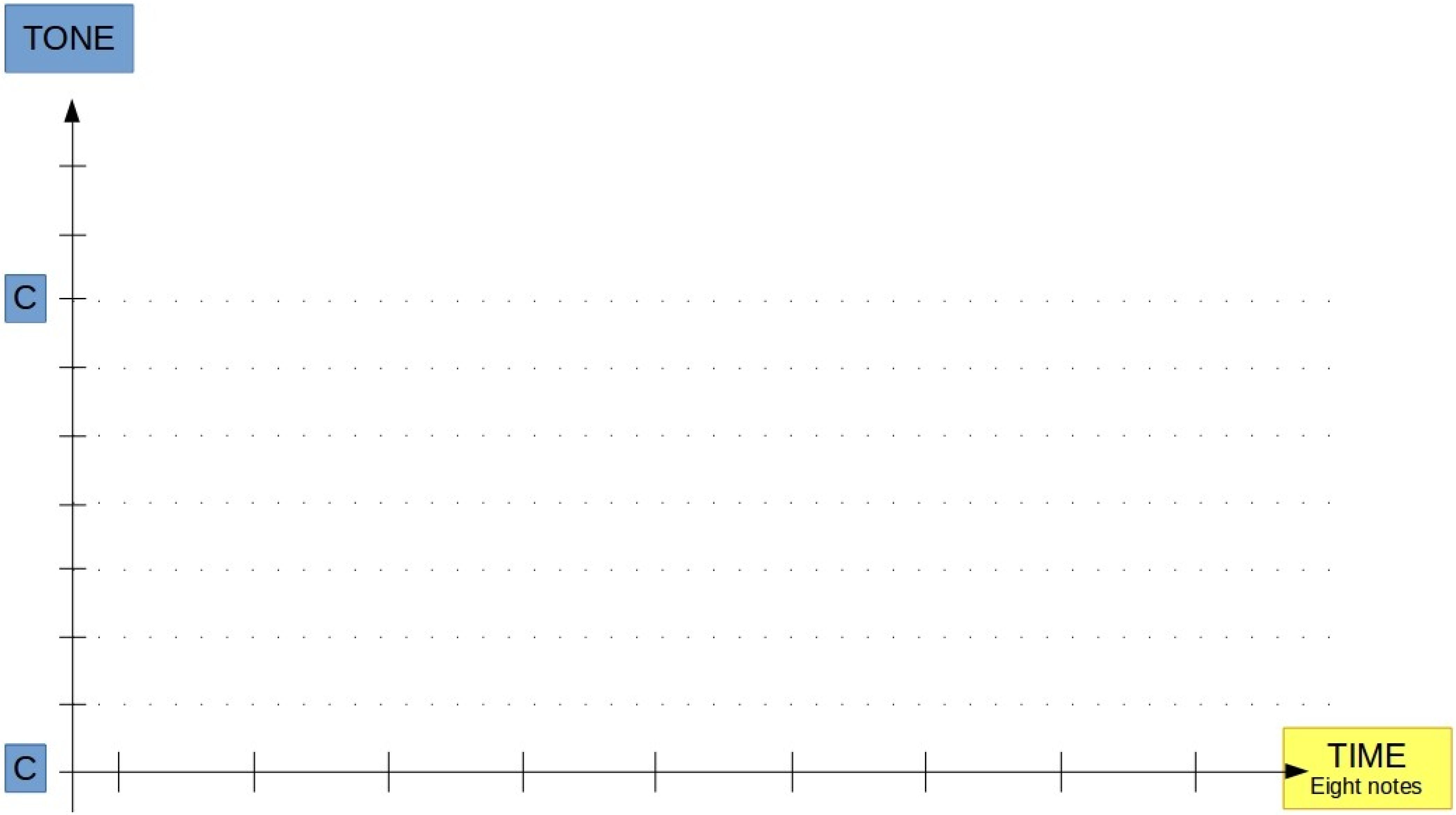
3. **Example** Building

1. Model

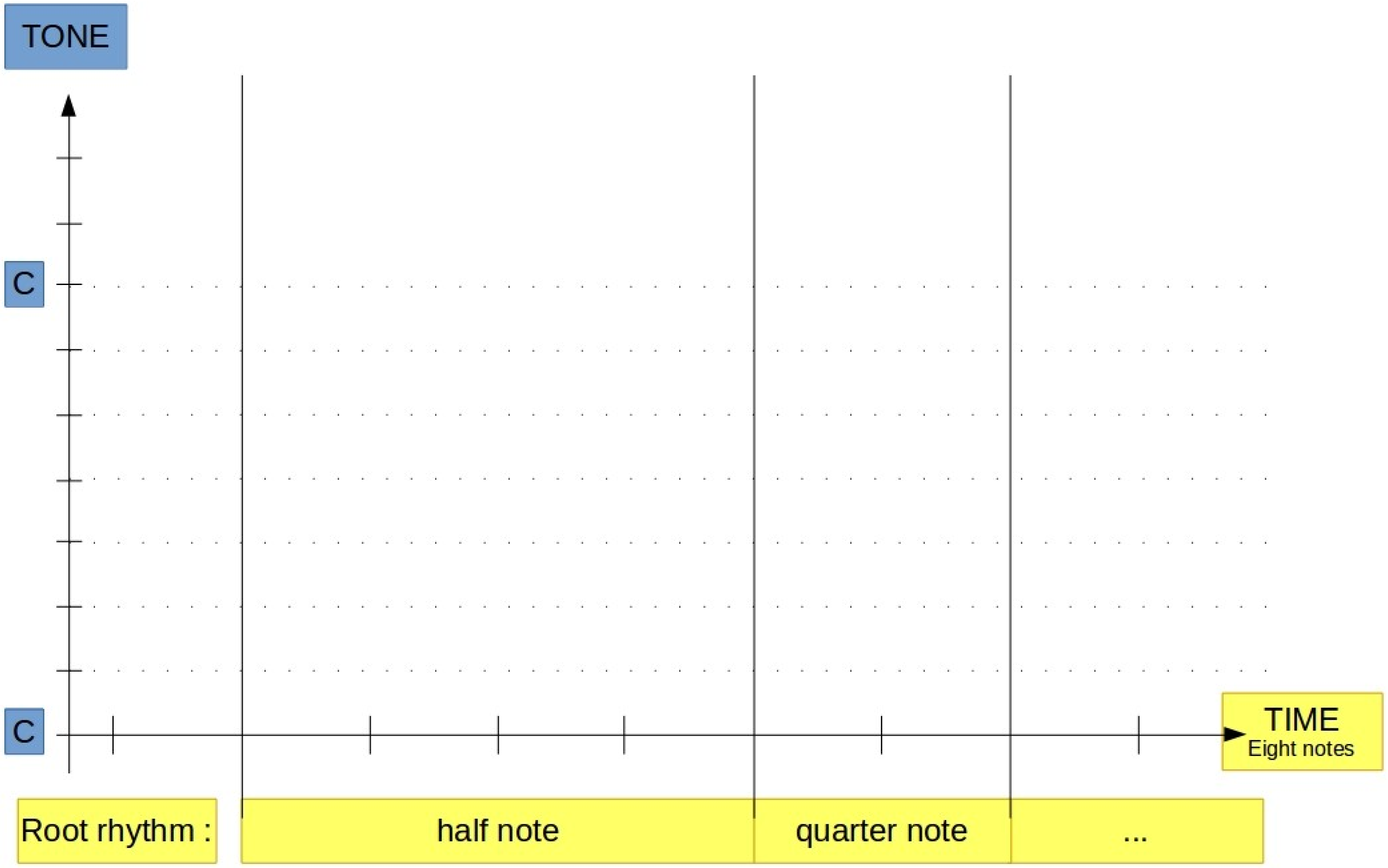
The music is represented as sequences of cells.
Each cell is composed with four elements.

1. **Root Rhythm** : cells duration
2. **Chord** : harmonic progression
3. **Rhythmic Cell** : rhythmic structure of cell
4. **Melodie** : sequence of tones

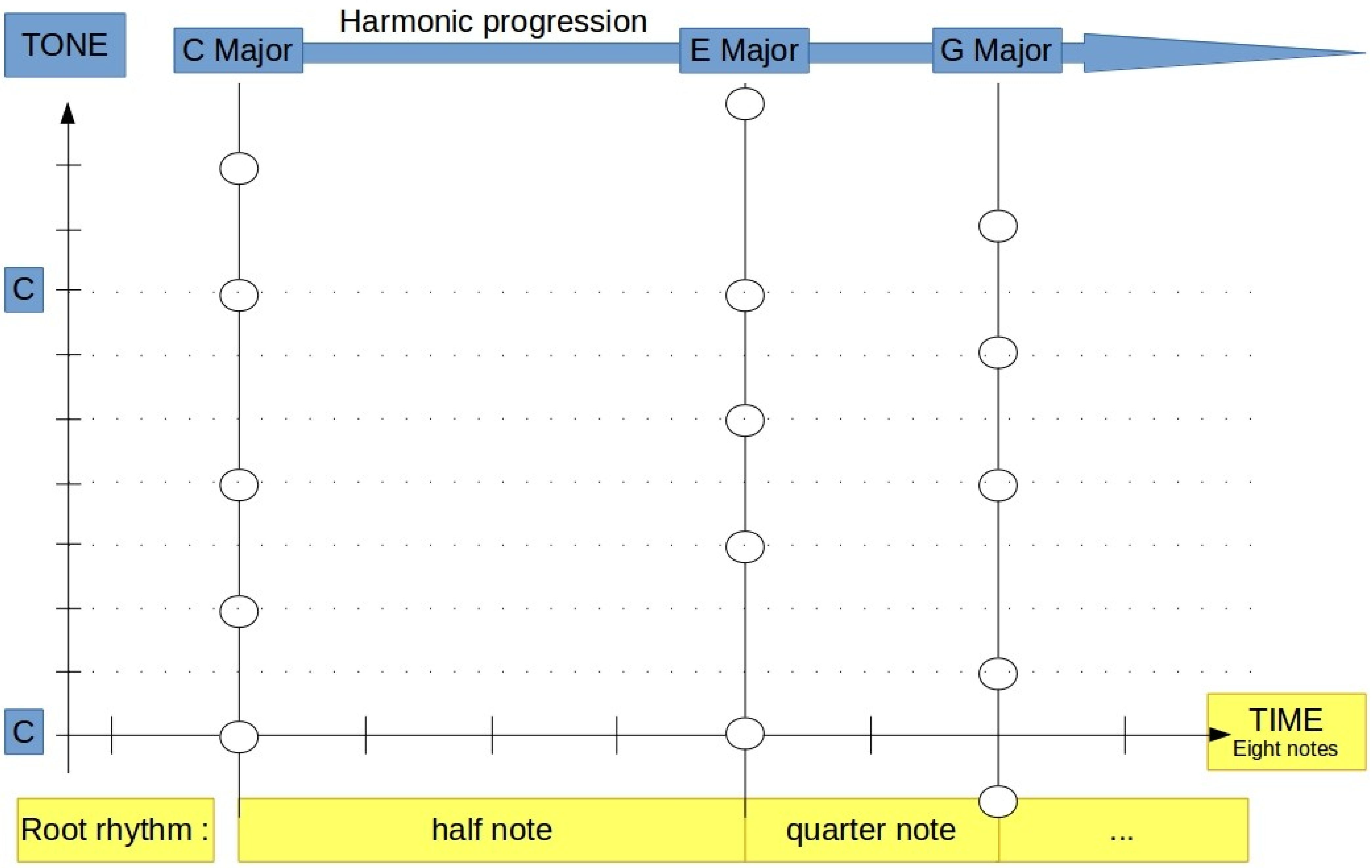
Model : empty



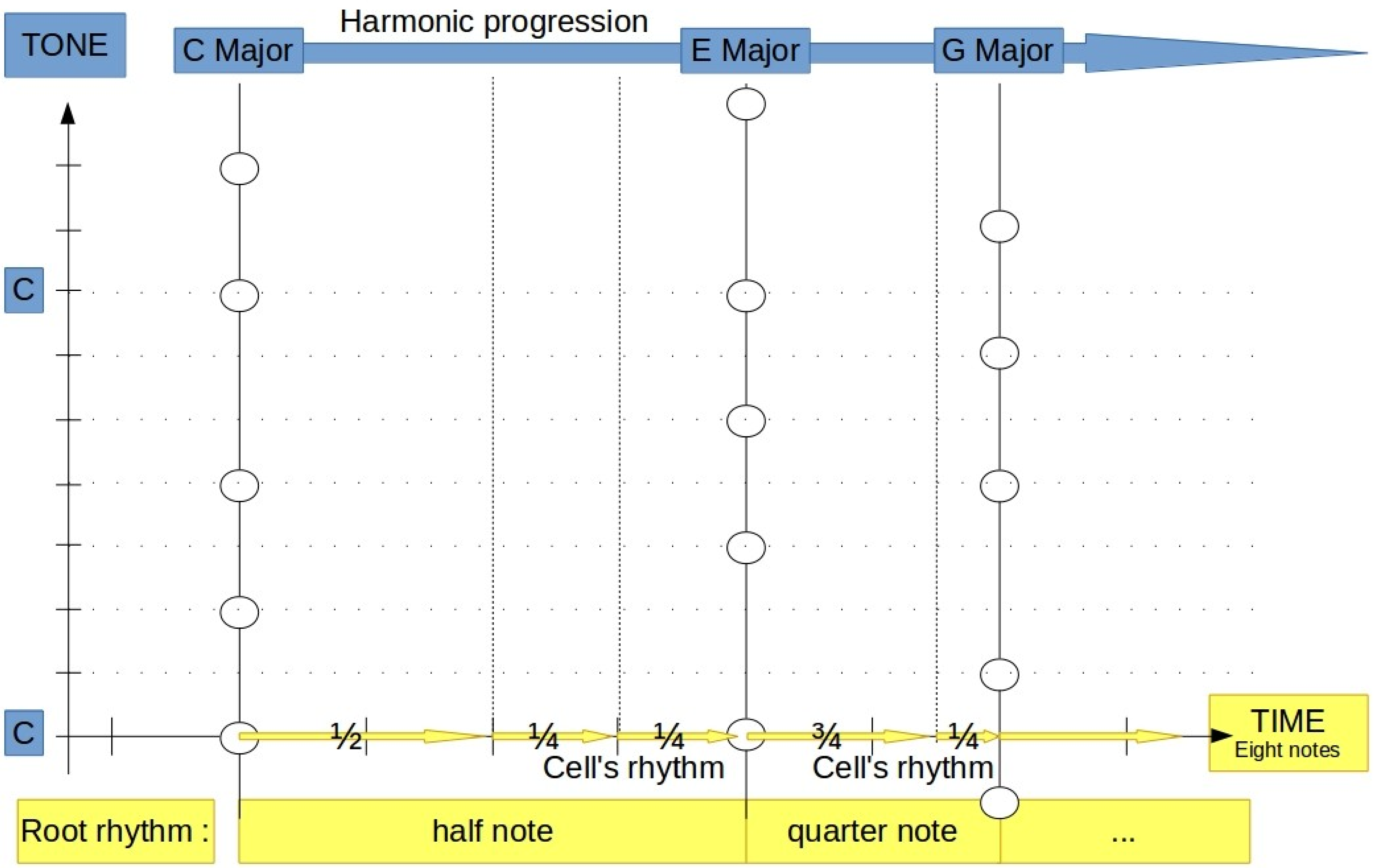
Model : root rhythm



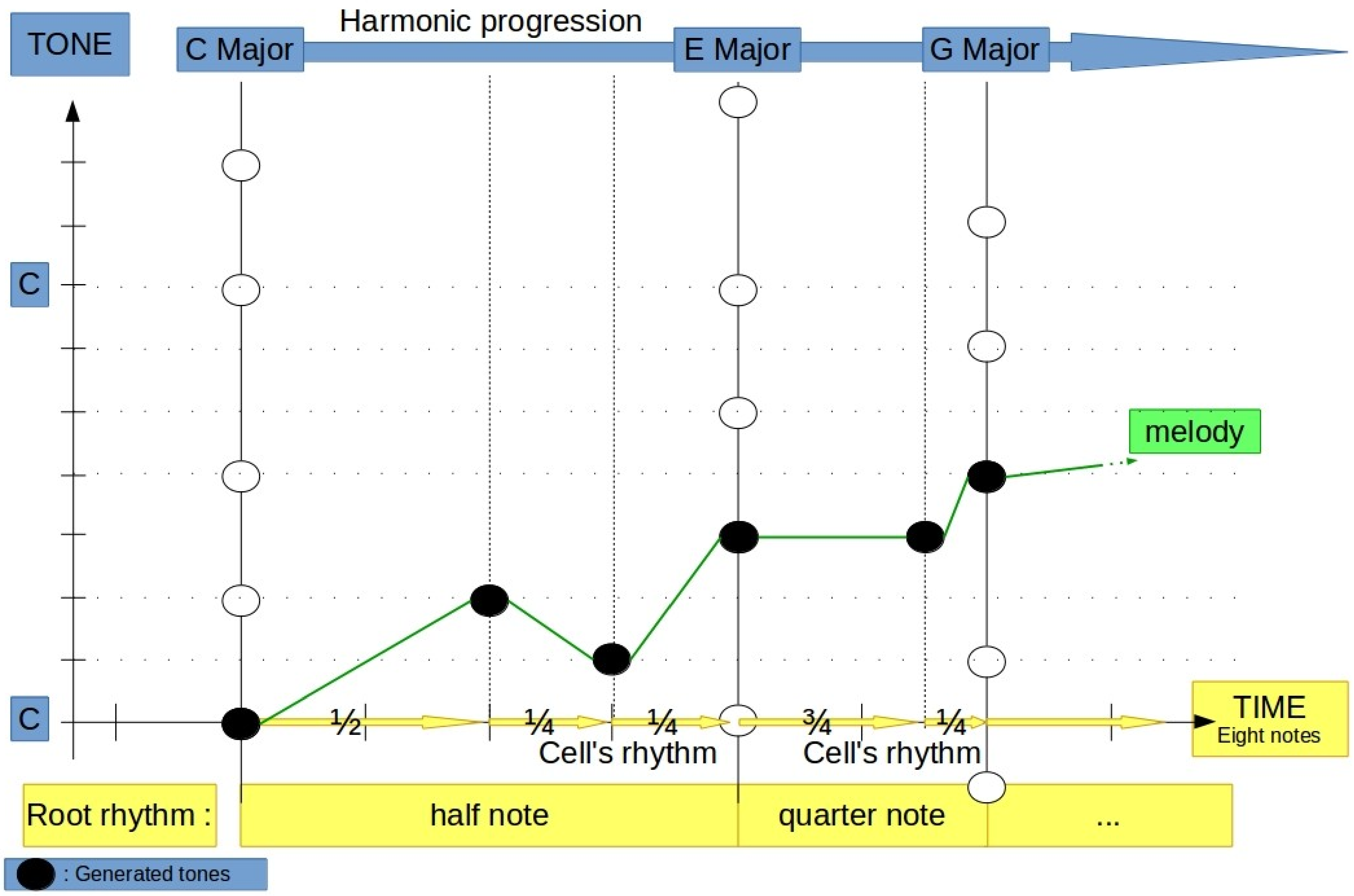
Model : chords



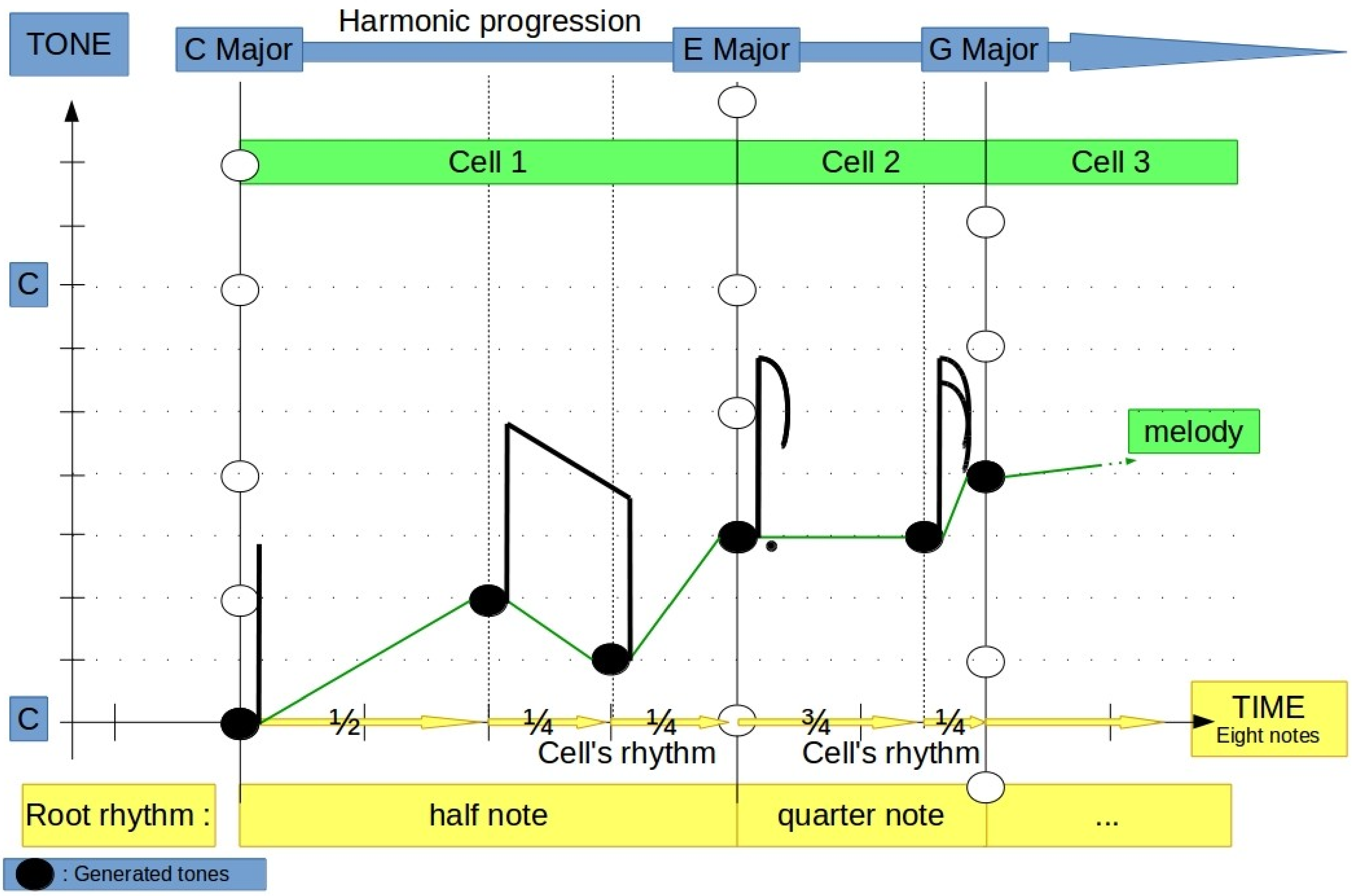
Model : rhythm cells



Model : tones


TONE
Harmonic progression
C Major
E Major
G Major
C
C
melody
½
¼
¼
¾
¼
TIME
Eight notes
Cell's rhythm
Cell's rhythm
Root rhythm :
half note
quarter note
...
● : Generated tones

Model : completed



2. Generation

The generation relies on grammar representation and manipulation

1. **Intermediate State** : grammar management
2. **Cell Generation** : the algorithm itself
3. **Closing Conditions** : to stop or not to stop ?
4. **Refinements** : interactive grammars

2. Generation

The generation relies on grammar representation and manipulation

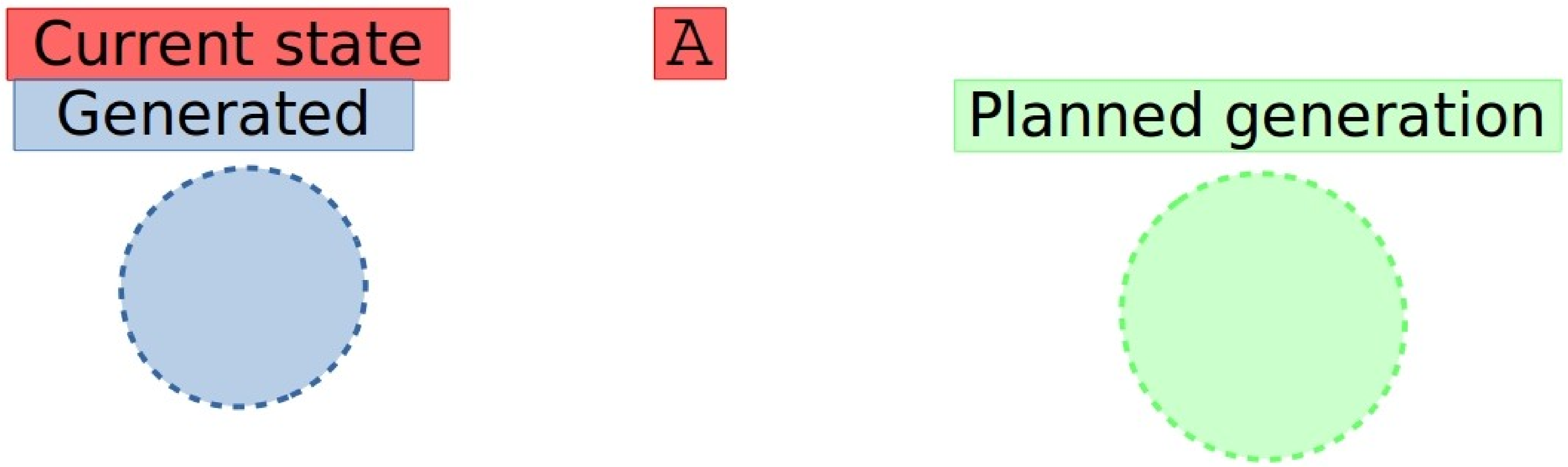
1. **Intermediate State** : grammar management

2. **Cell Generation** : the algorithm itself

3. **Closing Conditions** : to stop or not to stop ?

4. **Refinements** : interactive grammars

Intermediate State



Generated Tree as List

...

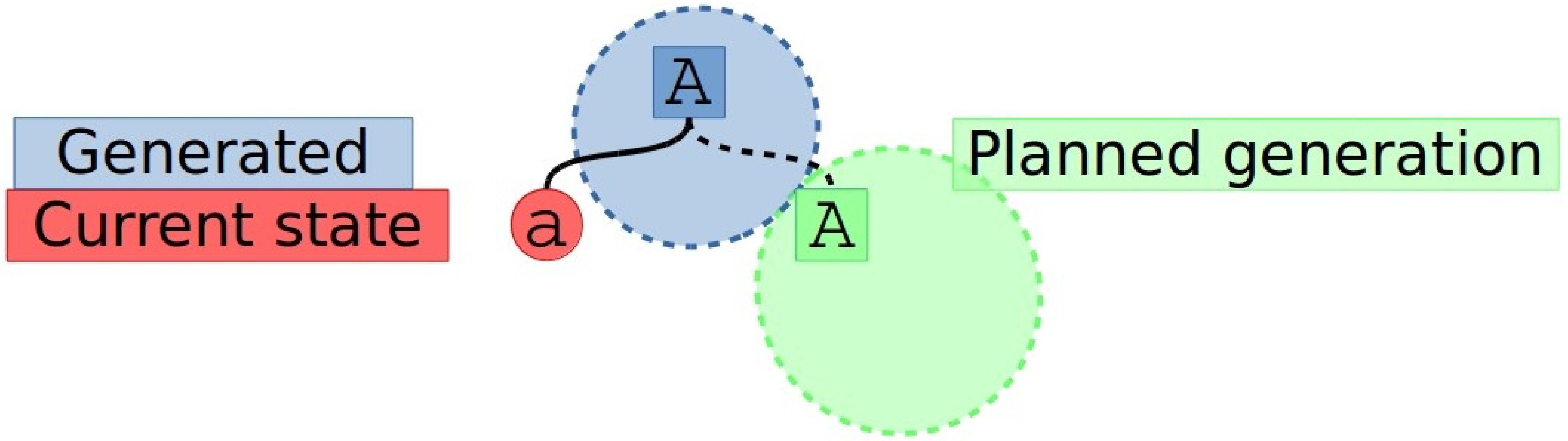
Planned Tasks as Stack

A

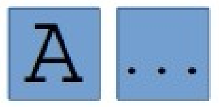
```
A ::= ( 'a'  A  )
    | (  A  'b' )
    |   ε
```

Sample Grammar

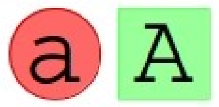
Intermediate State



Generated Tree as List



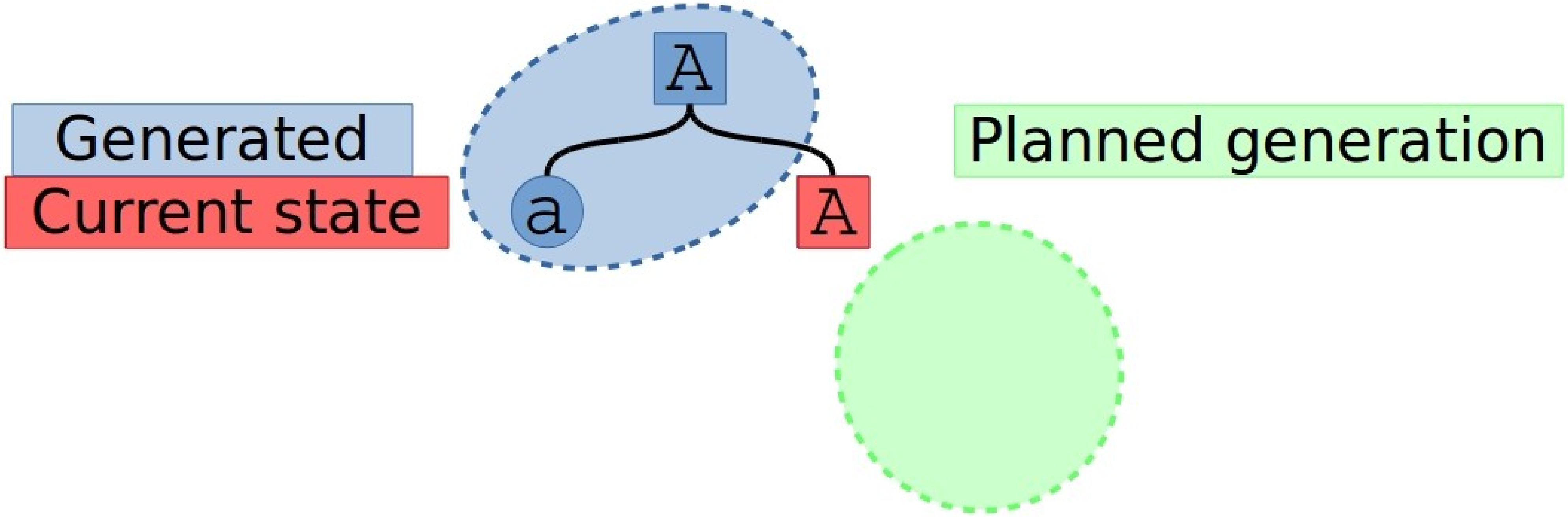
Planned Tasks as Stack



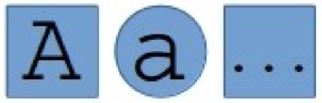
```
A ::= ( 'a'  A  )
    | (  A  'b' )
    |   ε
```

Sample Grammar

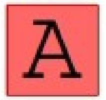
Intermediate State



Generated Tree as List



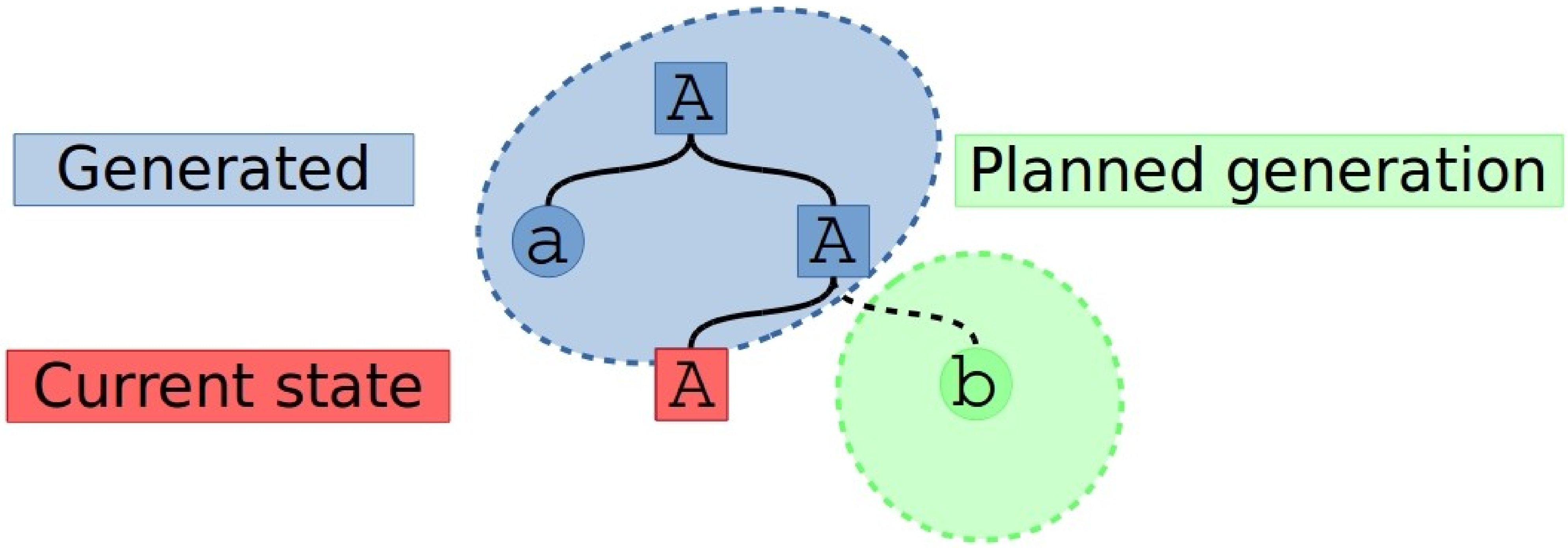
Planned Tasks as Stack



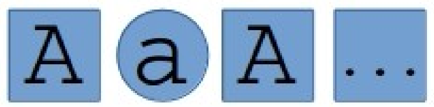
```
A ::= ( 'a'  A  )
    | (  A  'b' )
    |   ε
```

Sample Grammar

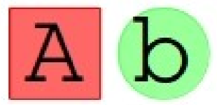
Intermediate State



Generated Tree as List



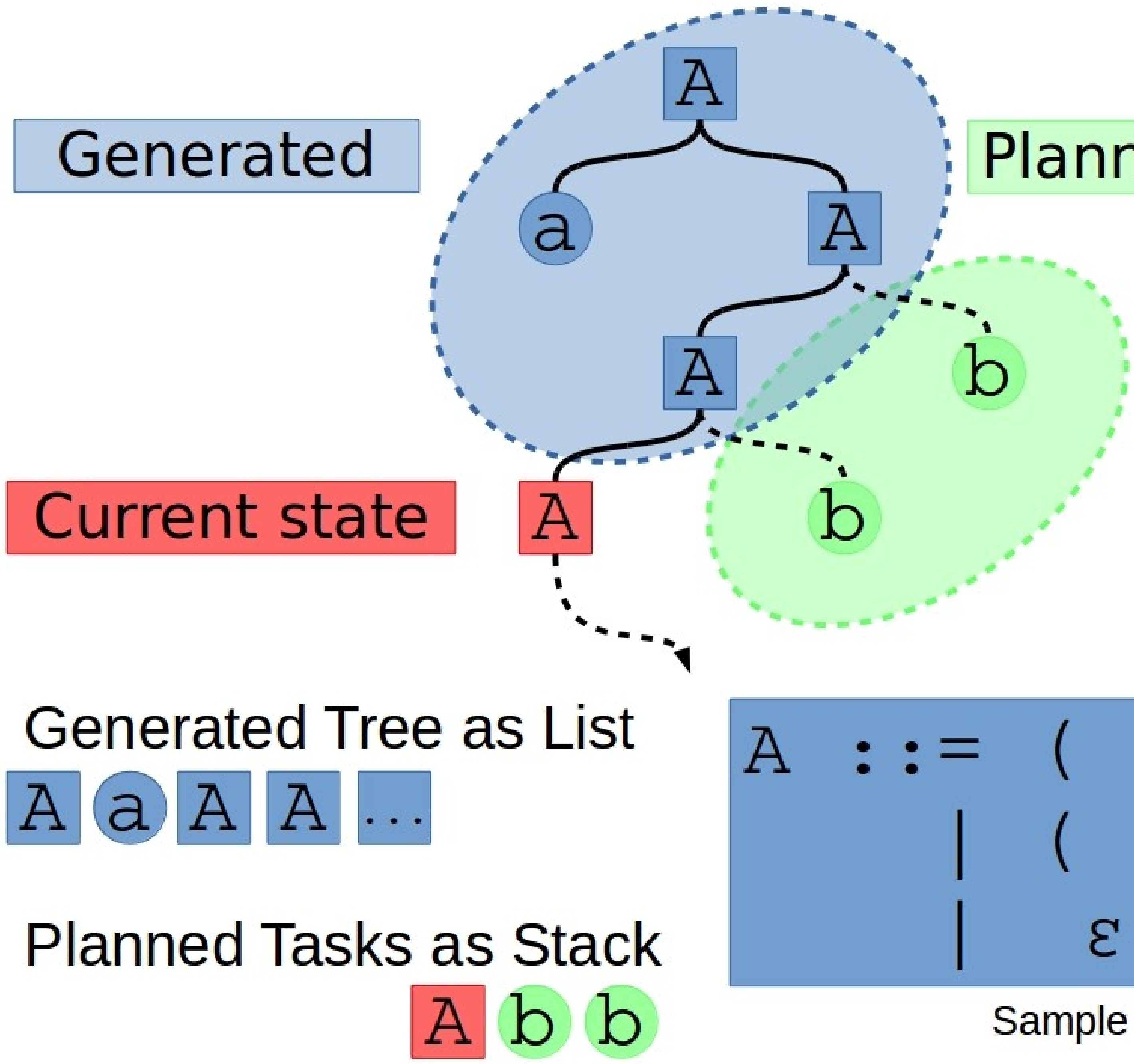
Planned Tasks as Stack



```
A ::= ( 'a'  A  )
    | (  A  'b' )
    |   ε
```

Sample Grammar

Intermediate State



```
A ::= ( 'a'  A  )
    | (  A  'b' )
    |   ε
```

Sample Grammar

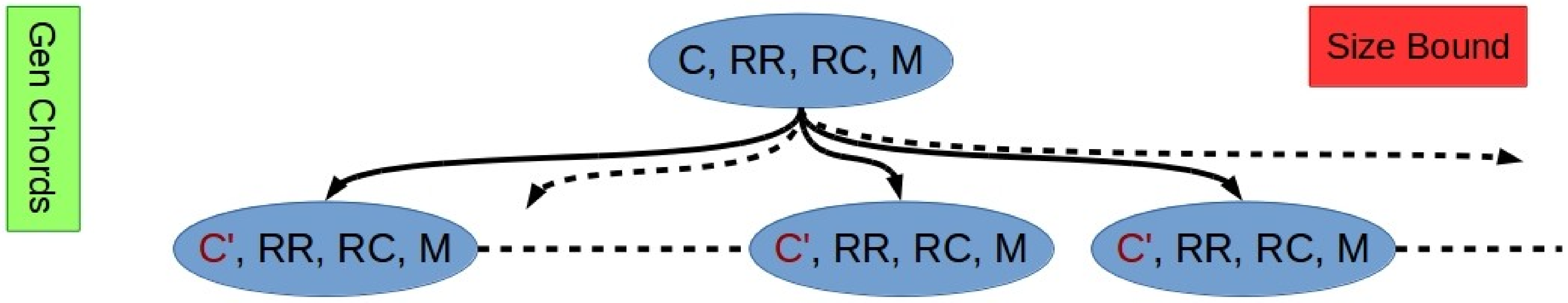
2. Generation

The generation relies on grammar representation and manipulation

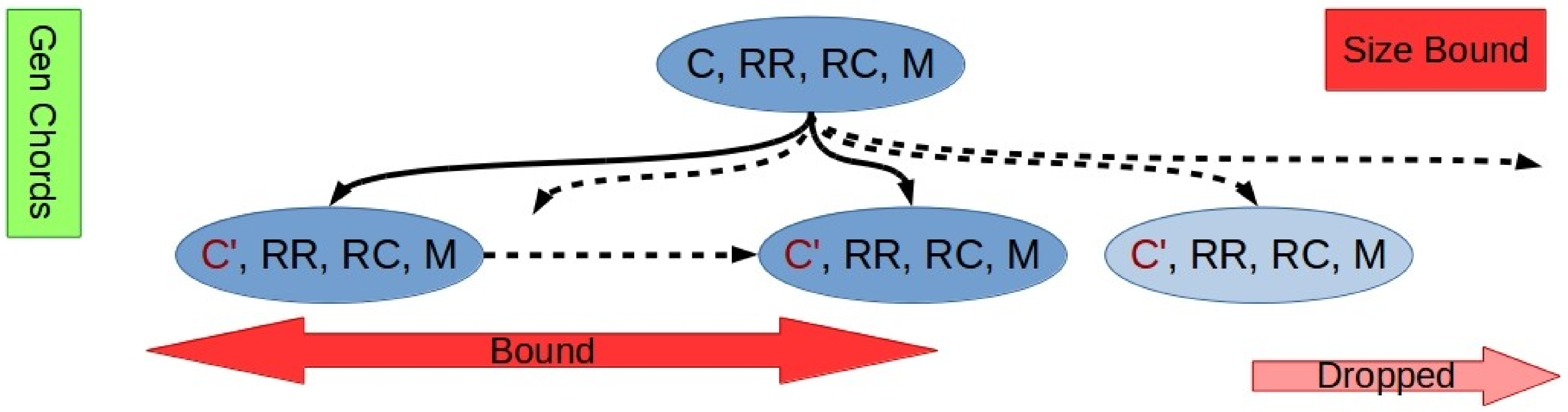
1. **Intermediate State** : grammar management
2. **Cell Generation** : the algorithm itself
3. **Closing Conditions** : to stop or not to stop ?
4. **Refinements** : interactive grammars

Generation

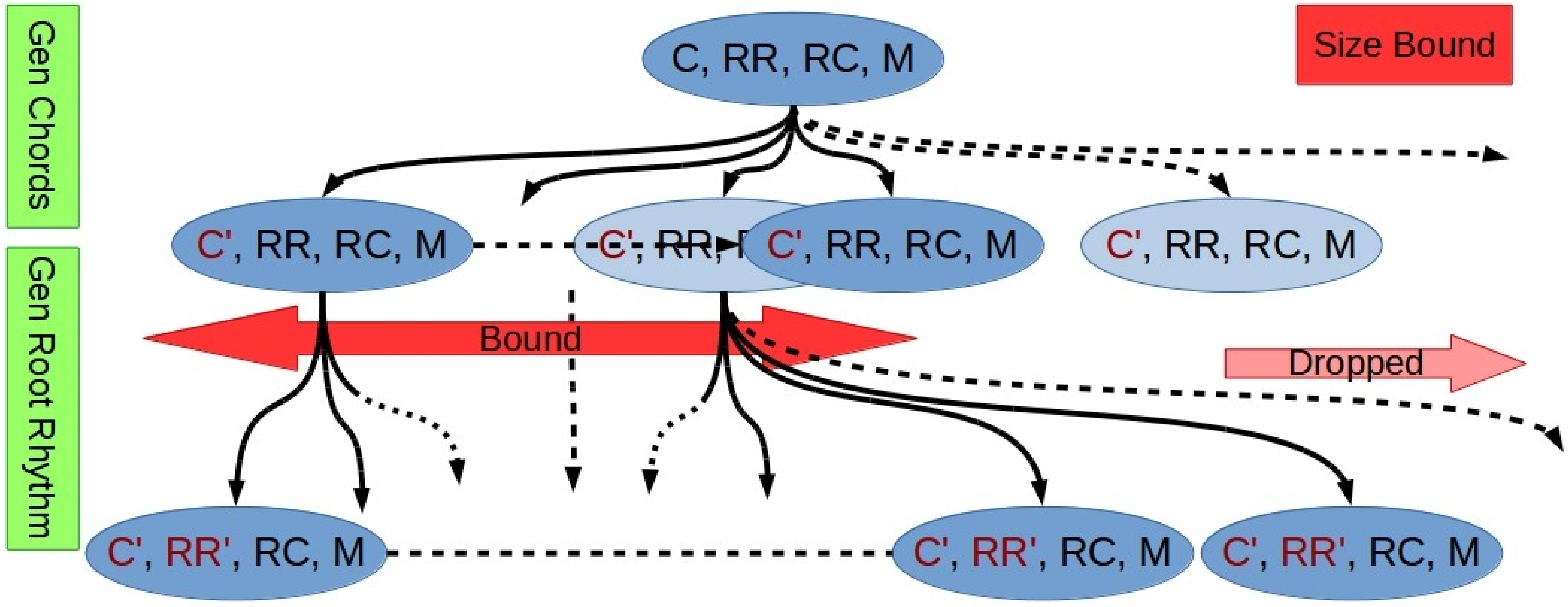
Generation : Chord



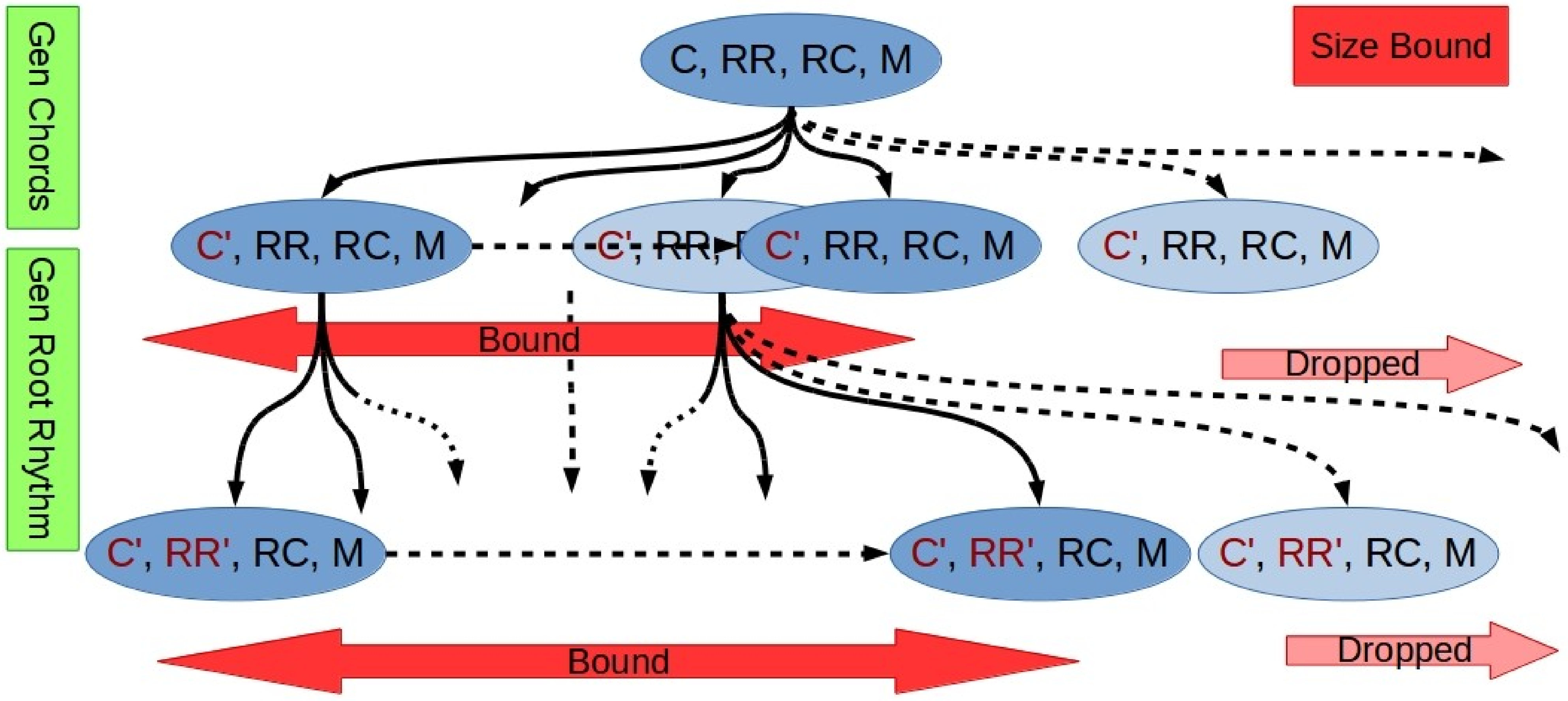
Generation : Bound



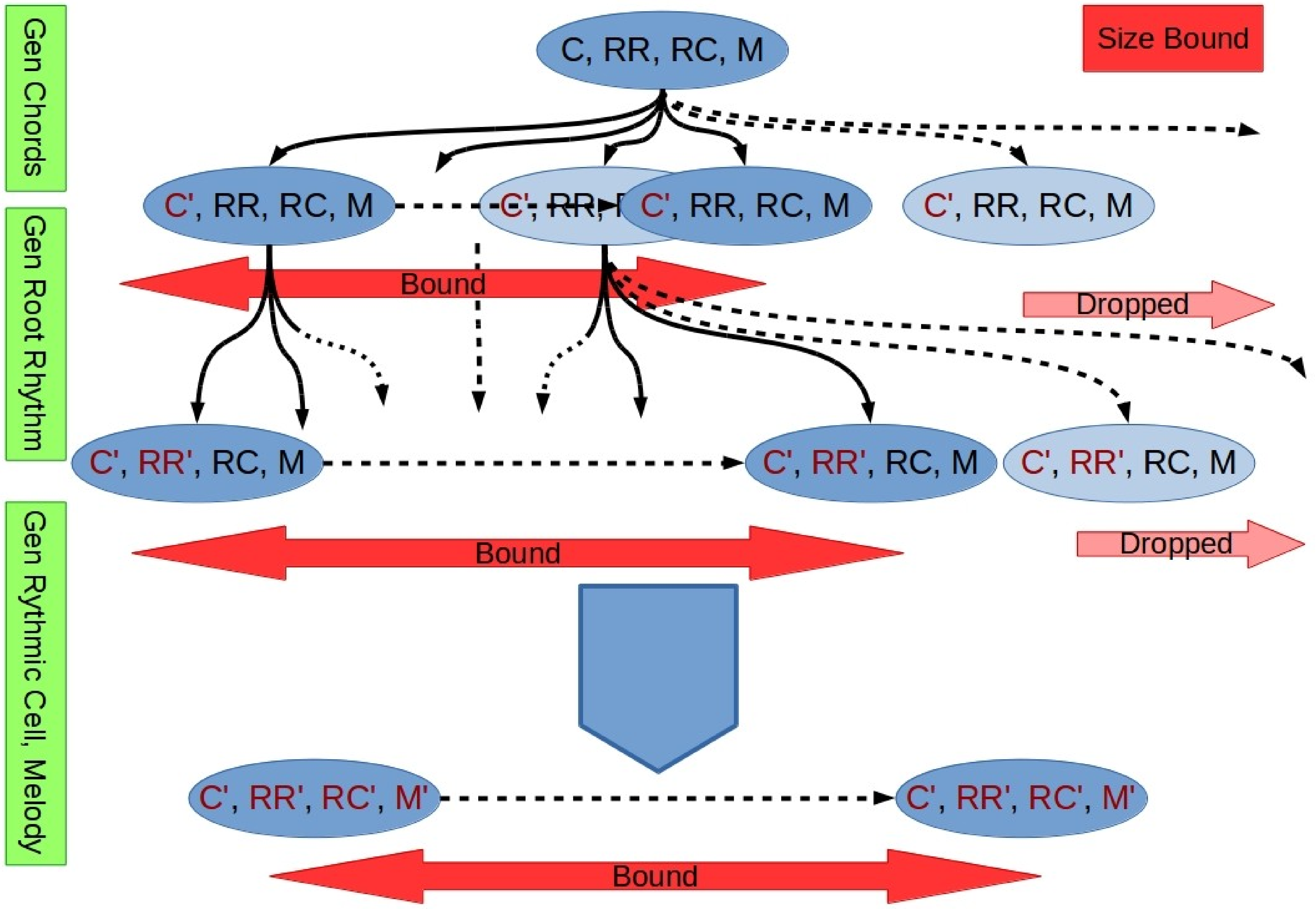
Generation : Root



Generation : Bound



Generation : ...


Gen Chords
Gen Root Rhythm
Gen Rythmic Cell, Melody
Size Bound
C, RR, RC, M
C', RR, RC, M
C', RR, RC, M
C', RR, RC, M
C', RR, RC, M
Bound
Dropped
C', RR', RC, M
C', RR', RC, M
C', RR', RC, M
Bound
Dropped
C', RR', RC', M'
C', RR', RC', M'
Bound

2. Generation

The generation relies on grammar representation and manipulation

1. **Intermediate State** : grammar management
2. **Cell Generation** : the algorithm itself
3. **Closing Conditions** : to stop or not to stop ?
4. **Refinements** : interactive grammars

2. Generation

The generation relies on grammar representation and manipulation

1. **Intermediate State** : grammar management
2. **Cell Generation** : the algorithm itself
3. **Closing Conditions** : to stop or not to stop ?
4. **Refinements** : interactive grammars

3. Example

We first build a stupid-simple example

Then refine it with to get better output

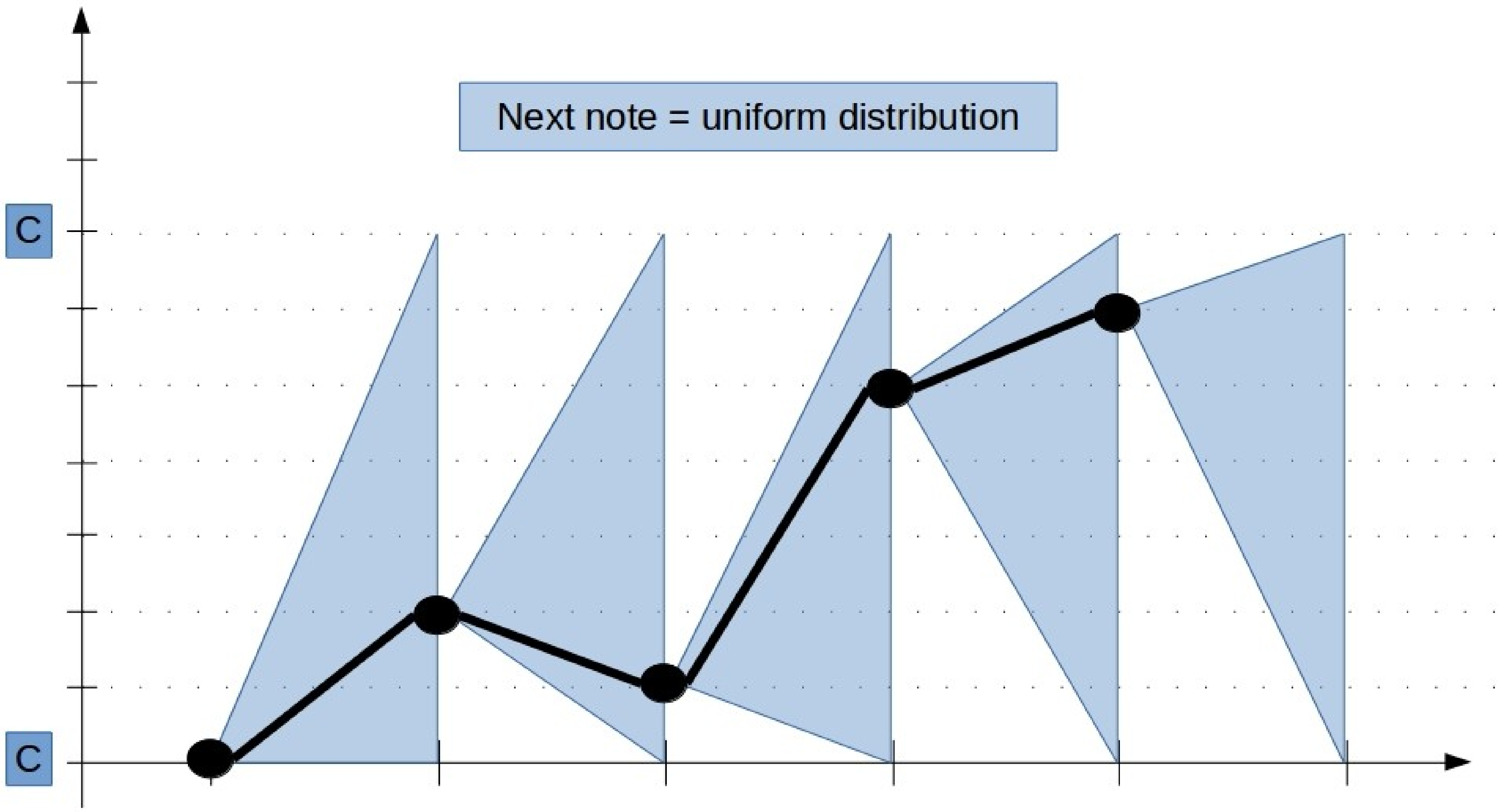
0. **Stupid-simple**
1. **Bound melody**
2. **Melody flow**
3. **Rhythm & Harmony variation**
4. **End control**
5. **End refinement**

3. Example : Notation

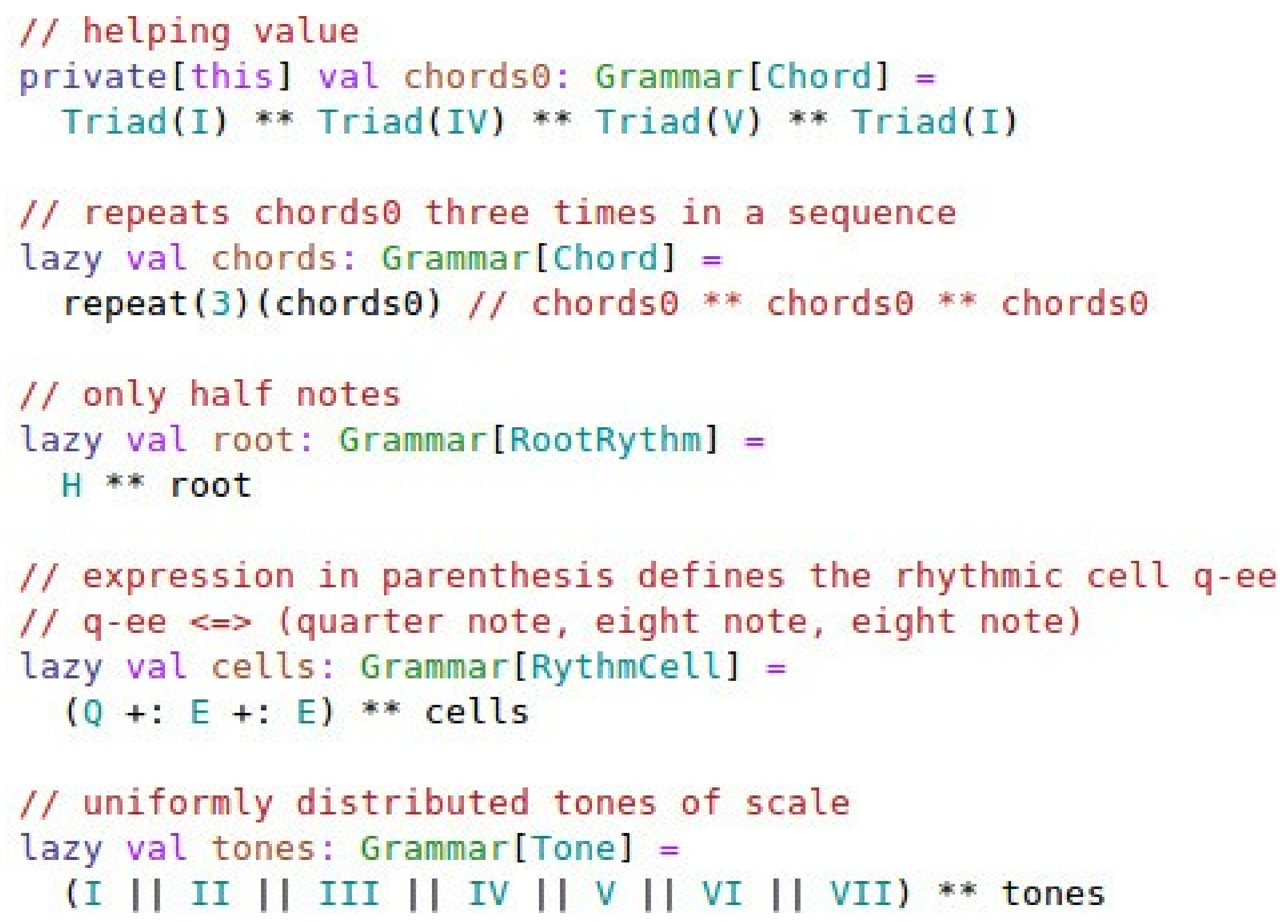
The following notations will be useful when referring to the code

1. R1 ****** R2
2. R1 **||** R2
3. (R1, **1.0**) || (R2, **2.0**)
4. **`def`** `R1(`**`x`**`)` = ...

Tone progression

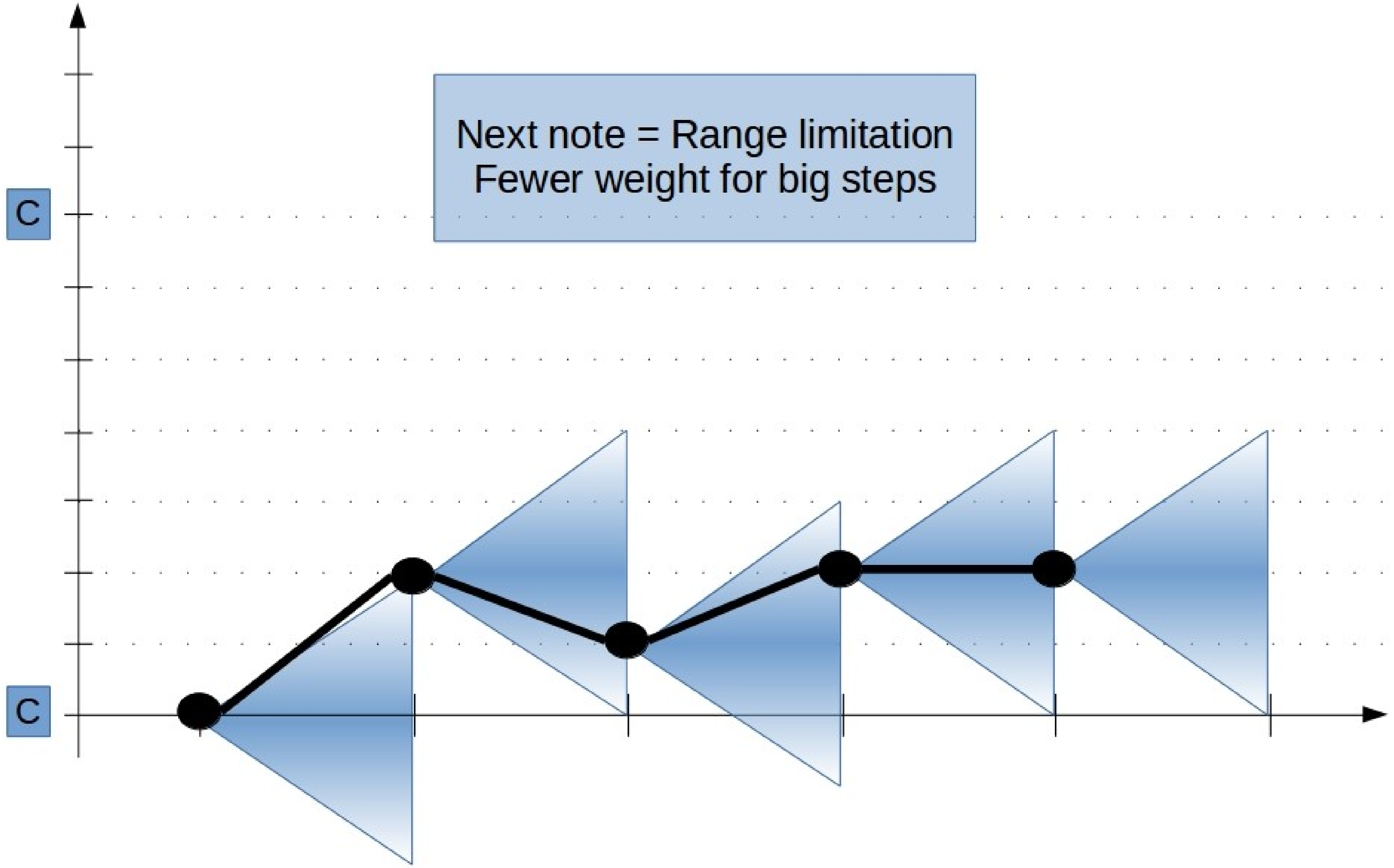


Stupid-simple

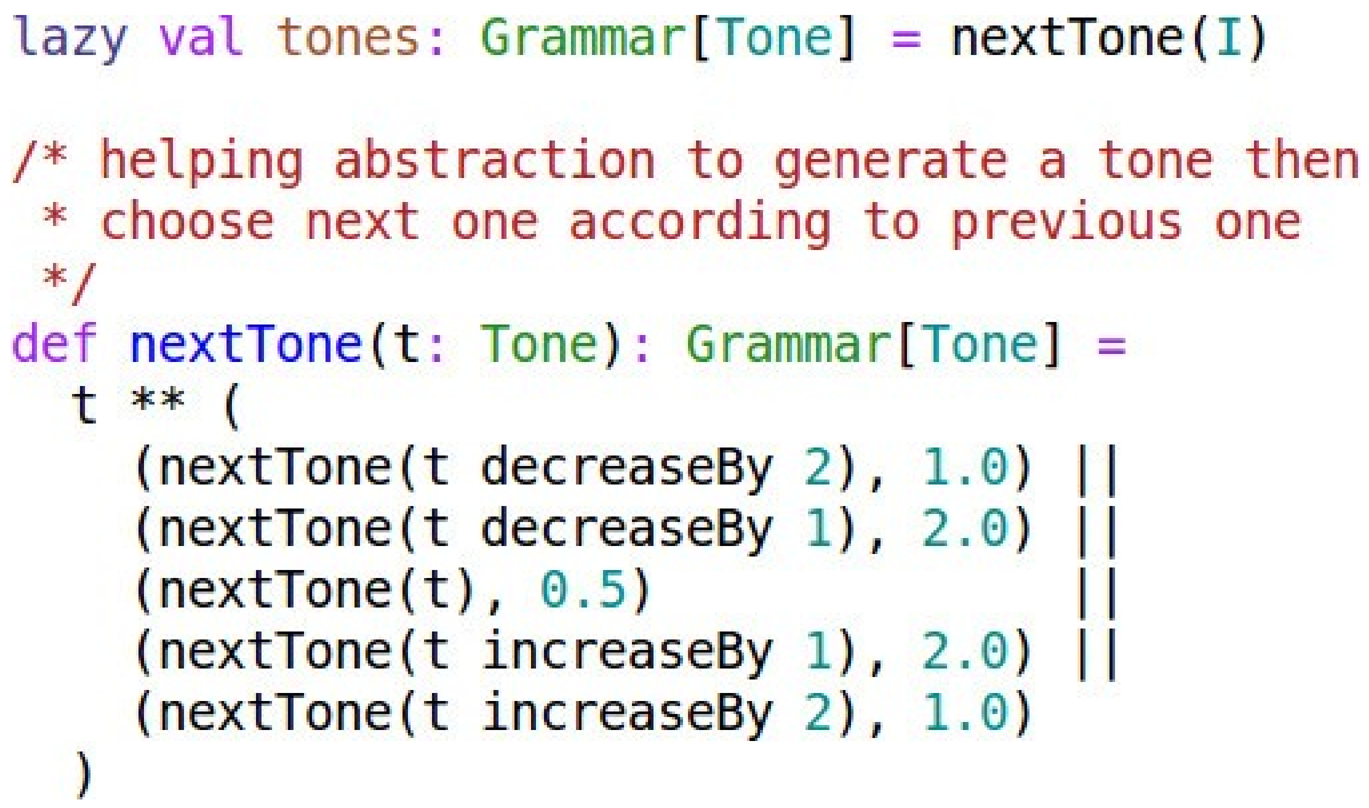


```
// helping value
private[this] val chords0: Grammar[Chord] =
  Triad(I) ** Triad(IV) ** Triad(V) ** Triad(I)

// repeats chords0 three times in a sequence
lazy val chords: Grammar[Chord] =
  repeat(3)(chords0) // chords0 ** chords0 ** chords0

// only half notes
lazy val root: Grammar[RootRythm] =
  H ** root

// expression in parenthesis defines the rhythmic cell q-ee
// q-ee <=> (quarter note, eight note, eight note)
lazy val cells: Grammar[RythmCell] =
  (Q +: E +: E) ** cells

// uniformly distributed tones of scale
lazy val tones: Grammar[Tone] =
  (I || II || III || IV || V || VI || VII) ** tones
```

Bound melody

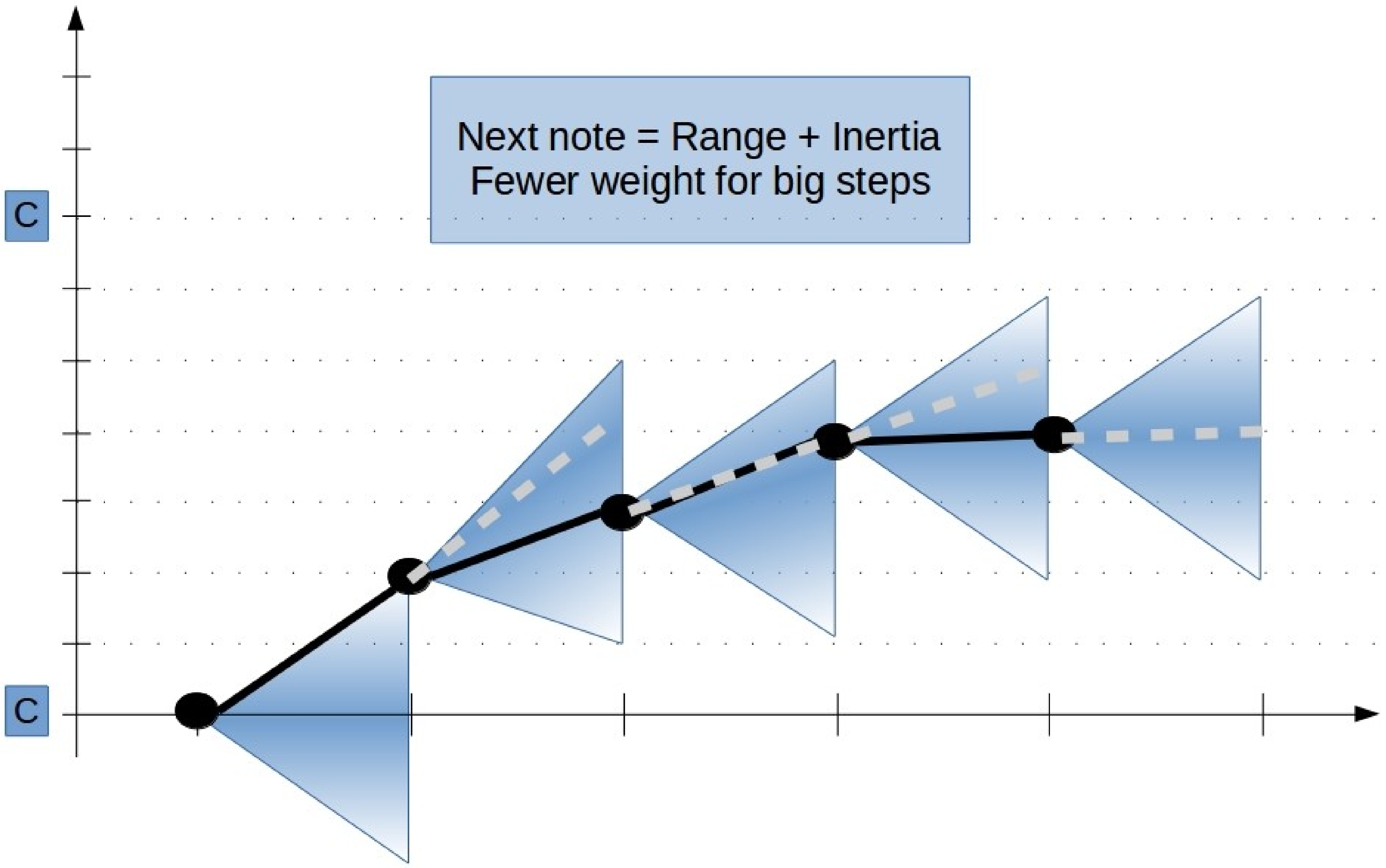


Bound melody



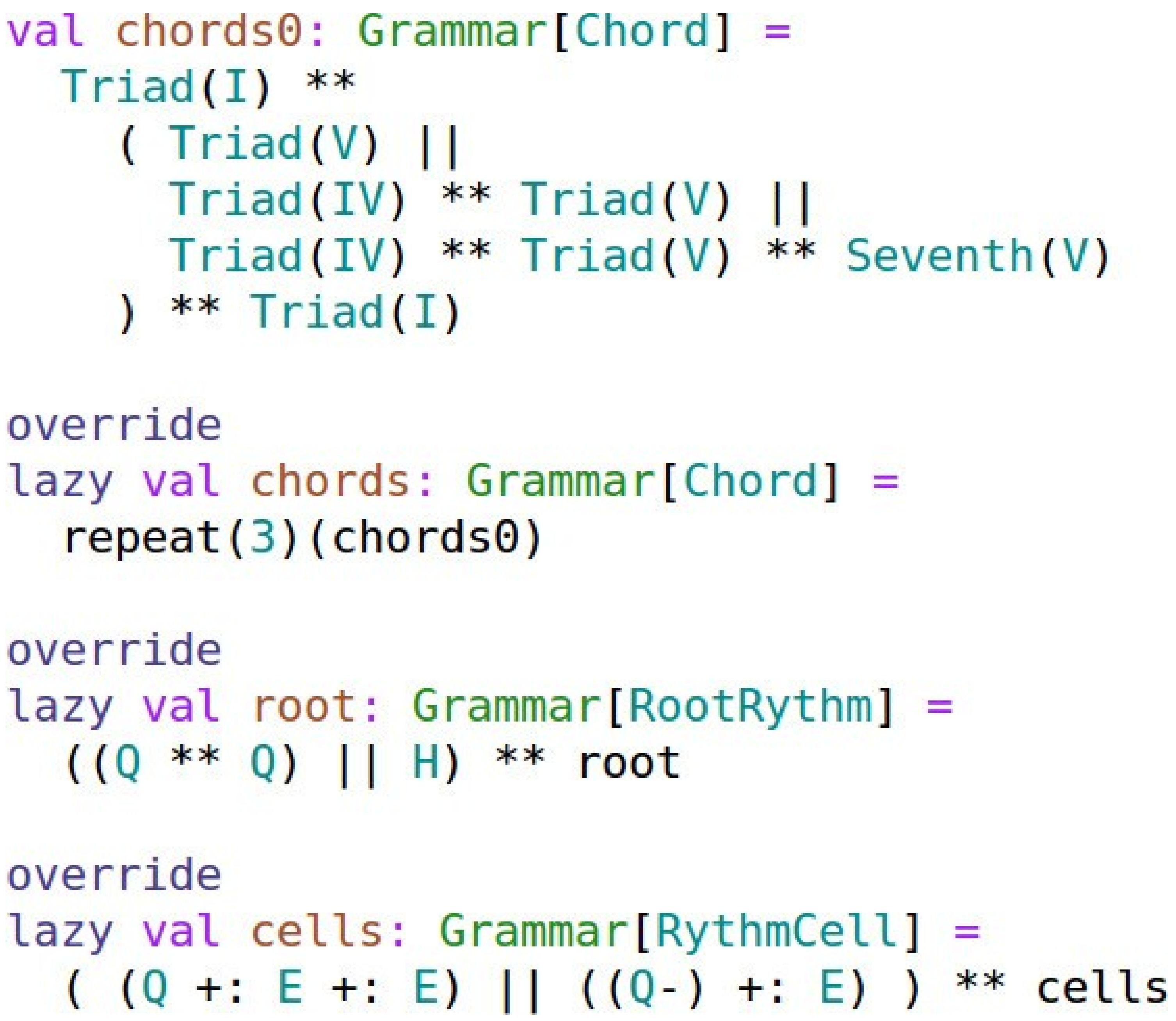
```
lazy val tones: Grammar[Tone] = nextTone(I)

/* helping abstraction to generate a tone then
 * choose next one according to previous one
 */
def nextTone(t: Tone): Grammar[Tone] =
  t ** (
    (nextTone(t decreaseBy 2), 1.0) ||
    (nextTone(t decreaseBy 1), 2.0) ||
    (nextTone(t), 0.5)              ||
    (nextTone(t increaseBy 1), 2.0) ||
    (nextTone(t increaseBy 2), 1.0)
  )
```

Melody flow



Next note = Range + Inertia
Fewer weight for big steps

Variations



```
val chords0: Grammar[Chord] =
  Triad(I) **
    ( Triad(V) ||
      Triad(IV) ** Triad(V) ||
      Triad(IV) ** Triad(V) ** Seventh(V)
    ) ** Triad(I)

override
lazy val chords: Grammar[Chord] =
  repeat(3)(chords0)

override
lazy val root: Grammar[RootRythm] =
  ((Q ** Q) || H) ** root

override
lazy val cells: Grammar[RythmCell] =
  ( (Q +: E +: E) || ((Q-) +: E) ) ** cells
```

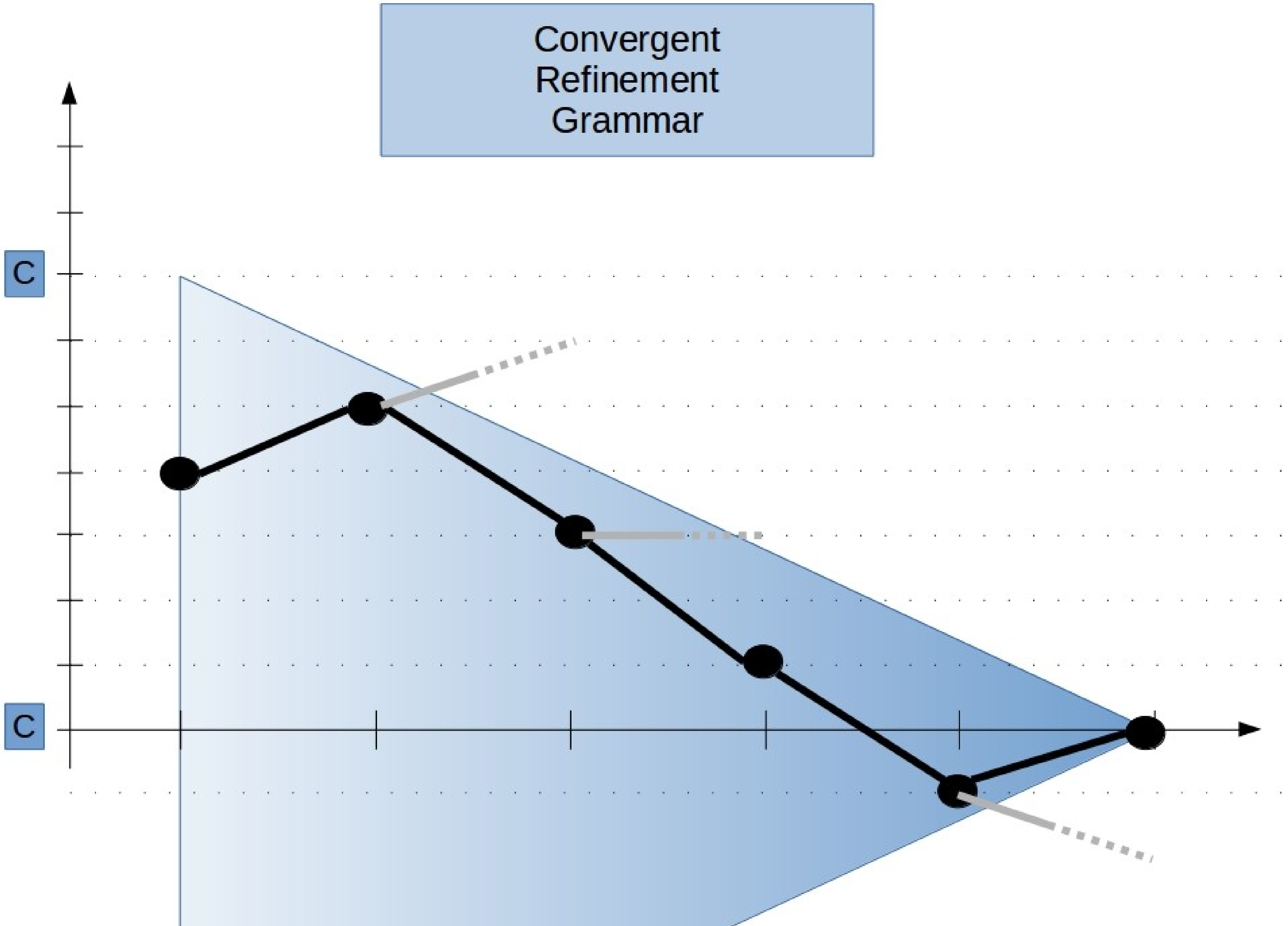
End control

```
/* specifies closing contition :
 * every grammar must be finished simultaneously
 */
override val closeRoot = true
override val closeCells = true
override val closeTones = true

// chords are inchanged since the grammar "decides" the end

override
lazy val root: Grammar[RootRythm] =
  (rootBody ** root) || rootEnd

val rootBody: Grammar[RootRythm] = (Q ** Q) || H
val rootEnd: Grammar[RootRythm] = H
```

End control

```
override
lazy val cells: Grammar[RythmCell] =
  (cellsBody ** cells) || cellsEnd

val cellsBody: Grammar[RythmCell] = (Q +: E +: E) || ((Q-) +: E)
val cellsEnd: Grammar[RythmCell] = RythmCell(H::Nil)


/* recall that inertialNext defined in InertialMelody is already
 * recursive and nullable
 */
override
lazy val tones = inertialNext(I) ** tonesEnd

val tonesEnd: Grammar[Tone] = I
```

Refinement



Refinement

```
// recall that chords0 was defined in Variations
override
lazy val chords: Grammar[Chord] =
  ( repeat(3)(chords0) **                        // beginning
    MelodyRefine[Chord](converge(10)) ** // near the end, send refinement message
    repeat(2)(chords0))                          // end


// creates a n infinite grammar that converges in n steps
def converge(n: Int): Grammar[Tone] =
  // still allows some oscillation after converged
  if (n < 2) converge(2)
  else {
    Production((for (i <- -n to n) yield (Word(I increaseBy i), 1.0)).toList) **
    converge(n-1)
  }
```

And there is a last one ...

Thank you for your attention !