



Declarative Music: Using affix grammars to compose music

Lambert Meertens

Declarative Amsterdam

8 November 2024



Intro

The year is 1962.

Kees Koster and I are undergraduate students at the University of Amsterdam.

Instead of going to the boring lectures we are supposed to follow, we attend graduate seminars, such as a seminar “Machine and Language”, where we read Chomsky’s book *Syntactic Structures*.

Context-Free Grammar (CFG)

In his book, Chomsky defines “Phrase Structure Grammars”, a family of grammar formalisms intended as a tool to describe natural languages. In their simplest form these are the so-called context-free grammars. To most participants of the seminar this was something new.

Kees and I recognize **CFG** as something we are familiar with: **BNF**, the grammar formalism used in the *Report on the Algorithmic Language ALGOL 60* to define a programming language.

Chomsky (1957):

Sentence \rightarrow *NP* + *VP*

NP \rightarrow *T* + *N*

VP \rightarrow *Verb* + *NP*

T \rightarrow *the*

N \rightarrow *man, ball, etc.*

Verb \rightarrow *hit, took, etc.*

ALGOL 60 report (1960):

```
<assignment statement> ::=
    <left part list><arithmetic expression> |
    <left part list><Boolean expression>
<left part list> ::=
    <left part> | <left part list><left part>
<left part> ::= <variable> :=
```



Back to 1962

Kees and I take it upon ourselves to give a demo of the use of a CFG to generate grammatically well-formed non-trivial English sentences.

Kees is to write the program (for the Electrologica X1 of the Mathematical Centre, now CWI).

My role is to supply the grammar.

Turning a grammar into a generator

SENT \rightarrow NP *are* COLOUR

NP \rightarrow *roses* | *violets* | *lemons*

COLOUR \rightarrow *red* | *blue* | *yellow*

```
def sent():
```

```
    np()
```

```
    print(' are')
```

```
    colour()
```

```
def colour():
```

```
    cc = ['red', 'blue', 'yellow']
```

```
    c = random.choice(cc)
```

```
    print(' ' + c)
```

First attempt

SENT → NP VERB | NP VERB NP

NP → *the boy* | *mice* | *grains* | ...

VERB → *falls* | *like* | ...

✓ *the boy falls*

✓ *mice like grains*

✗ *the boy like mice*

✗ *mice falls*

**These sentences fail
subject–verb agreement**

Second attempt

SENT →

NP_{sg} VERB_{sg} | NP_{sg} VERB_{sg} NP_{sg} | NP_{sg} VERB_{sg} NP_{pl} |
NP_{pl} VERB_{pl} | NP_{pl} VERB_{pl} NP_{sg} | NP_{pl} VERB_{pl} NP_{pl}

NP_{sg} → *the boy* | ...

NP_{pl} → *mice* | *grains* | ...

VERB_{sg} → *falls* | ...

VERB_{pl} → *like* | ...

✗ *grains like*

✗ *mice falls the boy*

**These sentences fail
transitivity agreement**

Agreement in English grammar

- number agreement between subject and verb
- transitivity agreement between verb and object
- person agreement between subject and verb
- case agreement of pronoun with grammar role

To take account of these requirements I would have to replace the rules by multiple copies, and then multiple copies of these multiple copies, and multiple copies of multiple copies of multiple copies, a horrible *combinatorial explosion!*



Let the computer do the work!

Use a shorthand notation:

SENT \rightarrow NP_N VERB_N | NP_N VERB_N NP_{N'}

N, N' \rightarrow sg, pl

 **affix rule**

Write a program to expand this into:

SENT \rightarrow

NP_{sg} VERB_{sg} | NP_{sg} VERB_{sg} NP_{sg} | NP_{sg} VERB_{sg} NP_{pl} |
NP_{pl} VERB_{pl} | NP_{pl} VERB_{pl} NP_{sg} | NP_{pl} VERB_{pl} NP_{pl}



Fuse expansion with generation

Do not consider this notation to be shorthand:

$$\text{SENT} \rightarrow \text{NP}_N \text{ VERB}_N \mid \text{NP}_N \text{ VERB}_N \text{ NP}_{N'}$$

$N, N' \rightarrow \text{sg, pl}$

Instead, view it as a new type of grammar in its own right, a two-level grammar.

Thus, affix grammars were born.

(We finished the project in time for the demo, which went smoothly, without a glitch.)

Affix grammars

An affix grammar has *two levels* of rules.

- The second level consists of affix rules, which form a grammar for the affixes.
- The ground level has rule *schemas*, which are like normal CFG rules, but nonterminal symbols may have nonterminal affixes. These schemas are turned into normal rules by the *systematic* replacement of nonterminal affixes by terminal affixes (such as, either replace each N in a given schema by 'sg' or replace each N by 'pl').

The IFIP Competition

COMPUTER-COMPOSED MUSIC — COMPETITION FOR 1968

International Federation for Information Processing
IFIP Congress Office
23 Dorset Square
London, N.W. 1, England

The International Federation for Information Processing (IFIP) has organised a computer-composed music competition in connection with IFIP Congress 68. Entries submitted for the competition must be produced entirely by the agency of a computer and form an artistic whole. Entries will be judged on musical merit, and medals will be awarded for the best three pieces of music composed by computer. It is hoped that the prizewinning entries will be performed during IFIP Congress 68. The Congress is to be held in Edinburgh from August 5th to 10th, 1968.

Rules

1. Entries may be submitted either by individuals or jointly, by groups of people. Each entry must be accompanied by a statement signed by every member of the group (or by the individual) certifying that to their (his) knowledge all the computer programming — excluding general service routines — was executed by the group (or by the individual).
2. The performing time for an entry should be not less than three minutes and not more than 15 minutes.
3. Entries may be submitted in any of the following forms:
 - a) a score for a string quartet.
 - b) a recording accompanied by a score (where other instruments are involved)
 - c) a recording without a score (if the sound is produced by a computer)

Computers & Automation,
May 1967



A procedural grammar for rhythm

Affixes are modeled as procedure arguments:

```
def rhythm(duration):  
  if unbroken(duration):      # random decision  
    tick(duration)  
  else:  
    rhythm(halve(duration))  
    rhythm(duration – halve(duration))
```

A procedural grammar for melody

```
def melody(duration, height):  
    if unbroken(duration):  
        tone(height, duration)  
    else:  
        if ascending( ):           # random decision  
            melody(halve(duration), height - 1)  
            melody(duration - halve(duration), height)  
        else:  
            melody(halve(duration), height)  
            melody(duration - halve(duration), height - 1)
```

In ALGOL 60

```
procedure compose (melodic voice, num beats, left function code, right function code,  
  steady function, cadence, bars to go, constant, new start, figurating, melos, rhythm, others,  
  max height, left branch, beat strength, right beat strength);  
value melodic voice, num beats, left function code, right function code, steady function, cadence,  
  bars to go, constant, new start, figurating, melos, rhythm, others, max height, left branch,  
  beat strength, right beat strength;  
integer melodic voice, num beats, left function code, right function code, bars to go,  
  beat strength, right beat strength;  
real melos, rhythm, others, max height;  
Boolean steady function, cadence, figurating, left branch;  
Boolean array constant, new start; comment [soprano : bass];  
begin integer voice, bass voice, voice 2, round, left right function code, right left function code;  
  Boolean array split up[soprano : bass]; Boolean some split up, last of cadence;  
  real left melos, right melos, left rhythm, right rhythm;  
  
  bass voice := if melodic voice = bass then tenor else bass;  
  round := (left function code - right function code) / 3 + (num beats - 1) / mean cycle beats;  
  right function code := right function code + 3 × round;  
  if right function code < left function code then right function code := right function code + 3;  
  left right function code := (left function code + right function code) ÷ 2;  
  if cadence ∧ num beats = bars to go × bar beats ∧ bars to go ≥ 9 then  
    begin right left function code := right function code - bars to go ÷ 2 + 1;  
      maxim (left right function code, right left function code - 1)  
    end else  
    right left function code := (left function code - left right function code +  
      right function code);  
  
  last of cadence ∧ bars to go = 1;
```



The result, String Quartet No. 1 in C

Allegro I

Violin I
Violin II
Viola
Violoncello

page 1

The image shows a page of handwritten musical notation for a string quartet. The score is written for four instruments: Violin I, Violin II, Viola, and Violoncello. The tempo is marked 'Allegro' and the movement is 'I'. The music is in 3/4 time and the key signature is C major. The notation includes various rhythmic values such as eighth and sixteenth notes, and rests. The page is numbered 'page 1' in the top right corner. The score is presented on a white background with a decorative border on the left side.



A longer account, with more emphasis on the musical aspects, is given in:

- Lambert Meertens. An Early Experiment in Algorithmic Composition. In: Gerard Alberts, Jan Friso Groote, editors, *Tales of Electrologica*. History of Computing. Springer, 2022

The score of the string quartet is available as:

- Lambert Meertens. *Quartet No. 1 in C Major for 2 Violins, Viola and Violoncello*. Mathematical Centre Report MR 96. Mathematisch Centrum, Amsterdam, 1968 (<https://ir.cwi.nl/pub/9184>)

The same web page also has links to four mp3 files, together a full performance from 1968 by the Amsterdam String Quartet.