Department of Computer Science
Technical University of Cluj-Napoca

Intelligent Systems

Laboratory activity 2018-2019

Project title: Answer Song Programming
Tool: Potassco, the Potsdam Answer Set Solving Collection

Name: Hristache Alexandru
Group: 30431
Email: hristache.alex@gmail.com

Contents

Chapter 1

Introduction to Answer Set Programming and Potassco

1.1 Answer Set Programming

Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. As an outgrowth of research on the use of nonmonotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications. ASP is based on the stable model (answer set) semantics of logic programming (Gelfond and Lifschitz 1988), which applies ideas of autoepistemic logic (Moore 1985) and default logic (Reiter 1980) to the analysis of negation as failure.

In ASP, search problems are reduced to computing stable models, and answer set solvers—programs for generating stable models—are used to perform search. The search algorithms used in the design of many answer set solvers are enhancements of the Davis-Putnam-Logemann-Loveland procedure, and they are somewhat similar to the algorithms used in efficient SAT solvers (Gomes et al. 2008).2 Unlike SLDNF resolution employed in Prolog, such algorithms, in principle, always terminate.

The ASP methodology is about ten years old. The planning method proposed by Dimopoulos, Nebel and Koehler (1997) is an early example of answer set programming.

Their approach is based on the relationship between plans and stable models described in (Subrahmanian and Zaniolo 1995). Soininen and Niemela (1998) applied what is now known as answer set programming to the problem of product configuration. The use of answer set solvers for search was identified as a new programming paradigm in (Marek and Truszczy'nski 1999) (the term "answer set programming" was used for the first time as the title of a part of the collection where that paper appeared) and in (Niemela 1999).

1.2 Potassco Tool Description

The "Potsdam Answer Set Solving Collection" gathers a variety of tools for Answer Set Programming, including grounder gringo, solver clasp, and their combination within the integrated ASP system clingo. Their common goal is to enable users to rapidly solve computationally difficult problems in ASP, a declarative programming paradigm based on logic programs and their answer sets.

1.3 Potassco Popular Projects

- ASPeRiX (University of Angers, France)

  The program ASPeRiX is an implementation of the stable model semantics for normal logic programs. The main specifity of our system is to realize a forward chaining of first order rules that are grounded on the fly. So, unlike others available ASP systems, ASPeRiX does not need a pregrounding processing.

- ASP Tools (Aalto University, Finland)

  A collection of software that has been developed for Answer Set Programming (ASP) purposes by the Computational Logic Group at the ICS Department of Aalto SCI.

- Cmodels (University of Texas at Austin, USA)

  Cmodels is a system that computes answer sets for either disjunctive logic programs or logic programs containing choice rules. Answer set solver Cmodels uses SAT solvers as a search engine for enumerating models of the logic program.

- DLV (University of Calabria and the Vienna University of Technology, Italy and Austria)

  DLV is a deductive database system, based on disjunctive logic programming, which offers front-ends to several advanced KR formalisms.

- GASP (University of Udine, Italy)

  GASP is a lazy grounding based interpreter for ASP.

- IDP (University of Leuven, Belgium)

  The IDP system is a knowledge base system that implements different forms of inference for a knowledge base consisting of FO(.) theories.

- SeaLion (Vienna University of Technology, Austria)

  SeaLion is an integrated development environment for ASP. It comes as an Eclipse plugin and aims at helping the programmer to write, evaluate, debug, and test answer-set programs. SeaLion supports the languages of the Potassco solver family (clasp/clingo) and other solvers.

- smodels+lparse (Helsinki University of Technology, Finland)

  The program smodels is an implementation of the stable model semantics for logic programs. Lparse is a front-end for smodels that generates a variable-free simple logic program that can be given to smodels.

- pyzcasp

  This package provides a python framework to build on top of Answer Set Programming tools using the Zope Component Architecture. Currently pyzcasp supports the usage of most common tools from Potassco.

Chapter 2

Installing the tool

The Potassco tools gringo, clasp, and clingo are written in C++ and published under the GNU General Public License. Source packages as well as precompiled binaries for Linux, MacOS, and Windows are available at. For building the tools from sources, please download the most recent source package, consult the included `README` file, and make sure that the machine to build on has all required software installed. If you still encounter problems in the building process, please consult the support pages at or use the Potassco mailing list: potassco-users@lists.sourceforge.net.

An alternative way to install the tools is to use a package manager. Currently, packages and ports are available for Debian, Ubuntu, Arch Linux (AUR), and for MacOS X (via Homebrew or MacPorts). Note that packages installed this way are not always up to date; the latest versions are available at our Sourceforge page at.

Afterward, one can check whether everything works fine by invoking the tool with flag –version (to get version information) or with flag –help (to see the available command line options). For instance, assuming that a binary called gringo is in the path (similarly with the other tools), you can invoke the following two commands:

```
gringo --version
gringo --help
```

Note that gringo, clasp, and clingo run on the command line (Linux shell, Windows command prompt, or the like). To invoke them, their binaries can be "installed" simply by putting them into some directory in the system path. In an invocation, one usually provides the file names of input (text) files as arguments to either gringo or clingo, while the output of gringo is typically piped into clasp. Thus, the standard invocation schemes are as follows:

```
gringo [ options | files ] | clasp [ options | number ]
clingo [ options | files | number ]
```

A numerical argument provided to either clasp or clingo determines the maximum number of answer sets to be computed, where 0 means "compute all answer sets". By default, only one answer set is computed (if it exists).

Chapter 3

Running and understanding examples

As an introductory example, we consider a simple Towers of Hanoi puzzle, consisting of three pegs and four disks of different size. The goal is to move all disks from the left peg to the right one, where only the topmost disk of a peg can be moved at a time. Furthermore, a disk cannot be moved to a peg already containing some disk that is smaller. Although there is an efficient algorithm to solve our simple Towers of Hanoi puzzle, we do not exploit it and below merely specify conditions for sequences of moves being solutions.

In ASP, it is custom to provide a *uniform* problem definition. Following this methodology, we separately specify an instance and an encoding (applying to every instance) of the following problem: given an initial placement of the disks, a goal situation, and a number n, decide whether there is a sequence of n moves that achieves the goal. We will see that this problem can be elegantly described in ASP and solved by domain-independent tools like and . Such a declarative solution is now exemplified.

3.1 Problem Instance

We describe the pegs and disks of a Towers of Hanoi puzzle via facts over the predicates `peg/1` and `disk/1` (the number denotes the arity of the predicate). Disks are numbered by consecutive integers starting at 1, where a disk with a smaller number is considered to be bigger than a disk with a greater number. The names of the pegs can be arbitrary; in our case, we use a, b and c. Furthermore, the predicates `init_on/2` and `on/2` describe the initial and the goal situation, respectively. Their arguments, the number of a disk and the name of a peg, determine the location of a disk in the respective situation. Finally, the predicate `moves/1` specifies the number of moves in which the goal must be achieved. When allowing 15 moves, the Towers of Hanoi puzzle is described by the following facts:

```
peg(a;b;c).
disk(1..4).
init_on(1..4,a).
goal_on(1..4,c).
moves(15).
```

Note that the ';' in the first line is syntactic sugar that expands the statement into three facts: `peg(a).,peg(b)., and peg(c)`. Similarly, `'1..4'` refers to an interval. Here, it abbreviates distinct facts over four values: `1, 2, 3, and 4`. In summary, the facts describe the Towers of Hanoi puzzle along with the requirement that the goal ought to be achieved within 15 moves.

3.2 Problem Encoding

We now proceed by encoding Towers of Hanoi via schematic rules, i.e., rules containing variables (whose names start with uppercase letters) that are independent of a particular instance. Typically, an encoding can be logically partitioned into a *Generate*, a *Define*, and a *Test* part. An additional *Display* part allows for restricting the output to a distinguished set of atoms, and thus, for suppressing auxiliary predicates.

We follow this methodology and mark the respective parts via comment lines beginning with '%' in the following encoding:

```
 1  % Generate
 2  { move(D,P,T) : disk(D), peg(P) } = 1 :- moves(M),
       T = 1..M.
 3  % Define
 4  move(D,T)    :- move(D,_,T).
 5  on(D,P,0)    :- init_on(D,P).
 6  on(D,P,T)    :- move(D,P,T).
 7  on(D,P,T+1) :- on(D,P,T), not move(D,T+1),
       not moves(T).
 8  blocked(D-1,P,T+1) :- on(D,P,T), not moves(T).
 9  blocked(D-1,P,T)   :- blocked(D,P,T), disk(D).
10  % Test
11  :- move(D,P,T), blocked(D-1,P,T).
12  :- move(D,T), on(D,P,T-1), blocked(D,P,T).
13  :- goal_on(D,P), not on(D,P,M), moves(M).
14  :- { on(D,P,T) } != 1, disk(D), moves(M), T = 1..M.
15  % Display
16  #show move/3.
```

Note that the variables `D, P, T, M` are used to refer to disks, pegs, the number of a move, and the length of the sequence of moves, respectively.

The Generate part, describing solution candidates, consists of the rule in Line 2. It expresses that, at each point `T` in time (other than `0`), exactly one move of a disk `D` to some peg `P` must be executed. The head of the rule (left of `:-`) is a so-called cardinality constraint. It consists of a set of literals, expanded using the conditions behind the colon, along with the guard '`= 1`'. The cardinality constraint is satisfied if the number of true elements is equal to one, as specified by the guard. Since the cardinality constraint occurs as the head of a rule, it allows for deriving ("guessing") atoms over the predicate `move/3` to be true. In the body (right of '`:-`'), we define, `T = 1..M`, to refer to each time point `T` from `1` to the maximum time point `M`. We have thus characterized all sequences of `M` moves as solution candidates for Towers of Hanoi. Up to now, we have not yet imposed any further conditions, e.g., that a bigger disk must not be moved on top of a smaller one.

The Define part in Line 4–9 contains rules defining auxiliary predicates, i.e., predicates that provide properties of a solution candidate at hand. (Such properties will be investigated in the Test part described below.) The rule in Line 4 simply projects moves to disks and time points. The resulting predicate `move/2` can be used whenever the target peg is insignificant, so that one of its atoms actually subsumes three possible cases. Furthermore, the predicate `on/3` captures the state of a Towers of Hanoi puzzle at each time point. To this end, the rule in Line 5 identifies the locations of disks at time point `0` with the initial state (given in an instance). State transitions are modeled by the rules in Line 6 and 7. While the former specifies the direct effect of a move at time point `T`, i.e., the affected disk `D` is relocated to the target peg `P`, the latter describes inertia: the location of a disk `D` carries forward from time point `T` to `T+1` if D is not moved at `T+1`. Observe the usage of `not moves(T)` in Line 7, which prevents deriving disk locations beyond the maximum time point. Finally, we define the auxiliary predicate `blocked/3` to indicate that a smaller disk, with a number greater than `D-1` is located on a peg `P`. The rule in Line 8 derives this condition for time point `T+1` from `on(D, P, T)`, provided that `T` is not the maximum time point. The rule in Line 9 further propagates the status of being blocked to all bigger disks on the same peg. Note that we also mark `D - 1 = 0`, not referring to any disk, as blocked, which is convenient for eliminating redundant moves in the Test part described next.

The Test part consists of the integrity constraints in Line 11–14, rules that eliminate unintended solution candidates. The first integrity constraint in Line 11 asserts that a disk `D` must not be moved to a peg `P` if `D-1` is blocked at time point `T` This excludes moves putting a bigger disk on top of a smaller one and, in view of the definition of `blocked/3` also disallows that a disk is put back to its previous location. Similarly, the integrity constraint in Line 12 expresses that a disk `D` cannot be moved at time point `T` if it is blocked by some smaller disk on the same peg `P`. Note that we use `move(D, T)` here because the target of an illegal move does not matter in this context. The fact that the goal situation, given in an instance, must be achieved at maximum time point `M` is represented by the integrity constraint in Line 13. The final integrity constraint in Line 14 asserts that, for every disk `D` and time point `T`, there is exactly one peg `P` such that `on(D, P, T)` holds. Although this condition is implied by the definition of `on/3`in Line 6 and 7 with respect to the moves in a solution, making such knowledge explicit via an integrity constraint turns out to improve the solving efficiency.

Finally, the meta-statement of the Display part in Line 16 indicates that only atoms over the predicate move/3 ought to be printed, thus suppressing the predicates used to describe an instance as well as the auxiliary predicates `move/2`, `on/3`, and `blocked/3`. This is for more convenient reading of a solution, given that it is fully determined by atoms over `move/3`.

3.3 Problem Solution

We are now ready to solve our Towers of Hanoi puzzle. To compute an answer set representing a solution, invoke one of the following commands:

```
    clingo toh_ins.lp toh_enc.lp
    gringo toh_ins.lp toh_enc.lp | clasp
```

The output of the solver, in this case, should look somehow like this:

```
clingo version 4.4.0
Reading from toh_ins.lp ...
Solving...
Answer: 1
move(4,b,1) move(3,c,2) move(4,c,3) move(2,b,4) \
move(4,a,5) move(3,b,6) move(4,b,7) move(1,c,8) \
move(4,c,9) move(3,a,10) move(4,a,11) move(2,c,12) \
move(4,b,13) move(3,c,14) move(4,c,15)
SATISFIABLE

Models    : 1+
Calls     : 1
Time      : 0.017s (Solving: 0.01s 1st Model: 0.01s \
                                Unsat: 0.00s)
CPU Time  : 0.010s
```

The first line shows the clingo version. The following two lines indicate clingo's state. clingo should print immediately that it is reading. Once this is done, it prints `Solving...` to the command line. The Towers of Hanoi instance above is so easy to solve that you will not recognize the delay, but for larger problems it can be noticeable. The line starting with `Answer:` indicates that the (output) atoms of an answer set follow in the next line. In this example, it contains the true instances of `move/3` in the order of time points, so that we can easily read off the following solution from them: first move disk 4 to peg b, second move disk 3 to peg c, third move disk 4 to peg c, and so on. We use '\' to indicate that all atoms over `move/3` actually belong to a single line. Note that the order in which atoms are printed does not bear any meaning (and the same applies to the order in which answer sets are found). Below this solution, we find the satisfiability status of the problem, which is reported as SATISFIABLE by the solver.[1] The '1+' in the line starting with Models tells us that one answer set has been found.[2] Calls to the solver are of interest in multi-shot solving. The final lines report statistics including total run-time (wall-clock Time as well as CPU Time) and the amount of time spent on search (Solving), along with the fractions taken to find the first solution (1st Model) and to prove unsatisfiability[3] (Unsat).

[1]Other possibilities include UNSATISFIABLE and UNKNOWN, the latter in case of an abort.

[2]The '+' indicates that the solver has not exhaustively explored the search space (but stopped upon finding an answer set), so that further answer sets may exist.

[3] No unsatisfiability proof is done here, hence, this time is zero. But for example, when enumerating all models, this is the time spent between finding the last model and termination.

Chapter 4

Gringo and Clingo

The tool gringo is a grounder capable of transforming user-defined logic programs (usually containing variables) into equivalent ground (that is, variable-free) programs. The output of gringo can be piped into solver clasp, which then computes answer sets. System clingo internally couples gringo and clasp, thus, it takes care of both grounding and solving. In contrast to gringo outputting ground programs, clingo returns answer sets.

Usually, logic programs are specified in one or more (text) files whose names are provided as arguments in an invocation of either gringo or clingo. In what follows, we describe the constructs belonging to the input language of gringo and clingo.

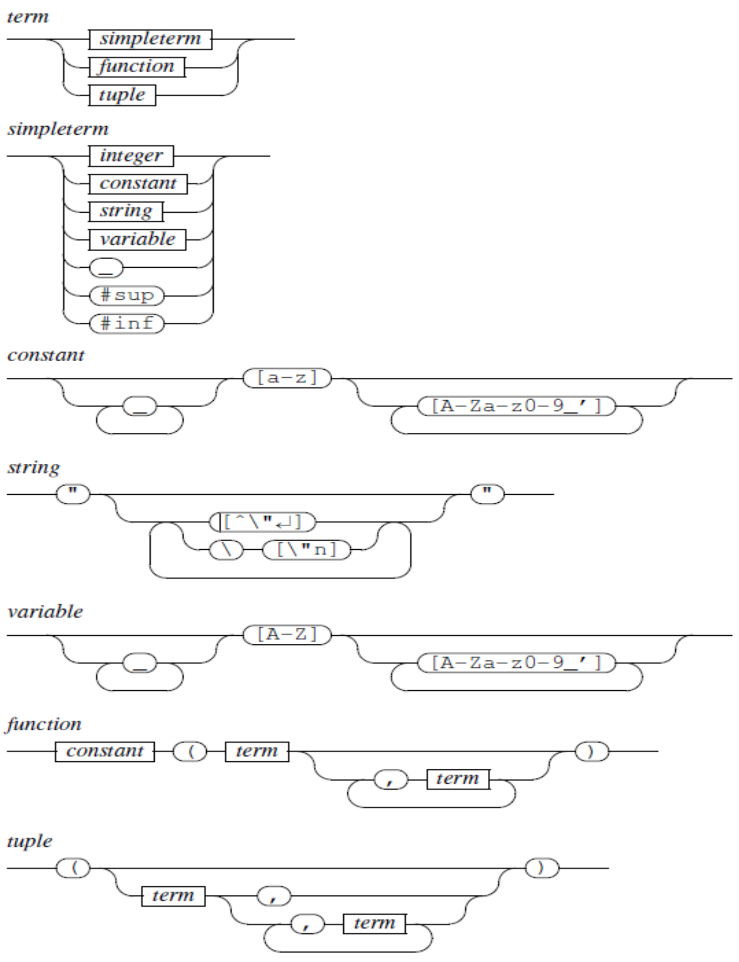

Figure 4.1: Grammar for Terms

Chapter 5

Project description

1. What will the system do?

   The system generates automatically a musical chord progression and also, based on the chords, notes that will accompany, resulting in a proper song.

2. Which is the scope of coverage your system aims for?

   The system is meant to be used by any human that enjoys listening to music or by a music player that needs ideas for new songs or backing tracks he can practise on.

3. What will be the input of your program?

   The input of my program is optional, but it consists of an integer which represents a certain answer set (i.e. a particular song).

4. What will be the output of your program?

   The output of the program will be the set of chords, the set of notes and the audio (the notes being played).

5. Which are the stakeholders of your system?

   The interesting part of the system is that it can also generate music based on given chords, thus helping in compositions. Many artists struggle with coming up with new and original sounding music. This simple tool helps them through its randomness.

5.1 Objective

The application will read all the tasks that the user has to do and the expected amount of time needed for each one of them, and then will process the data. The important activities will be prioritized (will be placed sooner in the schedule), while the optional ones will be postponed if there is not enough time for them.

5.2 Facts

It is known that music is an essential part of life and it represents many different things, such as inspiration, relaxation, a way of expressing one's thoughts, a great influence (especially for young persons) and it can bring people together.

Studies have proven that brain stimulation occurs between 3,000 and 8,800 hertz, and most music falls between his range, and classical types of music are proven to accurately do this. Some people believe that only certain types of music do this, but studies have proven that any type of music with a tone, rhythm, and pitch will stimulate the brain, but most people do not believe this. Regardless of what specific type of music does this, it proves that music is important when performing tasks that require the brain, by stimulating it.

Repetitive patterns in music are getting recognized by the brain and make people enjoy it. The application has a feature that loops the music once it's over, offering it a more stable context and making it sound more like an actual song and less like a random sequence of notes.

5.3 Specifications

The user of this application will have to input by keyboard a number (in a particular range which can be modified, initially up to 2000) or just pressing 'Enter', which is like giving as input a '1'. Additionally, constraints such as tasks which have a certain fixed time interval will be included. The Answer Set Programming tools will compute one or several solutions for a good scheduling.

5.4 Knowledge acquisition

There are not many resources on the internet about music generating software. On Answer Set Programming, I used the *Potassco* guide for `clingo`. As for inspiration, I read about the use of ASP in Music Generation in the articles linked and the end of this chapter. The first and main inspiration that I heard of is the `chasp` project from the *Potassco* official website.

Knowledge Representation

Knowledge is represented as first order logic. The syntax of ASP (clingo) is very similar to logic programming syntax (e.g. Prolog), but it also comes with new types of data. In general, we can talk about predicates, facts, integrity constraints, classical negation, double negation, head literals, conditions and condition literals.

In this particular project, the knowledge consists mainly of beats, chords and notes. These are represented as facts and predicates:

- `note(N)`
- `chord(C)`
- `beat(B)`
- `play_note(Beat,Time,Note)`,
- `play_chord(Beat,Chord)`.

The harmonic rule that give songs a nice flow is represented through *constraints* which are used to filter the bad results.

Knowledge/Data Sources

- `https://potassco.org/doc/start/`
- `https://www.cs.uni-potsdam.de/wv/chasp/`
- `https://link.springer.com/chapter/10.1007/978-3-540-89982-2_21`

5.5 Related work

There are not many resources on the internet about music generating software. On Answer Set Programming, I used the *Potassco* guide for `clingo`. As for inspiration, I read about the use of ASP in Music Generation in the articles linked and the end of this chapter. The first and main inspiration that I heard of is the `chasp` project from the *Potassco* official website.

Creating melodies, that is sequences of pitched sounds, is not as easy as it looks (sounds). We have cultural preferences for certain sequences of notes and preferences dictated by the biology of how we hear. This may be viewed as an artistic (and hence not scientific) issue, but most of us would be quick to challenge the musicality of a composition created purely by random whim. Students are taught rules of thumb to ensure that their works do not run counter to cultural norms and also fit the algorithmically definable rules of pleasing harmony when sounds are played together ([**boenn2008anton**]).

Chapter 6

Preliminary results

6.1 First Problems

The first problem encountered was the very essence of an Answer Set, specifically of a *Set*. A song should consist of multiple musical notes played in a specific order. There are 7 notes in the C Major scale and a song should definitely have more than 7 notes, therefore some (if not all) of the notes should repeat at some point in the song, but a *set* cannot have multiple instances of the same element (musical note). There must exist a way to represent facts having the same notes, but played at different moments of time, and that is exactly how the data is represented in the project.

6.2 First Lines of Code

```
note(a;b;c;d;e;f;g).
beat(1..8).

1 { play_note(B,N) : note(N) } 1 :- beat(B).
```

These are the first and most important lines of code in the project. The first line generates facts for each of the 7 notes in the C Major scale, in the following way: `note(a), note(b), ... note(g)`, the second line consists of the `beat` facts and is the short for `beat(1), beat(2), ... beat(8)`.

Finally, the third line does the *hard* job: it multiplicates the `play_note` predicate, for each note `N` and it *picks* one and only one of them for each one of the `beat` facts. In other words, random notes are generated for each one of the 8 beats, with no constraints.

Here are a few of the generated answer sets:

```
clingo version 5.3.0
Reading from prel.lp
Solving...
Answer: 1
play_note(1,c) play_note(2,e) play_note(3,g) play_note(4,d)
   play_note(5,c) play_note(6,g) play_note(7,e) play_note(8,c)
Answer: 2
play_note(1,c) play_note(2,e) play_note(3,g) play_note(4,d)
   play_note(5,c) play_note(6,g) play_note(7,f) play_note(8,c)
Answer: 3
play_note(1,c) play_note(2,e) play_note(3,g) play_note(4,d)
   play_note(5,c) play_note(6,g) play_note(7,c) play_note(8,c)
```

```
Answer: 4
play_note(1,c) play_note(2,e) play_note(3,g) play_note(4,d)
   play_note(5,c) play_note(6,g) play_note(7,e) play_note(8,g)
Answer: 5
play_note(1,c) play_note(2,e) play_note(3,g) play_note(4,d)
   play_note(5,c) play_note(6,g) play_note(7,c) play_note(8,g)
SATISFIABLE

Models       : 5+
Calls        : 1
Time         : 0.031s (Solving: 0.03s 1st Model: 0.00s Unsat: 0.00s)
CPU Time     : 0.000s
```

[language=Prolog, breaklines]

Chapter 7

Implementation details

7.1 Relevant code

Here are the provided the more important lines of code in the system.

7.1.1 Musical Notes

This is the relevant code for the musical notes which are automatically generated using `clingo`. The corresponding file for this section is called *notes.lp*.

```
note(c;d;e;f;g;a;b).
time(1;2).

lim(32).
beat(1..L) :- lim(L).

1 { play_note(B,T,N) : note(N), time(T) } 1 :- beat(B).
```

An observation is that the `play_note` predicate had been extended with a `T` parameter, which represents the Time Measure of the note; it can have the value `1` or `2`, meaning the note has the measure of 1/4 or 2/4 (supposing that the song is in 4/4). This feature makes the song more diverse and less plain sounding.

The following lines are constraints that are applied to the answers (song sets). These are usually no music theory harmony rules, but common sense rules that don't let the song get too repetitive or end out in a different key.

- `:- play_note(B,_,N), play_note(B+1,_,N), play_note(B+2,_,N).`

  This doesn't allow for more than one repetition of a note in succession.

- `:- play_note(B,1,N), play_note(B+1,2,N2), play_note(B+2,1,N3)`

  This line doesn't let 1-2-1 time measure patterns happen, because that might lead to an inconsistency and an incorrect time measure.

- `:- play_note(B,2,N), play_note(B+1,1,N2), play_note(B+2,2,N3)`

  This line doesn't let 2-1-2 time measure patterns happen, for the same reason as above.

-
  ```
  :- play_note(B,1,N1), play_note(B+1,1,N2), play_note(B+2,1,N3
  ), play_note(B+3,1,N4), play_note(B+4,1,N5).
  ```

This line doesn't allow for 5 consecutive 1-time beats.

-
  ```
  :- play_note(B,2,N1), play_note(B+1,2,N2), play_note(B+2,2,N3
     ), play_note(B+3,2,N4).
  ```

  This line doesn't allow for 4 consecutive 2-time beats.

-
  ```
  :- play_note(1,1,_), play_note(2,2,_).
  ```

  This line doesn't allow for 1-2 as the first two starting beats.

-
  ```
  :- play_note(L-1,2,N), play_note(L,1,N), lim(L).
  ```

  This line doesn't allow 2-1 as the last two ending beats.

7.1.2 Musical Chords

This is the relevant code for the musical chords which are automatically generated using `clingo`. The corresponding file for this section is called *chords.lp*.

-
  ```
  chord(cmaj;dmin;emin;fmaj;gmaj;amin).
  lim(32).
  cbeat(1..L) :- lim(P), L = P/4.

  1 { play_chord_aux(B,C) : chord(C), cbeat(B) } 1 :- cbeat(B).
  ```

  This part is very similar to the first part in the *notes* file and it works in the same way. The difference is that the chords do not have a time measure, since we are assuming that the time measure is 4/4.

-
  ```
  :- play_chord_aux(1,dmin; 1,emin; 1,fmaj; 1,gmaj).
  :- play_chord_aux(L-3,dmin; L-3,emin), lim(L).
  ```

  Two other constraints which relate more to the music theory part. The first one says that the song should only start with the C Major chord or its minor relative, the A Minor chord. This decision is made because we only work with notes and chords from the C Major scale. The second clause doesn't allow the D Minor or the E Minor chords as the last chords in the song, since they won't resolve to the C Major or the A Minor chords, in case the song must loop.

-
  ```
  play_chord(F,C) :- play_chord_aux(B,C), F = B*4 - 3.
  play_chord(F,C) :- play_chord(R,C), F = B*4 - 2, R = B*4 - 3.
  play_chord(F,C) :- play_chord(R,C), F = B*4 - 1, R = B*4 - 3.
  play_chord(F,C) :- play_chord(R,C), F = B*4, R = B*4 - 3.
  ```

  Lastly, this new predicate multiplies the previously generated chords in groups of four, so that for 4 beats, only one chord should be played.

-
  ```
  :- play_chord_aux(B,C), play_chord_aux(B+1,C).
  :- play_chord_aux(B,C), play_chord_aux(B+1,C2),
     play_chord_aux(B+2,C).
  ```

  Two constraints regarding chords: the first one doesn't allow repetitions while the second doesn't allow one repetition to occur only after 2 other chords.

7.1.3 Matching Notes with Chords

Additionally, in *notes.lp*, there is a code section that doesn't allow the notes to go out of their matching chord's set of 3 notes.

```
:- play_chord(B,amin), play_note(B,_,b). % A C E
:- play_chord(B,amin), play_note(B,_,d). % A C E
:- play_chord(B,amin), play_note(B,_,f). % A C E
:- play_chord(B,amin), play_note(B,_,g). % A C E

:- play_chord(B,cmaj), play_note(B,_,a). % C E G
:- play_chord(B,cmaj), play_note(B,_,b). % C E G
:- play_chord(B,cmaj), play_note(B,_,d). % C E G
:- play_chord(B,cmaj), play_note(B,_,f). % C E G

etc.
```

7.1.4 Python Script

The ASP tool alone cannot do more than generate the music as a set of facts. In order to read this output and convert it into a real song, we need to work with an external party. Python is the language chosen for this project, since it allows for a concise and legible code.

```
answer = input("Enter␣a␣number:␣")
if answer == "":
    answer = str(random.randint(0, 1999))
    print(f"Randomly␣picked:␣{answer}")
```

The program starts by asking for an input for selecting which one of the answer sets should be processed. The next step reads the desired answer from the set and parses the facts, storing the needed values in lists.

```
for chord_item in chords:
    chord = chord_item.replace("(", "").replace(")", "").split(",")
    if int(chord[0]) % 4 == 0:
        print(chord[1].capitalize(), end=",␣", flush=True)
print("\b\b.")
```

The chords are displayed on the screen in a readable format. Example:

```
Amin, Gmaj, Fmaj, Amin, Gmaj, Dmin, Fmaj, Emin.
```

The actual outputting of the frenquencies being played, together with some pauses and drum sound, each note is being shown on the screen together with its measure, are done in the following code.

```
    for song_item in notes:

        beat = song_item.replace("(", "").replace(")", "").split(","
            )
        note = frequencies[beat[2].upper()]
        time = int(beat[1])
        print(beat[2].upper() + "(" + str(time) + ")", end=",␣",
            flush=True)
        Beep(note, time * 300)
        measure = measure + time

        if int(beat[0]) % 2 == 0:
            PlaySound("Kick.wav", SND_ASYNC)
        if int(beat[0]) % 4 == 0:
            if pause_mode:
                remaining_measure = 8 - measure
                # print(f"-- {remaining_measure} --\n")
                if remaining_measure == 3:
                    sleep(0.3)
                    PlaySound("Kick.wav", SND_ASYNC)
                    remaining_measure -= 1
                sleep(remaining_measure * 0.3)
                PlaySound("Drumbeat.wav", SND_ASYNC)
            measure = 0
```

The `PlaySound` and `Beep` functions are imported from the `winsound` library. The frequencies for each song are as follows:

```
    frequencies = {
    "A": int(440),
    "B": int(494),
    "C": int(523),
    "D": int(587),
    "E": int(659),
    "F": int(698),
    "G": int(784 / 2),
}
```

Chapter 8

Related work and documentation

8.1 Comparison with Related Work

As a competitor in a comparison we have chosen the `chasp` project developed by *Potassco*. The main difference between `chasp` and `Answer Song Programming` is that in `chasp` you can choose a genre of the song (so the time measure and scales can be different for different genres). It also uses *LilyPond* for generating a score. An output of running the program looks like this:

```
thiskey(e,min),
cadenza(1,(1,0)), cadenza(2,(4,7)),
cadenza(3,(5,7)), cadenza(4,(1,0)),
cadenza_chord(1,e,min,0), cadenza_chord(2,a,min,7),
cadenza_chord(3,b,min,7), cadenza_chord(4,e,min,0),
cadenza_notes(1,e,g,b,e), cadenza_notes(2,a,c,e,g),
cadenza_notes(3,b,d,f,a), cadenza_notes(4,e,g,b,e)
```

However, `Answer Song Programming` has some advantages over `chasp`, mainly the fact that you can also input by hand the chord progression or alter the given one; you can even input a (one or more) notes which will be played over the chord progression. Another benefit is that you can choose to play different songs over the same chord progression, so it can help in choosing the right notes for a composition.

8.2 Limitations

The limitations of `Answer Song Programming` are:

- It has a fixed time measure of 4/4.
- It uses only two scales, namely C Major and A Minor.
- It cannot generate different genres of music.
- The sound library has only one type of sounds (the Beep sound).

8.3 Possible extensions of the current work

- Dynamic time measure.
- Use of different scales and keys.

- Generate music of various genres.
- Change the sound library with one with more sound effects. The user should be able to choose a desired instrument the music should be played on.

Chapter 9

Project Demo

Here are presented several demonstrative scenarios:

```
Enter a number(chords): 10
Enter a number (notes): 3

Amin, Gmaj, Fmaj, Amin, Gmaj, Dmin, Fmaj, Cmaj.

E(2), A(1), C(1), A(2), E(2), B(2), G(1), G(1), C(1), A(2), F(2), F
    (2), C(1), B(1), C(1), E(1), G(2), G(2), D(1), D(1), A(1), D(1),
   D(2), A(2), F(2), F(1), C(1), F(2), G(2), G(1), E(1), C(2).
```

```
Enter a number(chords): 10
Enter a number (notes): 6

Amin, Gmaj, Fmaj, Amin, Gmaj, Dmin, Fmaj, Cmaj.

E(2), A(1), C(1), A(2), E(2), B(2), G(1), G(1), C(1), A(2), F(2), F
    (2), C(1), B(1), C(1), E(1), G(2), G(2), D(1), D(1), F(1), D(1),
   A(2), D(2), F(2), F(1), A(1), F(2), G(2), G(1), E(1), C(2).
```

```
Amin, Emin, Gmaj, Amin, Dmin, Gmaj, Fmaj, Amin.

E(2), A(1), C(1), A(2), E(2), B(2), G(1), G(1), C(1), A(2), F(2), F
    (2), C(1), B(1), C(1), E(1), G(2), G(2), D(2), G(1), F(1), D(1),
   A(1), A(2), F(2), F(1), C(1), F(2), G(2), G(1), E(1), C(2).
```

Chapter 10

Original Code

This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained. Including in this section any line of code taken from someone else leads to failure of IS class this year. Failing or forgetting to add your code in this appendix leads to grade 1. Don't remove the above lines.

`notes.lp`:

```
note(c;d;e;f;g;a;b).
time(1;2).

lim(32).
beat(1..L) :- lim(L).

1 { play_note(B,T,N) : note(N), time(T) } 1 :- beat(B).

% Don't allow more than one repetition %
:- play_note(B,_,N), play_note(B+1,_,N), play_note(B+2,_,N).

% No 1-2-1 time beats %
:- play_note(B,1,N), play_note(B+1,2,N2), play_note(B+2,1,N3).

% No 2-1-2 time beats %
:- play_note(B,2,N), play_note(B+1,1,N2), play_note(B+2,2,N3).

% No 5 consecutive 1-time beats.
:- play_note(B,1,N1), play_note(B+1,1,N2), play_note(B+2,1,N3),
   play_note(B+3,1,N4), play_note(B+4,1,N5).

%:- play_note(B,_,N1), play_note(B+1,_,N2), play_note(B,_,N1),
   play_note(B+1,_,N2).

% No 4 consecutive 2-time beats
:- play_note(B,2,N1), play_note(B+1,2,N2), play_note(B+2,2,N3),
   play_note(B+3,2,N4).

% No 1-2 starting beats.
:- play_note(1,1,_), play_note(2,2,_).
```

```
% No 2-1 ending beat.
:- play_note(L-1,2,N), play_note(L,1,N), lim(L).

% No 2 pairs of notes repeating
:- play_note(B,_,N1), play_note(B+1,_,N2), play_note(B+2,_,N1),
   play_note(B+3,_,N2).

% Song should end in C or A
%:- play_note(L,_,b;L,_,d;L,_,e;L,_,f;L,_,g), lim(L).

%play_note(L,2,c; L,1,c) :- play_chord(L,cmaj), lim(L).
%play_note(L,2,a; L,1,a) :- play_chord(L,cmaj), lim(L).

% Follow the chords.
:- play_chord(B,amin), play_note(B,_,b). % A C E
:- play_chord(B,amin), play_note(B,_,d). % A C E
:- play_chord(B,amin), play_note(B,_,f). % A C E
:- play_chord(B,amin), play_note(B,_,g). % A C E

:- play_chord(B,cmaj), play_note(B,_,a). % C E G
:- play_chord(B,cmaj), play_note(B,_,b). % C E G
:- play_chord(B,cmaj), play_note(B,_,d). % C E G
:- play_chord(B,cmaj), play_note(B,_,f). % C E G

:- play_chord(B,dmin), play_note(B,_,c). % D F A
:- play_chord(B,dmin), play_note(B,_,e). % D F A
:- play_chord(B,dmin), play_note(B,_,g). % D F A
:- play_chord(B,dmin), play_note(B,_,b). % D F A

:- play_chord(B,emin), play_note(B,_,c). % E G B
:- play_chord(B,emin), play_note(B,_,d). % E G B
:- play_chord(B,emin), play_note(B,_,f). % E G B
:- play_chord(B,emin), play_note(B,_,a). % E G B

:- play_chord(B,fmaj), play_note(B,_,d). % F A C
:- play_chord(B,fmaj), play_note(B,_,e). % F A C
:- play_chord(B,fmaj), play_note(B,_,g). % F A C
:- play_chord(B,fmaj), play_note(B,_,b). % F A C

:- play_chord(B,gmaj), play_note(B,_,c). % G B D
:- play_chord(B,gmaj), play_note(B,_,e). % G B D
:- play_chord(B,gmaj), play_note(B,_,f). % G B D
:- play_chord(B,gmaj), play_note(B,_,a). % G B D


#show play_note/3.
```

`chords.lp`:

```
chord(cmaj;dmin;emin;fmaj;gmaj;amin).
lim(32).
cbeat(1..L) :- lim(P), L = P/4.
```

```
1 { play_chord_aux(B,C) : chord(C), cbeat(B) } 1 :- cbeat(B).

:- play_chord_aux(B,C), play_chord_aux(B+1,C).
:- play_chord_aux(B,C), play_chord_aux(B+1,C2), play_chord_aux(B+2,C
   ).

:- play_chord_aux(1,dmin; 1,emin; 1,fmaj; 1,gmaj).
:- play_chord_aux(L-3,dmin; L-3,emin), lim(L).
:- play_chord_aux(B,emin), play_chord_aux(B+1,cmaj).

play_chord(F,C) :- play_chord_aux(B,C), F = B*4 - 3.
play_chord(F,C) :- play_chord(R,C), F = B*4 - 2, R = B*4 - 3.
play_chord(F,C) :- play_chord(R,C), F = B*4 - 1, R = B*4 - 3.
play_chord(F,C) :- play_chord(R,C), F = B*4, R = B*4 - 3.

#show play_chord/2.
```

Python script:

```
from winsound import *
from time import sleep
import random
import sys, string, os
import subprocess

file_path = "E:\\School\\IAI\\My Files\\music.txt"
chords_file_path = "E:\\School\\IAI\\My Files\\chords.txt"
chords_music_file_path = "E:\\School\\IAI\\My Files\\chords_music.lp"
pause_mode = True

frequencies = {
    "A": int(440),
    "B": int(494),
    "C": int(523),
    "D": int(587),
    "E": int(659),
    "F": int(698),
    "G": int(784 / 2),
    "PAUSE": int(37),
}

notes = []
chords = []
chords_facts = ""

answer = input("Enter a number(chords): ")
if answer == "":
    answer = str(random.randint(0, 1999))
    print(f"Randomly picked: {answer}")

with open(chords_file_path) as file:
    chords_line = file.readline()
    while not chords_line.startswith("Answer: " + answer):
        chords_line = file.readline()
    chords_line = file.readline()

facts = chords_line.split()
for fact in facts:
    chords_facts += fact + ", "
    chords.append(fact[10:])

chords_facts = chords_facts[:-2] + '.'

with open(chords_music_file_path, 'w') as file:
    file.write(chords_facts)

# os.system('"E:/School/IAI/My Files/run.bat"')
# subprocess.call('"E:/School/IAI/My Files/run.bat"')
answer = input("Enter a number (notes): ")
if answer == "":
    answer = str(random.randint(0, 1999))
    print(f"Randomly picked: {answer}")

# Reading Notes
with open(file_path) as file:
    line = file.readline()
    while not line.startswith("Answer: " + answer):
        line = file.readline()
    line = file.readline()

facts = line.split()
for fact in facts:
    if fact[5] == 'n':
        notes.append(fact[9:])

notes = sorted(notes, key=lambda x: int(x[1:].split(",")[0]))
chords = sorted(chords, key=lambda x: int(x[1:].split(",")[0]))
print()

""" Printing song's chord progression """
for chord_item in chords:
    chord = chord_item.replace("(", "").replace(")", "").split(",")
    if int(chord[0]) % 4 == 0:
        print(chord[1].capitalize(), end=", ", flush=True)
print("\b\b.")

# print(notes)

measure = 0
print()
```

```
if pause_mode:
    PlaySound("Drumbeat.wav", SND_ASYNC)

while True:
    for song_item in notes:

        beat = song_item.replace("(", "").replace(")", "").split(",")
        note = frequencies[beat[2].upper()]
        time = int(beat[1])
        print(beat[2].upper() + "(" + str(time) + ")", end=", ", flush=True)
        Beep(note, time * 300)
        measure = measure + time

        if int(beat[0]) % 2 == 0:
            PlaySound("Kick.wav", SND_ASYNC)
        if int(beat[0]) % 4 == 0:
            if pause_mode:
                remaining_measure = 8 - measure
                # print(f"-- {remaining_measure} --\n")
                if remaining_measure == 3:
                    sleep(0.3)
                    PlaySound("Kick.wav", SND_ASYNC)
                    remaining_measure -= 1
                sleep(remaining_measure * 0.3)
                PlaySound("Drumbeat.wav", SND_ASYNC)
            measure = 0
    print("\n")
print("\b\b.")
```

Bibliography and Related Work

- https://potassco.org/doc/start/
- https://www.cs.uni-potsdam.de/wv/chasp/
- https://link.springer.com/chapter/10.1007/978-3-540-89982-2_21
- https://researchportal.bath.ac.uk/en/publications/anton-answer-set-programming-in-t
- https://www.sciencedirect.com/science/article/pii/S0378475409001165
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.840.4057&rep=rep1&type=pdf