HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale

MODERN CONCURENT SYSTEMS
MCS

DECODE NETWORK TRACE ET DRUM CHALLENGE
LABORATOIRE 1

João Miguel Domingues Pedrosa
Loïc Haas
Rick Wertenbroek

13 avril 2016

Table des matières

1 Drum challenge

1.1 Résumé du problème

L'objectif de ce laboratoire est d'analyser un fichier sous forme binaire afin de pouvoir obtenir un affichage cohérent. Il n'y a pour seule aide le format de l'affichage désiré à nous d'étudier les fichiers binaires et de voir comment en extraire les informations.

1.2 Décomposition du fichier binaire

Les valeurs à décoder sont contenues dans un fichier possédant l'extension `.splice`. Il en a été fourni 5 avec leur affichages désirés respectifs.

Pour s'aider à trouver la manière dont les informations étaient encodées, nous avons utilisé le logiciel `hexinator` permettant l'affichage du contenu d'un fichier binaire en valeur hexadécimal ou ASCII.

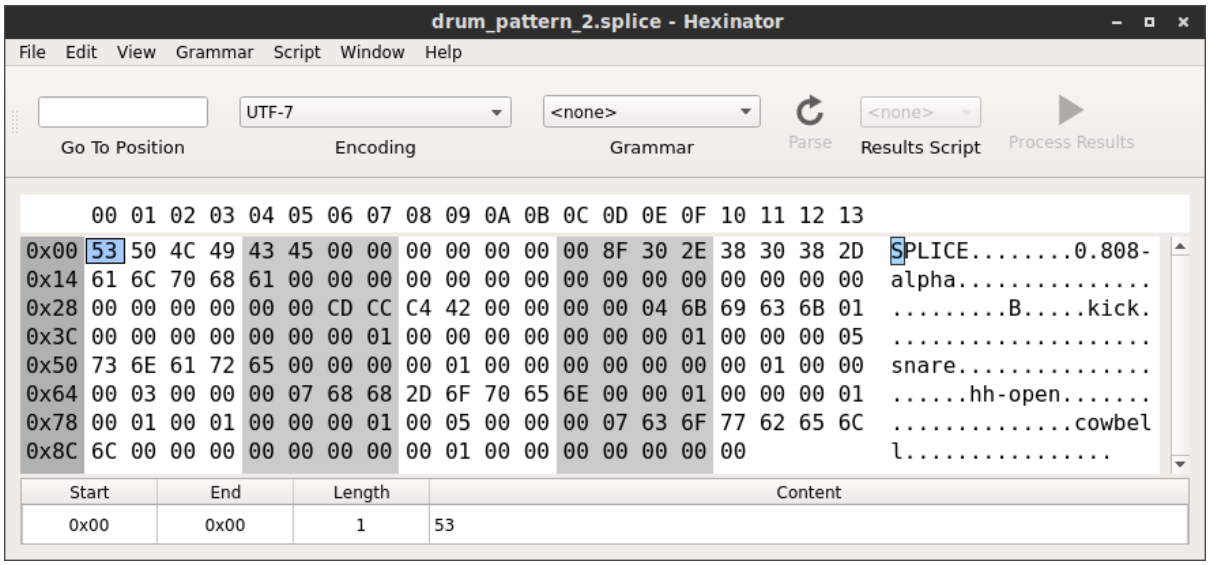

FIGURE 1 – Exemple de ce que nous sort Hexinator

Avec la comparaison des différents fichiers fournis et leur affichage désiré nous avons retiré les informations suivantes :

- Header
  - 6 octet : magic word (indique le type fichier)
  - 8 octet : longueur en octets des informations restantes
  - 32 octet : Nom de la version hardware (big-endian)
  - 4 octet : tempo en float (little-endian)
- Tracks
  - 1 octet : identifiant numérique
  - 4 octet : taille du nom de l'instrument (big-endian)
  - n octet : nom de l'instrument
  - 4 * 4 octet : la mesure décomposée en 4 quarter de 4 battement

À noter que les informations arrivent dans ce même ordre. La taille du header est constante pour chaque fichier. Par contre la taille des tracks change, parce qu'il peut y en avoir plusieurs dans un fichier et parce que le nom de l'instrument varie.

1.3 Parsing des informations

1.3.1 Header

Afin de pouvoir parser le header, nous avons réalisé la fonction suivante :

```
parse_header(Bin) ->
  case Bin of
    <<?MAGIC, Length:64, Payload:Length/binary, _/binary>> ->
      <<HW_version:32/binary, Tempo:32/little-float, Rest/binary>> = Payload,
      {ok, binary_to_string(HW_version), Tempo, Rest};
    _ -> {error, parse_header, Bin}
  end.
```

Nous commençons par regarder si le magic word est juste sinon on revoie un tuple contenant le type d'erreur ainsi que le binaire en brut. On récupère un payload de la taille définie par la valeur lue précédement. Il a été fait ainsi car il n'est possible d'avoir des octets en trop, cela n'est pas considéré comme une erreur. Ils doivent être ignorés d'où le dernier pattern matching. On peut voir ce cas dans le 5ème fichier pattern :

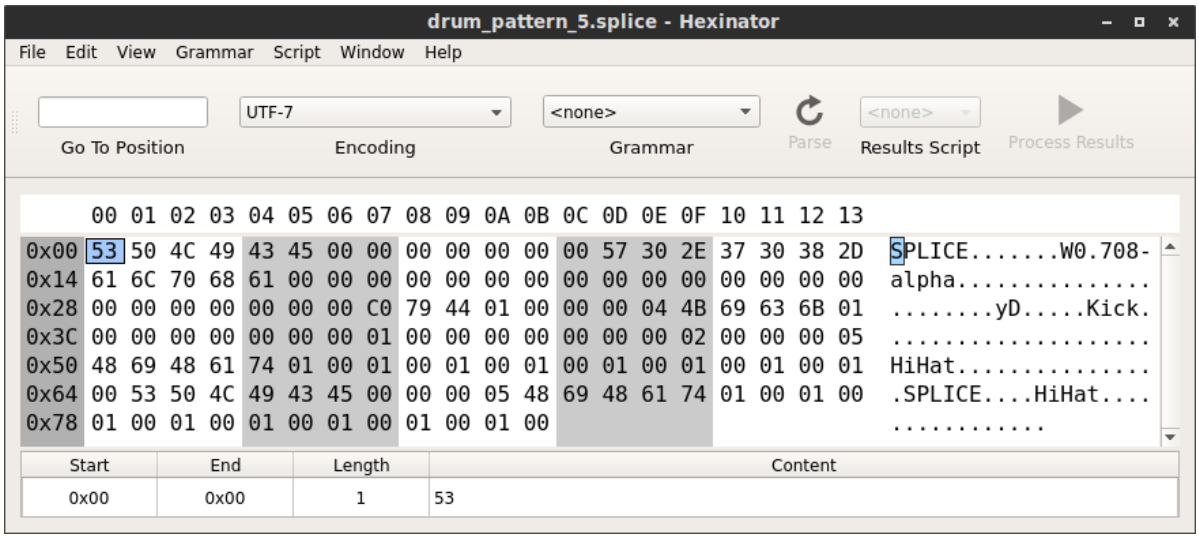

FIGURE 2 – Ici, le payload fait 87 octets, en regardant bien, on voit que l'on en a plus

Si la matching est juste, on récupère la version du hardware, le tempo et les tracks via un pattern matching avec le payload puis on retourne tout ça avec un tuple.

1.3.2 Tracks

Pour parser les tracks, nous avons décomposé la procédure en 2 fonctions, une qui se charge de faire la base (récupérer l'identifiant et le nom) et une autre pour la décomposition de la mesure.

Premièrement, la fonction pour le track complet :

```
parse_tracks(Bin) ->
  {ok,parse_tracks(Bin,[])}.

parse_tracks(<<>>,Acc) ->
  lists:reverse(Acc);

parse_tracks(<<TrackN:8, NameLen:32, Instrument:NameLen/binary, Measure:16/binary, Rest/binary>>,Acc) ->
  parse_tracks(Rest,[{TrackN, binary_to_list(Instrument), parse_measure(Measure)}|Acc]).
```

Cette partie se charge récupérer sous forme d'une liste composée du tuple id, nom de l'instrument et les mesures. Elle ne s'arrête que quand le binaire passé en argument est vide.
Ensuite, pour récupérer les mesures, on utilise la fonction suivante :

```
parse_measure(Bin) ->
  parse_measure(Bin,[]).

parse_measure(<<>>,Acc) ->
  lists:reverse(Acc);

parse_measure(<<Quarter:4/binary, Rest/binary>>, Acc) ->
  case Quarter of
    <<A:8,B:8,C:8,D:8>> when A < 2, B < 2, C < 2, D < 2 ->
      parse_measure(Rest, [binary_to_list(Quarter)|Acc]);
    _ -> {parse_measure, bad_value, Quarter}
  end.
```

Ici, on va lire récursivement le binaire passé jusqu'à ce qu'il soit vide. On récupère un quarter à la fois au quel on vérifie que chaque valeur soit 1 ou 0 sinon ça veut dire que le fichier est mal formé et qu'il y a une erreur. On revoit une liste contenant les quarter qui sont eux aussi des listes de 0 et 1.

1.4 Rendu des information

Pour le le format de l'affichage, nous avons réalisé la fonction suivante :

```
render(Version, Tempo, Tracks) ->
  io_lib:format("Saved with HW Version: ~s~nTempo: ~s~n~s",
    [Version,render_float(Tempo),render_tracks(Tracks, find_padding(Tracks))]).
```

On lui passe la version, le tempo et les tracks que l'on fait passer dans un `format` afin d'avoir un affichage correct. L'affichage du tempo se fait ainsi, s'il y a des valeurs après la virgule on les affiches sinon on affiche juste la valeur entière. Erlang n'arrivant pas à le faire de base, on passe par une fonction intermédiaire qui se charge de détecter et renvoyer le bon format :

```
render_float(Float) ->
  case Float - trunc(Float) > 0 of
    true  -> float_to_list(Float, [{decimals,3},compact]);
    false -> float_to_list(Float, [{decimals,0}])
  end.
```

Ensuite, pour les tracks, nous devons tout d'abord trouver le nombre d'espace à mettre entre le nom de l'instrument et les mesures (padding). Nous avons trouvé qu'entre 2 fichiers (par exemple : le pattern 1 et 4), l'espace minimum était différent. L'explication que nous avons trouvée à cela était : si la première lettre de l'instrument est en majuscule alors l'écart est plus grand. Ce qui donne la fonction suivante :

```
find_padding([]) ->
  0;

find_padding([{_, Instrument, _}|TTracks]) ->
  case Instrument of
    %% Check that the first character is capitalized
    [First|_] when First >= $A, First =< $Z -> 2;
    _ -> find_padding(TTracks)
  end.
```

On vérifie pour chaque instrument s'il y en a un qui commence par une majuscule au quel cas on renvoit ne nombre d'espaces à ajouter.

La fonction pour le `format` des tracks est le suivant :

```
render_tracks(Tracks,Padding) ->
  MaxLenInst = lists:foldl(
    fun({_, Instrument, _},Max) ->
      case length(Instrument) > Max of
        true -> length(Instrument);
        false -> Max
      end
    end,
    0,Tracks),

  MaxTrackN = lists:foldl(
    fun({TrackN, _, _},Max) ->
      case TrackN > Max of
        true -> TrackN;
        false -> Max
      end
```

```
17      end,
18      0,Tracks),
19
20    render_tracks(Tracks,
21      Padding + MaxLenInst + length(integer_to_list(MaxTrackN))
22      ,[]).
23
24  render_tracks([],_,Acc) ->
25    lists:reverse(Acc);
26
27  render_tracks([{TrackN, Instrument, Measure}|Rest],Padding, Acc) ->
28    render_tracks(Rest,Padding,
29      [io_lib:format("(~p) ~s ~s~s~n",
30      [TrackN,Instrument,lists:duplicate(Padding-length(Instrument)-length(integer_to_list(TrackN)),$ ),
            render_quarter(Measure)])|Acc]).
```

Le premier en-tête se charge de récupérer le nombre de caractères requis pour avoir les mesures en colonnes sur chaque ligne. On récupère la longueur du nom de l'instrument le plus grand et l'ID le plus grand. On additionne ces valeurs au padding. Avec ceci, on pourra calculer le nombre d'espaces à ajouter entre le nom et les mesures de l'instrument.

Ensuite, pour le format, on met l'ID entre parenthèses suivi du nom, après il y a le nombre d'espaces à rajouter (on crée un tableau fait que d'espaces) et on fini avec les mesures qui sont traitées dans une fonction à part et dont l'implémentation est la suivante :

```
1   render_quarter(Measure) ->
2     render_quarter(Measure,[$|]).
3
4   render_quarter([],Acc) ->
5     lists:reverse(Acc);
6
7   render_quarter([Quarter|Rest], Acc) ->
8     render_quarter(
9       Rest,
10      [[case E of 1 -> "x"; 0-> "-" end || E <- Quarter]++"|" | Acc]).
```

On crée juste un tableau remplaçant les 0 par des - et les 1 par des x. On accumule une liste des ces tableaux jusqu'à ce que l'on en ait plus à traiter.

1.5 Lecture de fichier

Pour le décodage du fichier, on utilise la fonction suivante :

```
1   decode_file(File) ->
2     Res = file:read_file(File),
3     case Res of
4       {ok, Bin} ->
5         {ok, Version,Tempo,Rest} = parse_header(Bin),
6         {ok, Tracks} = parse_tracks(Rest),
7         {ok, Version, Tempo, Tracks};
8       _ -> Res
9     end.
```

On se charge juste de parser les valeurs afin de récupérer les informations nécessaires au rendu final. On peut aussi directement obtenir l'affichage à partir du fichier. Il se charge juste d'utiliser la fonction précédente et utilise la fonction `render` avec les informations récupérées.

```
1   render_file(File) ->
2     Res = decode_file(File),
3     case Res of
4       {ok, Version, Tempo, Tracks} ->
5         {ok, render(Version, Tempo, Tracks)};
6       _ -> Res
7     end.
```

2 Decode Network Trace

2.1 Introduction

Lors de ce laboratoire Decode Network Trace, il nous est demandé de créer une fonction erlang d'analyse de binaires .pcap. Il s'agit ici, à l'effigie de tshark (ou wireshark avec sa GUI) d'analyser un fichier .pcap et d'en sortir les différents paquets capturés.

2.2 Analyse

Il a fallu analyser déjà le comportement de tshark avec ces fichiers, on peut facilement lancer `tshark -r ping.pcap` afin de voir ce qu'il nous affiche, on a aussi utilisé l'option -V pour verbose. Première choses que nous avons fait est de prendre ces affichages et de les enregistrer dans des fichiers texte afin d'avoir une référence. Ensuite nous avons analyser le contenu en détail avec wireshark. Il est très utile pour cela car il permet de voir exactement quel partie du paquet binaire correspond à quelle information.

2.3 Test Driven Development

Au départ nous avons créé deux tests comparant la sortie du programme tshark avec la sortie de la fonction erlang, étant donné que cette fonction est le but final il s'agit ici juste d'une référence et du dernier test à passer. Mais nous avons pensé bien de déjà l'avoir. Ensuite nous avons essayé de découper les différentes tâches du problème et de penser à des fonctions qui pourraient être utiles. Par exemple afficher le protocole associé à une valeur entière. Quelques tests ont été implémentés et nous avons commencé à développer. Les autres tests apparaitront au cours du développement car il est difficile de penser à toutes les choses nécessaires à la résolution du problème. Surtout que le choix de l'implémentation est laissé libre.

2.4 Réalisation

La fonction `tshark_from_file` a été implémentée afin de créer un affichage à la tshark. La version verbeuse (-V) est très longue à implémenter et nous avons décidé de ne par l'implémenter car trop longue, le test final est déjà présent mais il n'est pas encore passé. Réaliser les fonctions qui étaient déjà testées facilitait pas mal l'implémentation et le TDD nous a aidé. Cependant des choses plus complexes étaient dur à tester à l'avance. Une meilleure découpe du problème aurait du être fait (analyse plus fine) avant de commencer la programmation.

3 Conclusion

L'analyse de fichiers binaires revient à trouver l'organisation des information à l'intérieur de ceux-ci, il y a parfois de la documentation (par exemple dans le cas des binaires .pcap
https ://wiki.wireshark.org/Development/LibpcapFileFormat) et parfois nous devons faire le reverse engineering nous même, comme dans le cas du drum challenge. Le fait d'avoir déjà des tests donnés nous a aidé dans le développement. Utiliser un langage fonctionnel pour réaliser ces laboratoires a été intéressant pour nous, parfois on voit directement ce qu'il faut faire et comment l'implémenter, parfois on est perdu. Dans tous les cas nous commençons à mieux savoir comment utiliser ce nouvel outil, et il y a sûrement encore des choses qu'on a fait totalement à l'envers mais au fur à mesure nous deviendrons performant ! Il est un peu dommage que ce laboratoire demande tellement de temps dans l'analyse de format binaires plutôt que dans l'implémentation d'algorithmes intéressants et à des problèmes auxquelles ce langage particulier nous permettrait d'arriver à une solution innovante.