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 SYSTEM
MCS

PCAP ET DRUM CHALLENGE
LABORATOIRE 1

João Miguel Domingues Pedrosa
Loïc Haas
Rick Wertenbroek

Laboratoire 1

13 avril 2016

Table des matières

1 Drum challenge

1.1 Résume du problème

L'objectif de ce challenge est de pouvoir recomposer l'affichage d'un des valeurs d'un fichier contenant les informations sous forme binaire. Il y a que pour seul aide les informations sur comment ça doit s'afficher et donc quelle sont les champs qui doivent apparaître.

1.2 Décomposition du fichier binaire

Les valeurs à décoder sont contenu dans un fichier possédant l'extension `.splice`. Il en a été fournit 5 avec leur équivalent lors de l'affichage du format.

Pour m'aider dans la recherche de la décomposition des informations, on a utilisé le logiciel `hexinator` permettant l'affiche du contenu d'un fichier en valeur hexadécimal.

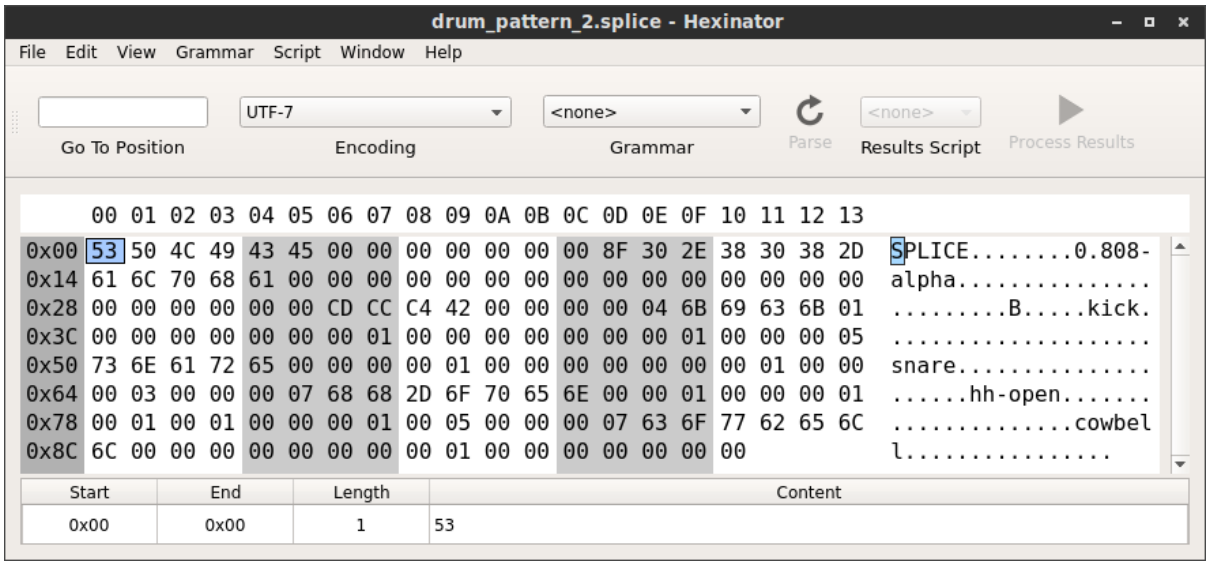

FIGURE 1 – Exemple de se que nous sort Hexinator

Avec la comparaison des différents fichiers fournit avec leur format d'affiche nous avons retiré les informations suivantes :

- Header
  - 6 octet : magic word (indique le type fichier)
  - 8 octet : longueur en octet 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 décomposer en 4 quarter de 4 battement

À noter que les information arrivent dans ce même ordre. La taille du header est constant pour chaque fichier. Par contre les tracks changent, déjà parce qu'il peut y en avoir plusieurs dans un fichier et le nom de l'instrument varie aussi.

1.3 Parsing des informations

1.3.1 Header

Pour pouvoir parser le header, nous avons fait la fonction suivant :

```
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 revoit un tuple contenant le type d'erreur et le binaire en brute. On récupére un payload de la taille définit par la valeur lu précédement. Il a été fait ainsi car il n'est pas impossible d'avoir des octet en trop, cela n'est pas considérer comme une erreur. Ils doivent être ignorer d'où le dernier match. Un exemple et trouvable dans le 5ème fichier pattern :

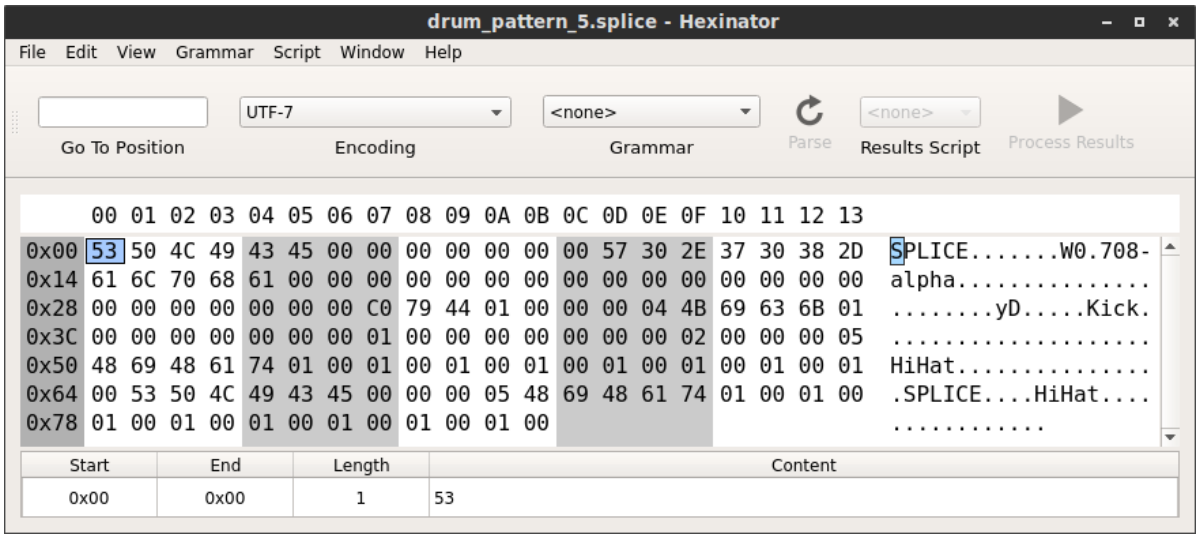

FIGURE 2 – Ici, le payload fait 87 octet, en regardant bien, on voie que l'on en a plus

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.

1.4 Tracks

Pour parser les tracks, nous avons décomposé 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 composer de 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 récursive le binaire passer 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.

1.5 Rendu des information

Pour le le format de l'affiche, on a fait 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 correcte. 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 que entre 2 fichier (exemple : le pattern 1 et 4), l'espace minimum étaient différent. L'explication que nous avons trouvé à cela était que 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'espace 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 caractère requit 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'espace à ajouter entre le nom et les mesures de l'instrument.

Ensuite, pour le format, on met l'ID entre parenthèse suivie du nom, après il y a le nombre d'espace à rajouter (on crée un tableau fait que d'espace) et on fini avec les mesure qui sont traité dans une fonction à part étant 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 tableau jusqu'à ce que l'on est plus à traiter.

1.6 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 nécessaire au rendu final. On peut aussi directement obtenir l'affichage à partir du fichier. Il se charge juste d'utilisé la fonction précédente et utilise la fonction `render` avec les informations récupéré.

```
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.
```