KOPUL - Kind Of Pack/Unpack Language

Auroux Lionel

23 juillet 2010

0.1 Description

Kopul est un langage de représentation de structure binaire et de flux. Interprété, il permet de gérer dynamiquement les parties segmentation, conversion, manipulation d'un flux de données binaire ou de structure statique. L'utilisateur disposera d'une API C permettant d'accéder de manière indexe ou par clef'*' (hashtable) au diffèrent éléments et structure du flux. Il permet de décrire de manière unique une représentation de données binaire et ce indépendamment de la machine. Enfin on pourra générer une fois pour toute un moteur de décodage en C/C++ pour l'utiliser dans tout type d'application.

Une expression KOPUL gère plusieurs chose de manière concise :

- elle decrit la structure binaire
- elle permet de gerer des variables
- elle decrit des contraintes permettant d'automatiser le decodage
- elle offre au langage C une API pour acceder dynamiquement au structure par index ou par nom

KOPUL se base sur la description de maniere cryptique de la taille des differents elements d'une structure. Elle utilise un ensemble de symbole et de type de base.

0.2 Syntaxe

0.2.1 Regles syntaxique generales

- Tout d'abord l'utilisateur a la possibilite de decrire ses structures binaires dans un fichier, ou directement en tant que parametre d'une fonction de l'API.

- La syntaxe suis les quelques regles suivantes.

- on peut ecrire des commentaires

```
commentaire ::= "//" tout caractere sauf eol EOL
          | "/*" tout caractere sauf fin comment "*/
            "
;
```

- on peut utiliser des identifiants. Attention c'est pas tout a fait pareil qu'un identifiant en C. On ne peux pas commencer par '_'

```
identifiant ::= ['a'..'z' | 'A'..'Z']['a'..'z' | '
   A'..'Z' | '0'..'9'|'_']*
;
```

on conciderera aussi les regles syntaxiques suivantes :

```
numeric ::= ['0'..'9']+
;

hex ::= '0' ['x'|'X'] [numeric | 'a'..'f'|'A'..'F'
   ]+
;

binary ::= ['0'..'1']+ ['b'|'B']
;

float ::= ...comme en c...
;

string ::= ...comme en c...
;

char ::= ...comme en c...
;
```

0.2.2 BitFields

Le but principale etant de decrire des structures binaires, nous allons au plus bas niveau manipuler des champs de bits de maniere abstraites.

```
bitfield ::= '#' numeric
;
```

ainsi nous pouvons ecrire

```
/* ceci lit un byte, un word, un dword, un qword
   */
#8 #16 #32 #64
/* ceci lit un qword,un byte, un dword, un word */
#64#8#32#16 /* et ou pas besoin de l'espace */
```

attention la valeur numeric est libre.

```
/* md5 */
#128
/* du padding */
#1024
```

```
/* et pourquoi pas */
#42
```

la seule chose a retenir c'est qu'il y a 3 champs et que le premier fait 128 bit, le second 1024, et le dernier 42. Les conditions d'acces a ces champs est decrit dans la suite de la documentation.

0.2.3 Flags

voila nous manipulons des champs de bit, mais sur nos machines nous ne possedons que des registres a taille fixe. Comment stocker un bitfield de taille 9 dans les registres du CPU qui sont a 32bit ? il y a la place mais si ces 9 bit represente une donnee numeric signee, il faudra etendre le bit de signe.

De meme comment gerer les architectures big endian ou little endian ? Les donnees peuvent finalement heberger aussi des nombres floattant en representation binaires. Cependant pour ces derniers il existe une norme ISO qui precise l'endianness et les tailles possibles. Mais ce type de donnees doit etre identifiable pour que l'on puisse charger les registre du FCPU correspondant.

Pour cela nous specifiont un certain nombre de flags permettant de preciser nos intentions.

Par defaut, nous manipulons des donnees non signes, et l'endianness des donnees lues corresponds a l'endianness de l'architecture executant les directives KOPUL.

voici la syntaxe des flags

```
bitfield ::= '#' [flags]? numeric
;

flags ::= [flags_sign [flags_endian]?] |
   flags_floattant
;

flags_sign ::= '+' /* pour indiquer que c un
   nombre signe */
;

flags_endian ::= '^' /* force en big endian */
         | '_' /* force en little endian */
;
```

```
flags_floattant ::= '.' /* indique que c'est un
   nombre floattant */
;
```

Grace au flags nous sommes capable de couvrir les types primitifs d'un langage tel que le C.

```
/* char */ #+8
/* unsigned char */ #8
/* short int */ #+16
/* unsigned short int */ #16
/* int */ #+32
/* unsigned int */ #32
/* long long int */ #+64
/* unsigned long long int */ #64
/* float */ #.32
/* double */ #.64
/* entier resaux  */ #+^16
/* entier resaux  */ #+^32
/* lire un entier intel */ #+_16
/* lire un entier intel */ #+_32
/* lit un 32 bit signe conforme a l'endianness de
   la machine */ #+~32
```

Maintenant comment pouvons nous gerer les cas suivants :

```
#+9
```

Ici nous pouvons gerer. Nous étendons le bit de signe pour tenir sur un registre. Le moteur KOPUL sera donc capable de lire une tel expressions.

```
#_14#+3
```

Avant d'appliquer l'endianness nous pouvons étendre a la taille d'un registre car l'endianness et le signe sont compatibles.

```
#.51
```

Il n'existe pas de regle evidente ou de raison evidente pour convertir #.51 en float 32 bit ou 64bit pendant le binding C. cette syntaxe est donc invalide, sauf si de nouveau standard ISO sur des tailles autres que 32 et 64 bit sont integré.

0.2.4 Types

L'utilisation brutale des bitfields rends le fichier de description rapidement hermetique. Nous allons donc rajouter la declaration de type. La declaration d'un type ne provoque aucune lecture dans le flux. C'est juste pour le rendre plus lisible.

```
type_field ::= identifiant '=' root_field
;

read_something ::= identifiant
;

root_field ::= bitfield | read_something
;

field ::= type_field | root_field
;
```

Nous pouvons donc decrire les types C de base sous forme de type kopul.

```
int8=#+8
char=int8
uint8=#8
int16=#+16
short=int16
uint16=#16
int32=#+32
uint32=#32
int64=#+64
llong=int64
uint64=#64

//ici on LIT 2 entier 16 bit
int16 uint16
```

0.2.5 Variables

L'utilisation des variables permets de stocker(ou de nommer) le contenu du flux. Au fur et a mesure que nous decrivons le flux de donnee nous pouvons definir une variable et la reutiliser. Cela permettra par exemple de parametrer le reste de la lecture suivant la valeur de la variable lue (TLV ou TV).

Nous allons prendre le principe qui veux que dans tout les cas meme si nous definissons une variable nous faisons l'action de lire le bitfield correspondant. L'avantage des variables c'est la description au file de l'eau des donnees.

```
variable ::= '$' identifiant
;

read_something ::= identifiant
;

root_field ::= bit_field | read_something
;

var_field ::= root_field ['->' identifiant]?
;

field ::= type_field | var_field
;
```

0.2.6 Structure

Outre les types primitifs, il est possible de manipuler des donnees structurees.

```
struct_field ::= '{' [field]+ '}'
;

root_field ::= bitfield | struct_field |
   read_something
;
```

maintenant les donnees peuvent etre un peu plus complexe

```
/* declarer en type une structure particulier */
coordonne={#.32->X #.32->Y #.32->Y}

// lire et stocker dans des variables et au fil de
    l'eau 2 structure imbrique
int=#+32
```

```
uchar=#8
float=#.32
{#128->padding {int uchar}->inner float }->outer
```

0.2.7 Tableau

Nous voulons aussi pouvoir repeter la lecture d'une donnee simple ou d'une donnee structure.

```
array_field ::= '[' field '(' numeric ')' ']'
;

root_field ::= bitfield | struct_field |
   array_field | read_something
;
```

voila une structure de type user

```
User={
  [#8 (20)]->Nom
  [#8 (20)]->Prenom
  [#8 (120)]->Adresse
  #16->Codepostal
  [#8 (250)]->Ville
}
```

Une autre fonction tres pratique c'est de pouvoir acceder au element a l'interieur d'une structure. Nous devons definir la localite des variables.

Par defaut,les variables sont locales au block(ou []) qui les contients. Une variable interne a une structure peut etre utiliser a l'interieur de la structure directement. Hors de la structure il faut utiliser le nom de la variable de la structure comme prefixe.

nous pouvons modifer la syntaxe de variable

```
variable ::= '$' identifiant ['.' identifiant]*
;

array_field ::= '[' field '(' numeric | variable '
   )' ']'
;
```

Donc, au lieu de donnee une valeur statique au nombre d'iteration, nous pouvons utiliser une variable dont la valeur va changer suivant ce qui a ete lue.

```
pascalString = {#8->len [#8 ($len)]}
```

ici tout va bien $len a une valeur definit hors de l'iteration. ou bien

```
cString = {[#8->c ($c)]}
```

ici $c est comparer a chaque iteration avec la valeur interne de la boucle. $c va changer a chaque iteration. Ce qui pourrait etre dangereux. La variable utilisé pour l'iteration devrait etre externe au [] pour ne pas varier pendant l'iteration et etre comparer avec la variable interne de l'implementation de []. Cela pourrait entreiner une boucle infini. Sauf que par defaut, $c est aussi comparer a 0. Lorsque celle-ci(la variable) sera egale a zero, il n'y aura plus d'iteration a faire, on sors de la boucle. Ca peut sembler une bonne implementation pour lire une chaine C. Cependant sur une chaine " x01Chaussure 0" cela n'aura pas l'effet escompté. En effet, le premier caractere x01 a pour valeur numerique 1. $c faudra donc 1 au premier caractere lue. Lorsqu'il sera compare avec la valeur interne de la boucle, il y aura egalite et donc arret. Il nous faut donc de vrais expression booleene.

Il est a noter que certaines variables sont cree implicitement. $_ correspond au dernier field lue. $[0], $[1], $[2], $[3], $... donnent acces aux valeurs des fields non-nommés (non affecte dans une variable anonyme)

(voir plus loin les variables implicites).

Et donc...

0.2.8 Expression booleene,arithmetique et switch

La traduction d'une chaine de caractere en C depend d'une condition d'arret dans la lecture d'un flux. Comment traduire cette condition au sein de KOPUL ?

Nous allons maintenant definir des expressions booleenes permettant par exemple de determiner dynamiquement la taille d'un tableau.

```
literal ::= numeric | hex |  binary | float |
   string | char
;

atom ::= variable | literal
;
```

```
shift_exp ::= atom [["<<"|">>"] atom]*
;

relat_exp ::= shift_exp [["<=" | ">=" | "<" | ">"]
    shift_exp]*
          | ["<=" | ">=" | "<" | ">"] shift_exp
;

equal_exp ::= relat_exp [["=="|"!="] relat_exp]*
          | ["=="|"!="] relat_exp // par defaut on
              prends la valeur de \$_
;

and_exp ::= equal_exp ["&&" equal_exp]*
;

exclu_or_exp ::= and_exp ["^" and_exp]*
;

incl_or_exp ::= exclu_or_exp ["|" exclu_or_exp]*
;

logical_and_exp ::= incl_or_exp ["&&" incl_or_exp
   ]*
;

bool_expr ::= '?' logical_and_expr ["||"
   logical_and_expr]*
;

array_field ::= '[' field '(' numeric | variable |
    bool_expr ')' ']'
;
```

TODO : ajouter les exp artihmetiques...
ainsi

```
/* chaine de caractere en C */
CString=[#8->c (?$c!=0)]
```

traduit une chaine de type C.

Par convention on considere que toute expression booleene fonctionne comme en C. valeur == 0 -> faux, != 0 vrais

```
/* chaine de caractere en C */
CString=[#8->c (?$c)]
```

De plus, si on omet le nom de la variable, cela sera implicitement comparer a $_.

donc

```
/* chaine de caractere en C */
CString=[#8(?)]
```

toutefois il arrive souvent dans un flux que plusieurs structure soit optionnel, et/ou que la presence d'une structure dans un flux soit determiner par la valeur de tel ou tel champs.

Pour cela nous devons introduire le concept de switch qui active la lecture d'un certain nombre de bitfield suivant la valeur d'autre champs.

```
switch ::= '(' [bool_expr field]+ ')'
;
```

ce qui donne par exemple :

```
#8->choice (
        ?==1 /*type chaine*/ [#8 (?)]->theString
        ?==2 /* double*/ #.64->theReal
        ?==4 /* struct user */ user->theUser
        ?==8 {
                #128->magic
                #2->flags_foo
                #5->flags_bar
                #1->flags_pad
                (?$flags_bar & 2 == 1 [[#8 (10)]
                   (?$flags_foo)])
        }
)
```

Dans les switchs, il est pratique d'avoir un bitfield vide. exemple :

```
#8->choice
(
        ?==1 #0 // rien de particulier
        ?==2 #32 // ici un truc
)
```

Gestion des literaux et autres constantes

En introduisant les expressions nous pouvons aussi rajouter les valeurs literales et les constantes. Cela permettra de comparer les variables plus simplement dans les switchs. Pour cela nous permettont d'affecter une valeur au variable.

```
literal_field ::= identifiant '='
        [numeric|string|hex|binary|char]
;

field ::= type_field | literal_field | var_field
;
```

exemples :

```
INT = 1
STRING = 2
MyStruct={#8
  (
  ?==INT #+32
  ?==STRING [#8(?)]
  )
}
```

On peut aussi utiliser des valeurs literales. Cependant on respectera les regles suivantes :

- 0x devant une valeur literal hexadecimale. Elle sera lue/ecrit a l'octet le plus proche.
- 0b devant une valeur literal binaire. Elle sera lue/ecrit a l'octet le plus proche.
- 0 devant une valeur octale. Elle sera lue/ecrit a l'octet le plus proche.
- 123 numerique en base 10. Elle sera lue a l'octet le plus proche.
- ' devant un caractere ascii-7 (comme en C). Elle sera lue/ecrit sur 1 octet.
- " devant une suite de caractere utf8. Elle sera lue/ecrit a l'octet le plus proche.

exemples :

```
{ 0x7f "ELF"  }
```

on peut ainsi cree des types basé sur des constantes.

```
magic_elf={ 0x7f "ELF"  }
```

```
// utilisation -> lit blabla suivie de magic_elf
blabla magic_elf
```

Enum, range et contraintes

On peut aussi definir des types enumeres avec #.... Cela fonctionne comme en C, on specifie une liste de nom. On peut specifier (ou non) la valeur associe au nom. Si la valeur n'est pas precisé on la calcul grace a la valeur precedente +1. Par defaut la premier valeur vaut 0. On peut aussi preciser une valeur shifter.... au lieu d'ecrire X=1, on ecrit X«1. Dans ce cas, pour le calcul des valeurs automatiques au lieu d'un +1 sur la valeur precedente on fait un decalage de 1 bit vers la gauche.

```
color=#{BLUE = 42, GREEN, RED} // comme en C,
    GREEN=BLUE+1, RED=GREEN+1

MyColor={#8
  (?==color.BLUE #+32
  ?==color.GREEN #[#8(?)]
  )
}

flags=#{BASE<<1,
  ON_EVENT1, ON_EVENT2, ON_EVENT3, // ici
     ON_EVENT1 == 2, ON_EVENT2 == 4, ON_EVENT3 ==
     8
  USER_CUSTOM<<1000,
  MY_EVENT1, MY_EVENT2, MY_EVENT3, // ici
     MY_EVENT1 == 1001, MY_EVENT2 == 1002,
     MY_EVENT3 == 1004
}

MyEvent={#8
  (?==flags.ON_EVENT3 #+32
  ?==flags.MY_EVENT1 #[#8(?)]
  )
}
```

Les types enumeres servent aussi pour devinir des plages de valeur (range) qui pourront etre testé ensuite.

```
skillLevelPerso=#{NOSKILL, COMMON=1..20, EXPERT
   =21..30, MASTER=31..35}

skillPerso=
{
        #8->value
        (?== skillLevelPerso.NOSKILL #0
        ?== skillLevelPerso.COMMON basicSkill
        ?== skillLevelPerso.EXPERT {basicSkill
           additionnalExpertValues}
        ?== skillLevelPerso.MASTER {basicSkill
           additionnalExpertValues
           additionnalMasterValues}
        )
}
```

A partir du moment ou un bitfield est associe a une enumeration/range on peut considerer que c'est une contrainte sur les valeurs possibles. Pour simplifier l'ecriture et marquer cette association nous rajoutons la syntaxe suivante.

```
skillLevelPerso=#{NOSKILL, COMMON=1..20, EXPERT
   =21..30, MASTER=31..35}
skillPerso=
{
        #8!skillLevelPerso->value
        $value.?NOSKILL #0 // si value vaut
           NOSKILL
        $value.?COMMON basicSkill // si value vaut
            COMMON
        $value.?EXPERT {basicSkill
           additionnalExpertValues} // si value
           vaut EXPERT
        $value.?MASTER {basicSkill
           additionnalExpertValues
           additionnalMasterValues} // si value
           vaut MASTER
}
```

L'operateur ! sur un bitfield permet d'ajouter comme contrainte d'integrite l'appartenance au valeur de l'enum. L'operateur .? permet de tester simplement la valeur d'une variable a un champ de l'enum.

Considerant que # definit non plus un enum mais un ensemble de definition, il n'est pas forcement utile de nommer les differentes valeurs possibles de l'enum. On peut se contenter de specifier simplement l'ensemble des valeurs possibles.

```
myRange=#{0..100,200..500,1000..}

// myRange peut etre utilise pour un root_field
   classique
#^20!myRange->choice

// myRange peut aussi etre utilise pour la
   definition d'un type
myInt=#16!myRange

// les variables auto sont aussi utilisable avec
   les types enum non nomme. ?0, ?1 ... reference
   dans l'ordre les
// different espace
$choice.?0 bla // 0..100
$choice.?1 blabla // 200..500
$choice.?2 blablabla // 1000.. max de #20
```

On peut aussi tout melanger. Dans certain cas, l'utilisation de () permet d'associe un sous-range a un nom.

```

contrainteDeLaMort=
        #{
          NULL, // ?.NULL ou ?.0, valeur=premier
             champs nomme par defaut 0
          15..20, // ?.1
          45, // ?.2
          50, // ?.3
          SOUS_RANGE=(100,666,700..750,999), // ?.
             SOUS_RANGE ou ?.4
          VALEUR_MAGIC=150, // ?.VALEUR_MAGIC ou
             ?.5
          VALEUR_MAGIC2, // ?.VALEUR_MAGIC2 ou
             ?.6, valeur = 151
          VALEUR_MAGIC3<<1000, // ?.VALEUR_MAGIC3
             ou ?.7, valeur = 1000
```

```
            VAL_EVE1, // ?.VAL_EVE1 ou ?.8, valeur =
                1001
            VAL_EVE2, // ?.VAL_EVE1 ou ?.8, valeur =
                1002
            VAL_EVE3, // ?.VAL_EVE1 ou ?.8, valeur =
                1004
        }

MonEntierSuperContraint=#32!contrainteDeLaMort
```

Variables implicites

Comme dit precedement $_ reference le dernier field lue.

$ reference le field global. En effet, de maniere implicite tous les fields sont conciderer comme faisant partie d'une structure global. C'est cette structure globale qui est lue ou ecrit implicitement dans le flux.

De plus ces variables possedes des proprietes statiques accessible a tout moment.

```
.byte_size // permet d'avoir la taille en octet de
    la variable
.bit_size // permet d'avoir sa taille en bit
.offset // permet d'avoir le nombre d'octet lue/
   ecrit dans le contexte actuel ($.offset
   position absolue) %%$
.parent // reference le contexte parent
... TODO
```

Fonction Type

Il est aussi pratique de definir non plus des types complets mais des types prenant en parametre des variables issues de la lecture pour coder ou decoder des valeurs issue de constante. C'est a dire des "Fonction types".

exemples :

```

/// faire un switch parametrer
VAL_CHOICE = #{UN, DEUX, TROIS}
choice($c)={$c.?UN int $c.?DEUX double $c.?TROIS
   myListe}
complexeStruct = {
```

```
    int -> entete
    char -> val
    choice($val)
}
```

Les parametres de la fonction type, sont en entrée/sortie. c'est a dire que lorsque l'expression est utilisé dans un contexte d'ecriture, ils sont en entrée. Lorsque l'expression est utilisé dans un contexte de lecture les parametres sont en sortie.

Dans les deux cas, la fonction type a pour valeur le resultat du parsing et peut etre stocké dans une variable.

En effet, dans :

```
/// encode/decode les parametres suivant la taille
    specifier par les fields
modRM($mod, $ro, $rm)={#3->rm #3->ro #2->mod}
sib($ss, $idx, $base)={#3->base #3->idx #2->ss}

R32_MAP=#{EAX=0,ECX,EDX,EBX,ESP,EBP,ESI,EDI}

// lit ou ecrit: add eax,ecx ou add \%ecx,\%eax
// soit : 0x01 11 001 000
ADD={0x01->opcode modRM(0b11, R32_MAP.ECX, R32_MAP
   .EAX)->operande }

//ADD.opcode == 01 ADD.operande == C8
```

Dereferencement

Dans les fichiers binaires on retrouve souvent des champs qui sont en faites des offsets dans le fichier pour d'autre structure. Pour ce faire on rajoute la syntaxe suivante :

```
padding ::=  "..." ['->' identifiant]?
;

deref ::= [padding]? '*' '(' offset_expression ')'
    var_field
;
```

```
root_field ::= bitfield | struct_field |
   array_field | deref | read_something
;
```

Le padding avant l'operateur de dereferencement permet de dire qu'on lit le flux jusqu'a la bonne valeur d'offset specifier par l'offset_expression. padding est obligatoire dans une description de field, optionnel dans un type. Comme la valeur de l'offset peut etre signe ou non, que le contexte peut etre la memoire (API C), le pointer peut etre n´importe ou.

exemple simple :

```
// le buffer console
console=*(0xb8000) [ {#8->c #8->a} (80*25)]

// on lit un pointeur vers une structure

#32->pointer ... *($pointer) maStruct

// un char[][] en C
[ {#32->v1 ... *($v1){[#8(?)]}}   (?$v1)]

// liste chaine
myListe={content pointer (?!=0  ...*($pointer)
   myListe)}
```

et au niveau des declarations de types et notament dans un champs de structure, on rajoute au niveau syntaxe :

```
deref_field ::=  identifiant ['.' '*' identifiant
   ]+
;

type_field ::= [identifiant|deref_field] '='
   root_field
;
```

cela permet de specifier le type issue d'un dereferencement.

par exemple un autre facon de faire une liste chaine.

```
// liste chaine
myListe={content #32->next_myListe}
myListe.*next_myListe=myListe
```

ce qui donne pour un fichier ELF.

```
ElfHeader = {
          [#8->c 16]->e_ident
          #16->e_type
          #16->e_machine
          #32->e_version
          #32->e_entry
          #32->e_phoff
          #32->e_shoff
          #32->e_flags
          #16->e_ehsize
          #16->e_phentsize
          #16->e_phnum
          #16->e_shentsize
          #16->e_shnum
          #16->e_shstrndx
}

ElfHeader.*e_shoff = {
         #32->sh_name
         #32->sh_type
         #32->sh_flags
         #32->sh_addr
         #32->sh_offset
         #32->sh_size
         #32->sh_link
         #32->sh_info
         #32->sh_addralign
         #32->sh_entsize
}
```

Padding

L'operateur ... permet de specifier des zones de padding. On ne souhaite pas decrire precisement combien d'octet sont representer par ce symbole. Cela depends de ce qui suit l'operateur de padding.

Ainsi, apres l'operateur de padding on trouve dans un premier temps l'expression pointeur *().

Mais on peut aussi preciser un literal. Dans ce cas, on va consommer les

octets jusqu'a matcher l'expression kopul suivante.

```
// pattern hexadecimal
0xacfe ... 0xffe8fbffca #8 0x24
```

Constantes etendues

Pour afiner le controle de l'encodage et du decodage des buffers on peut preciser un encodage autre que celui par defaut pour les literaux numerique ou buffer("). On peut associer un literal a un root_field scalaire ou composite qui encode/decode le buffer.

La syntaxe est la suivante :

```
// le literal 16 encode ou decode de 16 bit big
   endian
#^16->16

// constante magic correspondant a un champs 64
   bit ayant pour valeur le literal octal 0666.
Magic=#64->0666

// bitfield mapper sur une chaine utf8
#32->"ELF"

// bitfield mapper sur une chaine hexa
#32->0xcafebabe

// buffer mapper sur une structure
{#8#16}->0xAABBCCDD

// buffer mapper sur tableau
[#8(?)]->"bla\0alre"
```

Pattern matching

Nous pouvons etendre les fonctionnalites des switch et des tableaux en augmentant la capacite des expressions booleene. Comme il est possible d'ecrire des constantes buffer complexe, il suffit d'ajouter la possibilite de tester les types non-scalaire (tableau et structure) via un operateur !! .

```
souspattern1=#^54->0666
```

```
souspattern2=#64->"ze"
// parse un buffer d'une taille inconnu tant que
   celui-ci correspond au patron suivant->
[#8 (?$.parent !! {0xfce ... #8->d 0xaaaa (?$d==3
   souspattern1 ?$d==6 souspattern2)})]
```

Annotation

Apres avoir decrit un format binaire grace a kopul dans un fichier de specification on va utiliser cette specification soit en mode interpreter soit apres generation dans une librairie compile annexe. suivant les utilisations il va etre utile de disposer dans la specification d'information annexe consulter et manipuler par reflexivite. Ces meta-informations seront utile soit pour offrir differente vue de la meme specification. Attache a certain field de la specification, ces informations sont similaires aux attributs de C#.

la syntaxe est la suivante :

```
tag ::= identifiant
;

annotation ::=
  ['<' tag '->' -> '>']+
;

field ::= [annotation]*
         [type_field | literal_field | var_field]
;
```

Une annotation contient toujours un tag permettant d'identifier de quel type de note il s'agit. Le tag sers pour le programme utilisateur. Il peut donc avoir n'importe quel valeur, pourtant certaine valeur pourrait etre fixe.

exemple :

```
<autor:Auroux Lionel>
<doc: ceci est la doc associe au type ElfFormat
 bla bla bla
>
ElfFormat = {
//....
}
```

Hook

En plus du systeme d'annotation, il peut etre utile de donner la possibilite de lier un code utilisateur avec une spec afin de faciliter l'association d'un algorithme avec des donnees.

exemple :

```
&import("ctype.kopul")
&import("png.kopul")

myStruct={
        int->id
        &trace($id)
        int->key
        &decode_Key($key)->res  // ici res
           contient le resultat du parsing du hook
            decode_key
        ...*($res) png
}
```

utiliser simplement le hook se declenche lorsque l'endroit de sa definition est atteint. par exemple dans l'exemple precedent le hook "trace" se declanche apres avoir parser le int $id.

On peut aussi faire des backward declaration des hook et dans ce cas il faut preciser sur quel type,variable il se declenche. De plus, il faut aussi preciser si le hook se declenche avant ou apres le parsing du type/variable designe. Une syntaxe nouvelle est donc introduit. On peut donc s'attacher a un type/variable anonyme en utilisant la syntaxe $ et $[]

```
monType={int->id int->key ...*($res) png}

&trace($_)/monType.id // backward hook sur id de
   monType mais apres avoir parser id, $_
   reference le type courant
&decode_pkey($_._ptr)\monType.[2] //backward hook
   sur le 3ieme field de monType cad le pointer
   (...*(X) Y),
// on accede au propriete de $_ avec l'operateur
   '.' on accede ainsi a la valeur de l'expression
    pointer.
```

0.3 C Binding

Voici rapidement comment les expressions Kopul sont traduites en C.

0.3.1 Repertoire et Fichier

Un repertoire est cree par fichier .kopul parser. Ce repertoire contient tout le code correspondant a toute les types et fields decrit dans le fichier kopul.

Ce repertoire contient :

- Un fichier <type>.h par TYPE decrit dans le fichier <test>.kopul. Chaque type genere 1 fonction read_ et 1 fonction write_ declarer en static inline dans ce .h. - Un fichier <test>.h contenant les fields dans le scope $. Le scope $ genere 1 fonction read_ et 1 fonction write_ declarer en static inline dans ce .h. - Un fichier lib<test>.c instanciant toutes les fonctions de lecture/ecriture pour le fichier kopul decrit. On peut donc creer une librairie pour ce type de fichier.

0.3.2 Mapping des types scalaires

0.3.3 Mapping des types complexes