

**Dokumentacja końcowa**

# **Opis struktur binarnych**

Grzegorz Lewczuk

Projekt realizowany na przedmiot TKOM

06.06.2017

# 1 Opis projektu

---

*Język deklaracyjny opisujący struktury binarne (z możliwą dokładnością do pojedynczych bitów). Projekt powinien umożliwiać zakodowanie i zdekodowanie dowolnej opisanej struktury i prezentację jej w wybranym formacie. Powinno być możliwe zdefiniowanie pól zależnych - np. pole Length i Contents.*

Projekt zakłada opis struktury binarnej za pomocą języka opartego na **ASN1**, na potrzeby projektu używana będzie nazwa **MiniASN**, zważywszy na uproszczoną funkcjonalność języka względem pierwowzoru.

## Opis funkcjonalności

Dopuszczone są typy zmiennych **UINT**, **BITSTRING** oraz **BOOL** pierwszy przechowuje liczbę całkowitą nieujemną, drugi ciąg bitów, trzeci wartość boolowską. Pierwsze dwa typy mogą być parametryzowane, typ zmiennej parametryzujemy liczbą bitów przeznaczonych na jej zapis. Standardowo

`UINT ::= UINT_8`

`BITSTRING ::= BITSTRING_8`

Typ **SEQUENCE** będzie typową strukturą, zawierającą inne struktury

Dodatkowo sparametryzowany będzie typ **CHOICE**, który będzie przyjmował wielkość w bitach odpowiednią dla pola spełniającego warunek. W strukturze **CHOICE** powinna być także wartość domyślna, gdy żadna inna nie spełni swojego warunku.\*

\* pokazane dokładniej w przykładowym kodzie

Możliwa będzie także deklaracja tablic **ARRAY** parametryzowana identyfikatorem zmiennej lub liczbą zawierającą ilość elementów oraz opisem zawartości każdego elementu

## 2 Składnia i opis języka

---

### Lista tokenów

'SEQUENCE' , 'CHOICE' , 'ARRAY' , 'UINT' , 'BITSTRING' ,  
'BOOL' , '::=' , '\_' , '[' , ']' , '{' , '}' , '=' , '<' ,  
'>' , '<=' , '>=' , '!=' , ',' , 'DEFAULT' , 'AND' , 'OR' ,  
'TRUE' , 'FALSE' , 'TYPE'

## Gramatyka

```
data_structure = { declaration }
declaration = id '::=' ( sequence_declaration | choice_declaration | array_declaration |
simple_type )

sequence_declaration = 'SEQUENCE' [ arguments ] '{' attribute { attribute } '}'

choice_declaration = 'CHOICE' arguments '{' { choice_attribute } choice_attribute_default
'}'
choice_attribute = type '(' ( or_expression | 'DEFAULT' ) ')'

array_declaration = 'ARRAY' arguments '{' attribute { attribute } '}'

arguments = '[' id { id } ']'
attribute = id type

type = ( simple_type | declared_type )
simple_type = ( simple_type_parametrized | 'BOOL' )
simple_type_parametrized = ( 'UINT', 'BITSTRING' ) [ '_' number ]
declared_type = id [ parameters ]

parameters = '[' parameter { parameter } ']'
parameter = ( id | number )

or_expression = and_expression { 'OR' and_expression }
and_expression = simple_expression { 'AND' simple_expression }
simple_expression = id relational_operator ( value | id )
relational_operator = ( '==' | '!=' | '<' | '>' | '<=' | '>=' )

value = ( number | boolean )
number = ( digit )
digit = '0'..'9'
boolean = ( 'TRUE' | 'FALSE' )

id = letter { ( letter | digit ) }
letter = ( 'a'..'z' | 'A'..'Z' )
```

## 3 Wymagania

---

### Wymagania funkcjonalne

1. Odczyt danych binarnych na podstawie dostarczonej struktury danych
2. Sprawdzanie poprawności struktury danych (dostarczonego kodu)
3. Poprawne zdekodowanie danych binarnych do postaci wygodnej do odczytu dla człowieka

## Wymagania нефunkcjonalne

1. Jak najdokładniejsze informacje o błędach w strukturze danych

## 4 Sposób uruchomienia

---

Aplikacja uruchamiana będzie wraz z dostarczonym opisem struktury danych oraz plikiem binarnym zawierającym dane. Na ekranie powinien wyświetlić się odpowiedni wynik dekodowania zrozumiały dla człowieka.

Uruchomić będzie można poleceniem:

```
python miniASN.py example.miniasn dane.bin nazwa_deklaracji argumenty
```

## 6 Przykładowy kod

---

```
bit16 ::= BITSTRING_16
uint8 ::= UINT_8

testChoice::=CHOICE[a]
{
    UINT(a>0 AND a < 100)
    BOOL(a > 170 AND a < 200)
    bit16(a == 100 OR a == 110)
    uint8(a > 202)
    BITSTRING(DEFAULT)
}
```

```
intArray::=ARRAY[a]
{
    number UINT
}
```

```
testArray::=ARRAY[a]
{
    param uint8
    choice testChoice[param]
    array intArray[3]
}
```

```
littleSeq::=SEQUENCE[a b]
{
    check BOOL
    nums intArray[a]
    nums2 intArray[b]
}

mediumSeq::= SEQUENCE[a b c] {
    int9 UINT_9
    array testArray[c]
    seq littleSeq[b a]
}

bigSeq::= SEQUENCE[a b] {
    uint uint8
    mSeq mediumSeq[a 1 b]
    array testArray[b]
}

arrSeq::= ARRAY[a]
{
    param UINT_5
    seq littleSeq[param param]
}
```

Aplikacja uruchomiona dla przykładowych struktur z losowymi danymi:

```
python MiniASN.py example.miniasn data.bin arrSeq 3
```

Daje następujący wynik:

```
arrSeq['3'] = [
{
  param = 1,
  seq = {
    check = False,
    nums = [
      {
        number = 130,
      },
    ],
    nums2 = [
      {
        number = 147,
      },
    ],
  },
},
{
  param = 3,
  seq = {
    check = False,
    nums = [
      {
        number = 247,
      },
      {
        number = 38,
      },
      {
        number = 86,
      },
    ],
    nums2 = [
      {
        number = 210,
      },
      {
        number = 6,
      },
      {
        number = 151,
      },
    ],
  },
},
{
  param = 0,
  seq = {
    check = True,
    nums = [
    ],
    nums2 = [
    ],
  },
},
]
```

## 7 Bardziej życiowy przykład

---

```
Car ::= BITSTRING_0
Bus  ::= BITSTRING_0
Truck ::= BITSTRING_0
Motorbike ::= BITSTRING_0
None  ::= BITSTRING_0

VehicleType ::= CHOICE [type]
{
    Car      (type == 0)
    Bus      (type == 1)
    Truck    (type == 2)
    Motorbike (type == 3)
    None     (DEFAULT)
}

VimNumber ::= BITSTRING_18

Vehicle ::= SEQUENCE [type]
{
    vehicleType VehicleType [type]
    horsepower  UINT
    weight      UINT_15
    hasAirbags  BOOL
    maxSpeed    UINT
    vimNumber   VimNumber
}

Vehicles ::= ARRAY [n]
{
    type      UINT_2
    vehicle   Vehicle[type]
}

VehiclesInGarage ::= CHOICE [nVehicles nSpaces]
{
    Vehicles [nVehicles] (nVehicles <= nSpaces)
    Vehicles [nSpaces]   (DEFAULT)
}

EnoughSpaces ::= BITSTRING_0
NotEnoughSpaces ::= BITSTRING_0

IsEnoughSpaces ::= CHOICE [nVehicles nSpaces]
{
    EnoughSpaces (nVehicles <= nSpaces)
    NotEnoughSpaces (DEFAULT)
}

Garage ::= SEQUENCE [nVehicles nSpaces]
{
    isEnoughSpaces IsEnoughSpaces [nVehicles nSpaces]
    vehicles       VehiclesInGarage [nVehicles nSpaces]
}
```

Przy wywołaniu z losowymi danymi daje wynik:

```
Garage['7', '3'] = {
  isEnoughSpaces = (NotEnoughSpaces) ,
  vehicles = (Vehicles[3]) [
    {
      type = 1,
      vehicle = {
        vehicleType = (Bus) ,
        horsepower = 49,
        weight = 24292,
        hasAirbags = True,
        maxSpeed = 149,
        vinNumber = 101101001000000110,
      },
    },
    {
      type = 2,
      vehicle = {
        vehicleType = (Truck) ,
        horsepower = 92,
        weight = 3694,
        hasAirbags = True,
        maxSpeed = 91,
        vinNumber = 010010000001100100,
      },
    },
    {
      type = 1,
      vehicle = {
        vehicleType = (Bus) ,
        horsepower = 189,
        weight = 22750,
        hasAirbags = True,
        maxSpeed = 200,
        vinNumber = 100000011100110110,
      },
    },
  ],
}
```

Język pozwala na wprowadzenie konstrukcji typu Enum poprzez deklarację własnych typów BITSTRING\_0 oraz odpowiedniej struktury CHOICE, można także używać ich do przekazywania informacji

Pozwala także na operacje warunkowe, w powyższym przykładzie, gdy chcieliśmy zaparkować więcej samochodów niż było miejsc, liczba ta została ograniczona do liczby miejsc