



# ASN.1

## Oraz Kodowanie BER w kontekście protokołu SNMP

MODELOWANIE PROJEKTOWANIE I  
ANALIZA SIECI KOMPUTEROWYCH

Rafał Skowroński

# ASN.1, SMI, PLIKI MIB *CO TO TAKIEGO? :)*

## W skrócie:

**Notacja:** ASN.1 > SMI **Kodowanie:** BER **Baza danych:** MIB

## A konkretniej:

**ASN.1** – popularna semantyka / standard używany do opisu struktur danych.

**Cel:** Reprezentacja danych niezależna od sprzętu.

**SNMP** (RFC-1157) – definiuje formaty pakietów używanych przez protokół SNMP

**SMI** (RFC-1155) – definiuje pod-składnie ASN.1 czyli SMI. Jest to składnia której można używać do definiowania obiektów w MIBach.

**MIB-II** (RFC-1213) – jedna z wielu „grup zarządzalnych” definiujące same obiekty do których mamy dostęp przez sieć (parametry, zmienne, tablice). MIB-II musi być obsługiwany przez każde urządzenie obsługujące protokół SNMP.



# FUNKCJONALNOŚĆ: **PARSER SMI**

# LOKALIZACJA PLIKÓW MIB

Program agenta który tworzymy potrzebuje korzystać z plików MIB.

Uznajmy, że nasz program korzysta z plików znajdujących się w folderze MIBS pakietu NET-SNMP.

Do testów jako dane wejściowe podajemy plik RFC-1213-MIB.txt

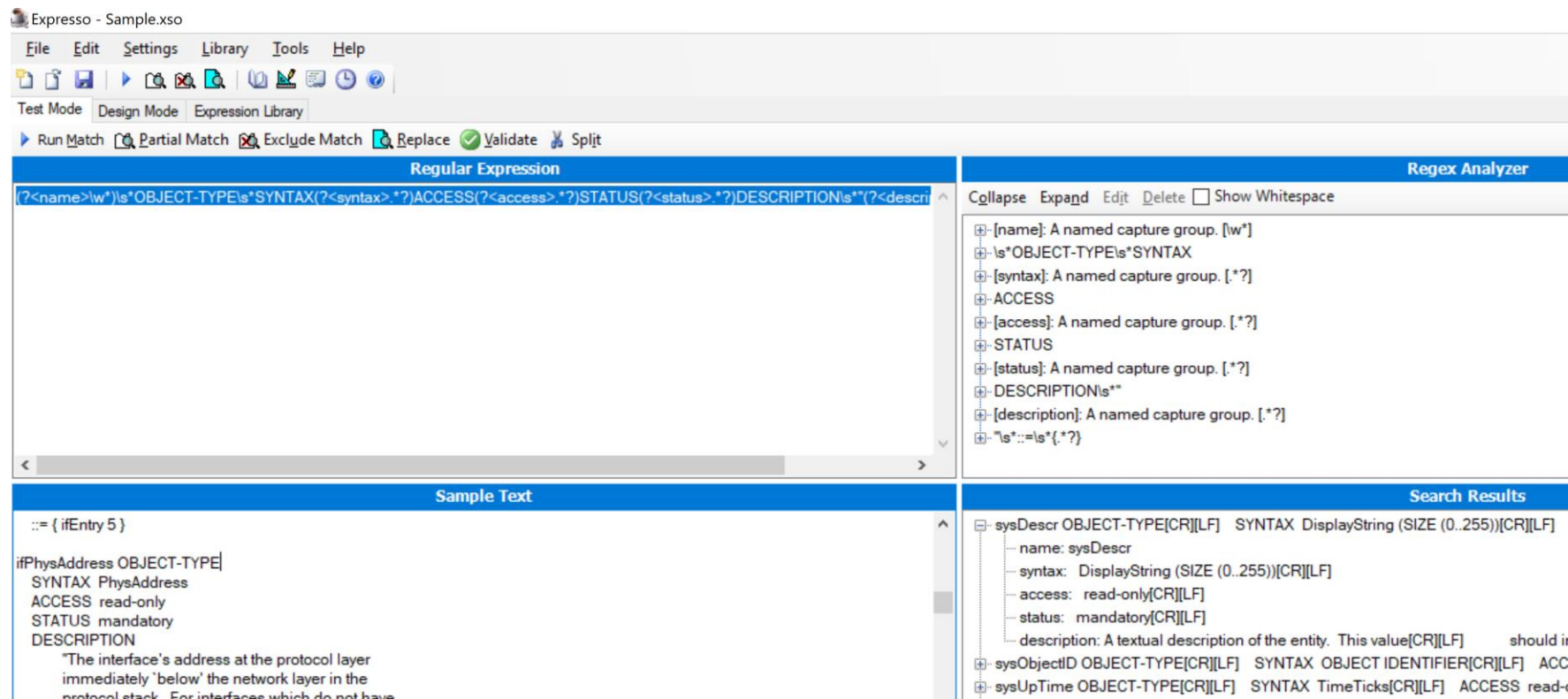


Plik RFC-1213-MIB.txt będzie importować inne pliki.

# WYRAŻENIA REGULARNE

Do parsowania plików wejściowych będzie nam potrzebny silnik wyrażeń regularnych. Np. REGEX.

Do testowania/projektowania wyrażeń regularnych możemy wykorzystać darmowy program EXPRESSO: [Pobierz](#)



# WYRAŻENIA REGULARNE C.D

Przykładowy (nie idealny) REGEX do parsowania makr OBJECT-TYPE:

```
\w*\s*OBJECT-TYPE\s*SYNTAX.*?ACCESS.*?STATUS.*?DESCRIPTION\s*".*?"\s*::=\s*{.*?}
```

Na przykładzie platformy .NET aby zadziałał poprawnie należy zaznaczyć flagę SINGLE-LINE, przydatna też będzie flaga COMPILED.

**W przykładzie:**

**\w\*** – dowolna ilość znaków alfanumerycznych – łapie nazwę obiektu

**\s\*** – dowolna ilość znaków pustych **UWAGA:** flaga SINGLE-LINE sprawia że łapie też znaki nowej linii

**.\*?** – – dowolna ilość dowolnych znaków **UWAGA: ?** Powoduje że działa w trybie **NON-GREEDY** (nie zachłanny, czyli nie złapie wszystkich znaków do końca ciągu wejściowego tylko sprawdza jeszcze co jest dalej; można nazwać też przeszukiwaniem z wyprzedzeniem bo patrzymy ,oknem przeszukiwania' co jeszcze jest dalej w wyrażeniu regularnym)

# WYRAŻENIA REGULARNE C.D

Przydatne mogą się okazać „GRUPY”. Przykładowo, silnik REGEX na platformie .NET wspiera nazwane grupy (*named groups*). Mogą być pomocne przy odczytywaniu fragmentów z rozpoznanego ciągu danych. Za pomocą *grup nazwanych* przeróbmy wcześniejsze wyrażenie aby w łatwy sposób pobrać poszczególne pola:

```
(?<name>\w*)\s*OBJECT-TYPE\s*SYNTAX(?<syntax>.*?)ACCESS(?<access>.*?)STATUS(?<status>.*?)DESCRIPTION\s*"(?<description>.*?)"\s*::=\s*{.*?}
```

**REZULTAT W CZASIE TESTÓW** (program EXPRESSO):

```
Search Results
- sysDescr OBJECT-TYPE[CR][LF] SYNTAX DisplayString (SIZE (0..255))[CR][LF] ACCESS read-only
  .... name: sysDescr
  .... syntax: DisplayString (SIZE (0..255))[CR][LF]
  .... access: read-only[CR][LF]
  .... status: mandatory[CR][LF]
  .... description: A textual description of the entity. This value[CR][LF] should include the full name
```



# TYPY DANYCH

*W protokole SNMP mamy do czynienia z pewnym zbiorem przesyłanych oraz przetwarzanych typów danych. Podstawowe typy danych są określone w ramach SMI, która to notacja jest podzbiorem ASN.1*



# DOZWOLONE TYPY DANYCH

SMI obsługuje „prymitywne” typy, ze składni ASN.1 czyli:

1. INTEGER
2. OCTET STRING
3. OBJECT IDENTIFIER
4. NULL

Oraz jeden typ złożony - [SEKWENCJE](#)



W MIBach, za pomocą podstawowych „prymitywnych” typów danych możemy definiować bardziej złożone typy.

# TYP *SEQUENCE (OF)*

## Cel:

Można przyrównać do struktury w C (może zawierać wiele typów)

## Składnia

SEQUENCE { <type1>, ..., <typeN> } – może zawierać elementy różnego typu

## Uwagi:

W porównaniu z notacją ASN.1, SMI nie pozwala na użycie parametru OPTIONAL oraz DEFAULT dla parametrów. Po zdefiniowaniu wszystkie muszą być podane.

## SKŁADNIA:

SEQUENCE OF <entry> - może zawierać elementy jedynie jednego typu  
(tak jak tablica w C)

Aby obsługiwać sekwencje możemy np. stworzyć uniwersalną klasę **TypDanych** w której stworzymy listę zawierającą nazwy typów które sekwencja zawiera.

# TYPY DZIEDZICZONE Z TYPÓW PODSTAWOWYCH

Dodatkowe typy danych mogą być zadeklarowane w dowolnym z zaimportowanych plików.

- Składają się z podstawowych typów danych ASN.1

## PRZYKŁADY:

IpAddress ::=

[APPLICATION 0]

IMPLICIT OCTET STRING (SIZE (4))

Adres IP ma 32 bity = 4 bajty – długość ciągu bajtów

Counter ::=

[APPLICATION 1]

IMPLICIT INTEGER (0..4294967295)

Dozwolone wartości z przedziału 0 - 4294967295

Gauge ::=

[APPLICATION 2]

IMPLICIT INTEGER (0..4294967295)

TimeTicks ::=

[APPLICATION 3]

IMPLICIT INTEGER (0..4294967295)



# PARSOWANIE NOWYCH TYPÓW DANYCH

IpAddress ::=  
[APPLICATION 0] -- in network-byte order  
IMPLICIT OCTET STRING (SIZE (4))

Wykorzystujemy wyrażenie regularne wyłapujące bloki zawierające:



**Znaki puste (white space):**  
spacje, tabulacje, znaki nowej linii

1) ciąg znaków alfa-numerycznych

2) znak przypisania ::=

2) nawiasy prostokątne, w środku:

- Jedna z czterech wartości: UNIVERSAL, APPLICATION, CONTEXT-SPECIFIC, PRIVATE – wartość ta oznacza „widoczność” deklarowanego typu.
- LICZBA (w tym przypadku 0) oznaczająca identyfikator typu

3) Słowo kluczowe IMPLICIT lub EXPLICIT

4) Nazwa typu macierzystego (w tym przypadku OCTET STRING)

5) **OPCJONALNE** rozmiar danych / dozwolone wartości. Obsługujemy dwie notacje ograniczeń:

- notacja (SIZE(ROZMIAR\_STRUKTURY))
- notacja (WARTOSC\_MINIMALNA..WARTOSC\_MAX)

Obsługujemy jedynie takie format zapisu. W wyniku czego może się zdarzyć, że OBJECT TYPE zawiera nieobsługiwany przez nas typ danych.

# PARSOWANIE NOWYCH TYPÓW DANYCH C.D

Poniżej przykładowe wyrażenie regularne parsujące nowe typy danych z wykorzystaniem *named capture groups* do rozpoznania poszczególnych pól:

```
\w*\s*::=\s*[/\s*\w*\s*(?<typeID>\d+)\s*[/\s*\w*\s+(?<parentType>\w*\s*\w*)\s*(?<restrictions>\(?.*?\)\s*)?
```

**Przykładowy efekt:**

Przykładowe wyrażenie może zawierać błędy / i zawiera braki.

Sample Text	Search Results
<pre>ipAddress ::= [APPLICATION 0] IMPLICIT OCTET STRING (SIZE(4))  ipAddress2 ::= [APPLICATION 1] IMPLICIT OCTET STRING (SIZE(4)) fdfg  valve ::= [APPLICATION 33] IMPLICIT INTEGER (0..255) fdfg</pre>	<pre>ipAddress ::= [CR][LF] [APPLICATION 0][CR][LF] IMPLICIT OCTET STRING (SIZE(4))   typeID: 0   parentType: OCTET STRING   restrictions: (SIZE(4)) ipAddress2 ::= [CR][LF] [APPLICATION 1][CR][LF] IMPLICIT OCTET STRING (SIZE(4))   typeID: 1   parentType: OCTET STRING   restrictions: (SIZE(4)) valve ::= [CR][LF] [APPLICATION 33][CR][LF] IMPLICIT INTEGER (0..255)   typeID: 33   parentType: INTEGER   restrictions: (0..255)</pre>

# CO ZNAJDZIEMY W MIBACH

```
RFC1213-MIB DEFINITIONS ::= BEGIN
```

```
IMPORTS
```

```
    mgmt, NetworkAddress, IpAddress, Counter, Gauge,
        TimeTicks
    FROM RFC1155-SMI
    OBJECT-TYPE
        FROM RFC-1212;
```

## Importy

Do „czarnej listy” –  
nie przetwarzamy

```
-- This MIB module uses the extended OBJECT-TYPE macro as
-- defined in [14];
```

```
-- MIB-II (same prefix as MIB-I)
```

```
mib-2      OBJECT IDENTIFIER ::= { mgmt 1 }
```

```
-- textual conventions
```

```
DisplayString ::=
```

```
    OCTET STRING
```

```
-- This data type is used to model textual information taken
-- from the NVT ASCII character set.  By convention, objects
-- with this syntax are declared as having
```

Uwaga: może być też zapis w formacie:

iso(1) org(3) dod(6) internet(1) mgmt(2) mib-2(1)  
system(1) 3 // z ID w nawiasie przy nazwie

```
-- groups in MIB-II
```

```
system      OBJECT IDENTIFIER ::= { mib-2 1 }
```

```
interfaces  OBJECT IDENTIFIER ::= { mib-2 2 }
```

```
at          OBJECT IDENTIFIER ::= { mib-2 3 }
```

```
ip          OBJECT IDENTIFIER ::= { mib-2 4 }
```

## Deklaracje samych identyfikatorów

```
sysUpTime OBJECT-TYPE
```

```
    SYNTAX  TimeTicks
```

```
    ACCESS  read-only
```

```
    STATUS  mandatory
```

```
    DESCRIPTION
```

```
        "The time (in hundredths of a second) since the
        network management portion of the system was last
        re-initialized."
```

```
    ::= { system 3 }
```

## Deklaracje obiektów

## Deklaracje typów danych

```
IpAddress ::=
```

```
    [APPLICATION 0]      -- in network-byte order
```

```
    IMPLICIT OCTET STRING (SIZE (4))
```

```
Counter ::=
```

```
    [APPLICATION 1]
```

```
    IMPLICIT INTEGER (0..4294967295)
```

```
Gauge ::=
```

```
    [APPLICATION 2]
```

```
    IMPLICIT INTEGER (0..4294967295)
```

```
TimeTicks ::=
```

```
    [APPLICATION 3]
```

```
    IMPLICIT INTEGER (0..4294967295)
```

# MAKRO OBJECT-TYPE

1) Plik MIB składa się z wielu obiektów **OBJECT-TYPE**

2) Każdy taki obiekt deklaruje parametr do **odczytu/zapisu**

wraz z jego typem danych, identyfikatorem OID, prawami dostępu oraz opisem.

Każdy obiekt zadeklarowany w pliku MIB można przyrównać do interfejsów w językach programowania wysokopoziomowego takich jak C++ / C#. MIB sam w sobie nie zawiera wartości. Definiuje jedynie zasady i to do czego mamy dostęp.

# PARSOWANIE IDENTYFIKATORÓW OID

Deklaracje identyfikatorów OID zawierają się w klamrach. Mogą zawierać:

- Nazwe słowną np. iso
- Nazwe słowną wraz z liczbą np. mgmt(2)
- Liczbe np. 1

## Przykłady:

```
mib-2      OBJECT IDENTIFIER ::= { mgmt 1 }
internet   OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
```

## Postępujemy w następujący sposób:

- 1) Jeśli napotkamy na nazwę słowną np. mgmt bez nawiasu – szukamy całego OID w drzewie, następujące później elementy będą dziećmi. Nie będzie sytuacji: ~~iso mgmt dod~~
- 2) Jeśli napotkamy na nazwę słowną z nawiasem np. org(3) oznacza liścia org z identyfikatorem 3 może ich być kilka po sobie. Org(3) dod(6) – oznacza liścia dod z identyfikatorem 6 pod rodzicem Org z identyfikatorem 3
- 3) Na końcu zawsze będzie liczba – jest potrzebny identyfikator dla nowego elementu



Ograniczenia mogą znajdować się w deklaracji typu, ale mogą też znajdować się w deklaracji obiektu. W takim przypadku ograniczenia w deklaracji obiektu mają większy priorytet.

# PARSOWANIE MAKRA OBJECT TYPE

```
sysDescr OBJECT-TYPE  
    SYNTAX  DisplayString (SIZE (0..255))  
    ACCESS  read-only  
    STATUS  mandatory
```

## DESCRIPTION

```
"A textual description of the entity. This value  
should include the full name and version  
identification of the system's hardware type,  
software operating-system, and networking  
software. It is mandatory that this only contain  
printable ASCII characters."
```

```
::= { system 1 }
```

Lub format ograniczeń ([MIN\\_VAL.. MAX\\_VAL](#)) – identycznie jak w przypadku ograniczeń przy deklaracjach typów można wykorzystać te same wyrażenia regularne.

Czyli ograniczenia potrzebujemy przechowywać w 2 miejscach:

- W typie danych
- W danym nodzie drzewa

## LUB

Tworzymy dynamicznie dodatkowe typy danych

Makro OBJECT-TYPE – „hard-kodujemy” tzn. tworzymy wyrażenie regularne, które pozwala na wyluskanie potrzebnych elementów. Elementy występują zawsze w tej samej kolejności. Tj. słowa kluczowe OBJECT-TYPE, SYNTAX, ACCESS, STATUS, DESCRIPTION, oraz następujący na końcu OID. Inne podejście wymagałoby dynamicznej obsługi notacji tworzącej dowolne makra w notacji ASN.1

**Dla chętnych:** obsługa pola DEFVAL – zawierającego wartość domyślną dla danej zmiennej. [RFC](#) punkt 4.1.7 opisuje możliwe formaty domyślnych wartości.

# OBIEKT SMI C.D

Przykładowe obiekty z MIB-II wykorzystujące już te makro:

```
sysUpTime OBJECT-TYPE
    SYNTAX  TimeTicks
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "The time (in hundredths of a second) since the
        network management portion of the system was last
        re-initialized."
    ::= { system 3 }
```

```
sysContact OBJECT-TYPE
    SYNTAX  DisplayString (SIZE (0..255))
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
        "The textual identification of the contact person
        for this managed node, together with information
        on how to contact this person."
    ::= { system 4 }
```

Nie piszemy pełnego parsera ASN.1; zajmujemy się tylko składnią SMI.  
Czyli np. projektujemy REGEX'a do obsługi składni OBJECT-TYPE:


Fragment RFC 1155-SMI zawierający definicje marka OBJECT TYPE:

```
ObjectName ::=
    OBJECT IDENTIFIER

OBJECT-TYPE MACRO ::=
BEGIN
    TYPE NOTATION ::= "SYNTAX" type (TYPE ObjectSyntax)
                    "ACCESS" Access
                    "STATUS" Status
    VALUE NOTATION ::= value (VALUE ObjectName)

    Access ::= "read-only"
              | "read-write"
              | "write-only"
              | "not-accessible"
    Status  ::= "mandatory"
              | "optional"
              | "obsolete"

END
```



# OPIS ZADANIA

Zadanie polega na zaimplementowaniu odczytu danych z pliku MIB-2. Aby struktury danych w programie były jak najlepiej dopasowane do formatu danych wejściowych, wykorzystujemy fakt, że identyfikatory OID tworzą strukturę drzewiastą.

W każdym elemencie drzewa, zapisujemy dane zawarte w danym obiekcie.

# TASK1 — ‚PARSER’ SMI; - *WERSJA PODSTAWOWA*

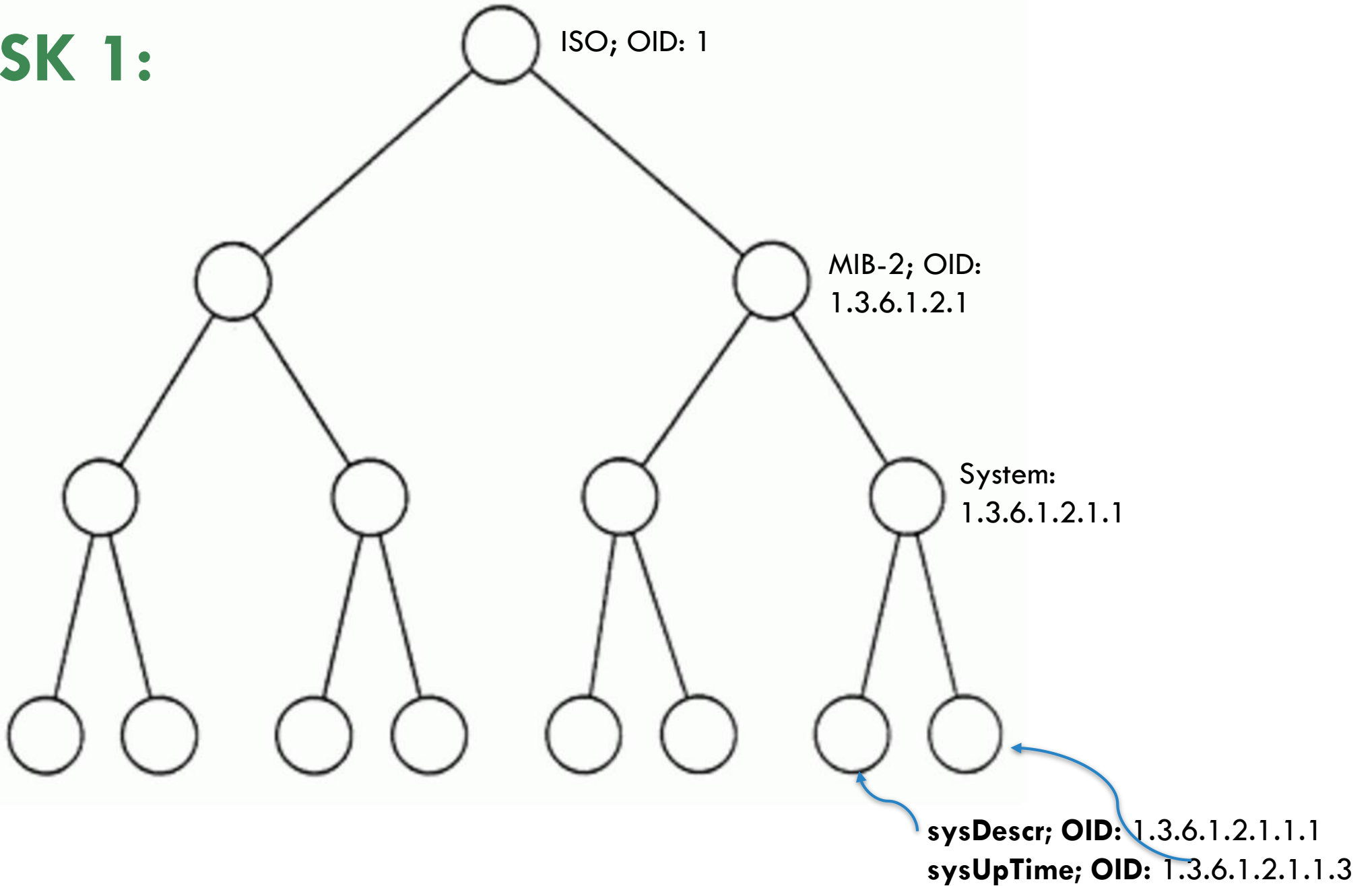
## Potrzebne Narzędzia:

- silnik wyrażeń regularnych np. REGEX
- język programowania np. COBOL
- plik MIB-II – *jako dane wejściowe do testów*

## Kompromisy na które pozwalamy:

- brak natywnego, pełnego, wsparcie notacji ASN.1
- to co szczegółowe – ‚hardkodujemy’ – ma działać

# TASK 1:



# REASUMUJĄC CO PARSER POTRZEBUJE PARSOWAĆ

1) Importy – deklaracje tego co poniżej z innych plików

Proponuje rekurencyjnie przeszukiwać pliki importów, a każdy plik traktować 'jednakowo'

2) deklaracje „OBJECT IDENTIFIER” – wprowadza nowe OIDy

3) deklaracje „OBJECT TYPE” – nowe obiekty zarządzalne

4) definicje nowych typów danych

## Deklaracje OIDów

OBJECT IDENTIFIER ::= { internet 2 }

# CO ZNAJDZIEMY W IMPORTACH:

Inne importy

IMPORTS

```
mgmt, NetworkAddress, IpAddress, Counter, Gauge,  
    TimeTicks  
    FROM RFC1155-SMI  
OBJECT-TYPE  
    FROM RFC-1212;
```

mgmt

## Definicje nowych typów danych:

```
Gauge ::=  
    [APPLICATION 2]  
    IMPLICIT INTEGER (0..4294967295)
```

```
TimeTicks ::=  
    [APPLICATION 3]  
    IMPLICIT INTEGER (0..4294967295)
```

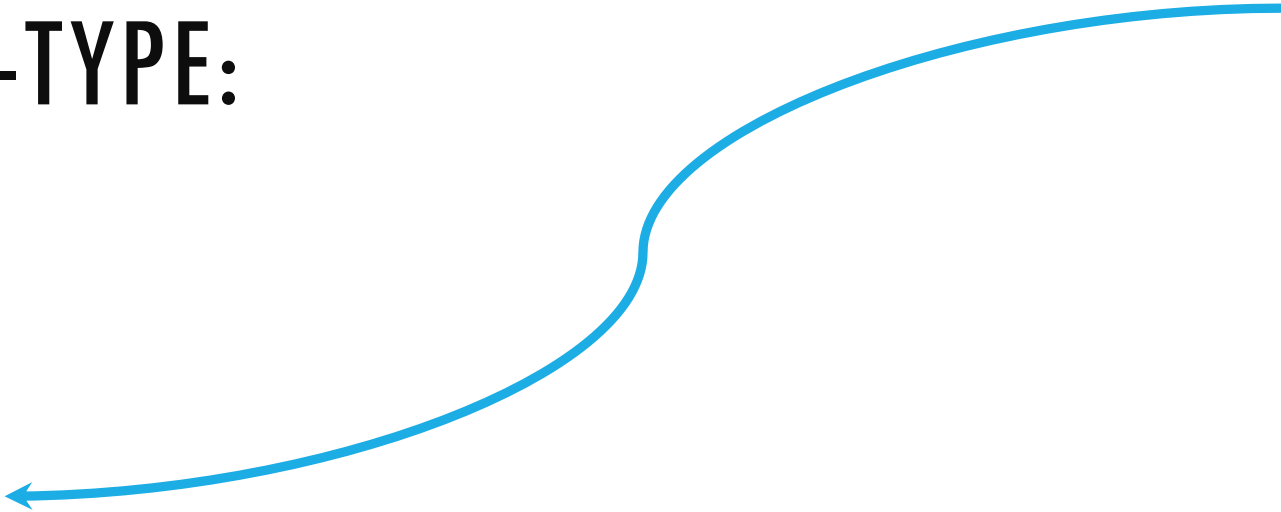
Zapisujemy w nodzie drzewa:

- Typ danych – Integer
- Zapisujemy ograniczenia: (0...429467295)
- Zapisujemy sposób kodowania:  
 typ: Implicit  
 wartość: 3
- Zapisujemy nazwę

# OBJECT-TYPE:

Zapisujemy:

- TYP-DANYCH
- Uprawnienia
- OPIS



```
sysUpTime OBJECT-TYPE
    SYNTAX  TimeTicks
    ACCESS  read-only
    STATUS  mandatory
    DESCRIPTION
        "The time (in hundredths of a second) since the
        network management portion of the system was last
        re-initialized."
    ::= { system 3 }
```

```
sysContact OBJECT-TYPE
    SYNTAX  DisplayString (SIZE (0..255))
    ACCESS  read-write
    STATUS  mandatory
    DESCRIPTION
        "The textual identification of the contact person
        for this managed node, together with information
        on how to contact this person."
    ::= { system 4 }
```



# UWAGI DOTYCZĄCE PARSOWANIA

- Kolejność wyszukiwania ma znaczenie. Np. być może najpierw warto sprawdzić importy, a na końcu makra OBJECT-TYPE
- W przypadku makra OBJECT-TYPE, zawiera ono pole DESCRIPTION. Które może zawierać praktycznie dowolne ciągi znaków. Dla tego, rozwiązaniem mogło by być na samym początku wyszukanie całych makr OBJECT-TYPE w strumieniu wejściowym S1, wycięcie ich do jakiegoś bufora B1 zawierającego tylko te struktury. Na pozostałości ciągu S1 (po wycięciu makr) wyszukiwanie innych elementów (takich jak definicje typów danych, importy, deklaracje identyfikatorów)

Dzięki temu unikniemy sytuacji w której np. deklaracje identyfikatora zawiera się w polu DESCRIPTION makra OBJECT-TYPE.

# PROPAGANDA


Po co to robimy?

- wszystkie informacje odnośnie typów danych, przydadzą się do sprawdzania czy z sieci otrzymujemy wartości w prawidłowym formacie; w prawidłowym dozwolonym przedziale wartości
- informacje odnośnie kodowania w typie zmiennej (APPLICATION, CONTEXT-SPECIFIC itp. Oraz identyfikator typu) będą przydatne w czasie kodowania danych do przesłania w formacie niezależnym od platformy (kodowanie BER)
- Sami sobie dajemy wycisk z danego języka programowania
- Wyrażenia regularne są bardzo często stosowane
- Możemy się poczuć jak twórca parsera języka programowania





# KODOWANIE BER CCITT X.209



Koder / dekodek tworzymy na podstawie tej prezentacji oraz powyższego dokumentu.  
Nie na podstawie innych informacji zawartych w Internecie.

# CZEMU KODOWANIE BER?

Na urządzeniach może być wykorzystywane kodowanie little-endian, big-endian, bądź też inne.

Dla tego też, wprowadzamy kolejną warstwę abstrakcji, po to aby uniezależnić reprezentacje danych od platformy sprzętowej.

# TYPY KODOWAŃ UŻYWANY W ASN.1

BER (Basic Encoding Rules) – najwięcej możliwości, mamy dostępne różne tryby kodowania

CER (Canonical Encoding Rules) - koduje dane o niekreślonej długości

DER (Distinguished Encoding Rules) – koduje dane o z góry określonej długości

# POJĘCIA

**Primitive Encoding** – bajty w polu Zawartość reprezentują pojedynczą wartość /jedną konkretną zmienną

**Constructed Encoding** – bajty zawarte w polu Zawartość reprezentują jedną lub kilka niezależnych wartości.

# TYPY DANYCH

- Typy proste

To takie które nie składają się z innych składowych

- Typy złożone

To takie które zawierają kilka komponentów:  
SEQUENCE, SEQUENCE OF, ~~SET, SET OF~~

Nie obsługiwane w SMI

Przykładowe typy proste z SMI (inne ASN.1 wykreślone):

Type	Tag number (decimal)	Tag number (hexadecimal)
INTEGER	2	02
BIT STRING	3	03
OCTET STRING	4	04
NULL	5	05
OBJECT IDENTIFIER	6	06
SEQUENCE and SEQUENCE OF	16	10
<del>SET and SET OF</del>	<del>17</del>	<del>11</del>
<del>PrintableString</del>	<del>19</del>	<del>13</del>
<del>T61String</del>	<del>20</del>	<del>14</del>
<del>IA5String</del>	<del>22</del>	<del>16</del>
<del>UTCTime</del>	<del>23</del>	<del>17</del>

# TYPY PROSTE C.D

**Typy proste dzielą się na dwie kategorie:**

- Typy łańcuchowe (string)

~~BIT STRING, IA5String, OCTET STRING, PrintableString, T61String, and UTCTime~~

- Typy nie-łańcuchowe

Możemy wysyłać OCTET STRING przy użyciu kodowania o nieznanej długości, jako typ złożony (Constructed)



# Ogólna struktura kodowania BER

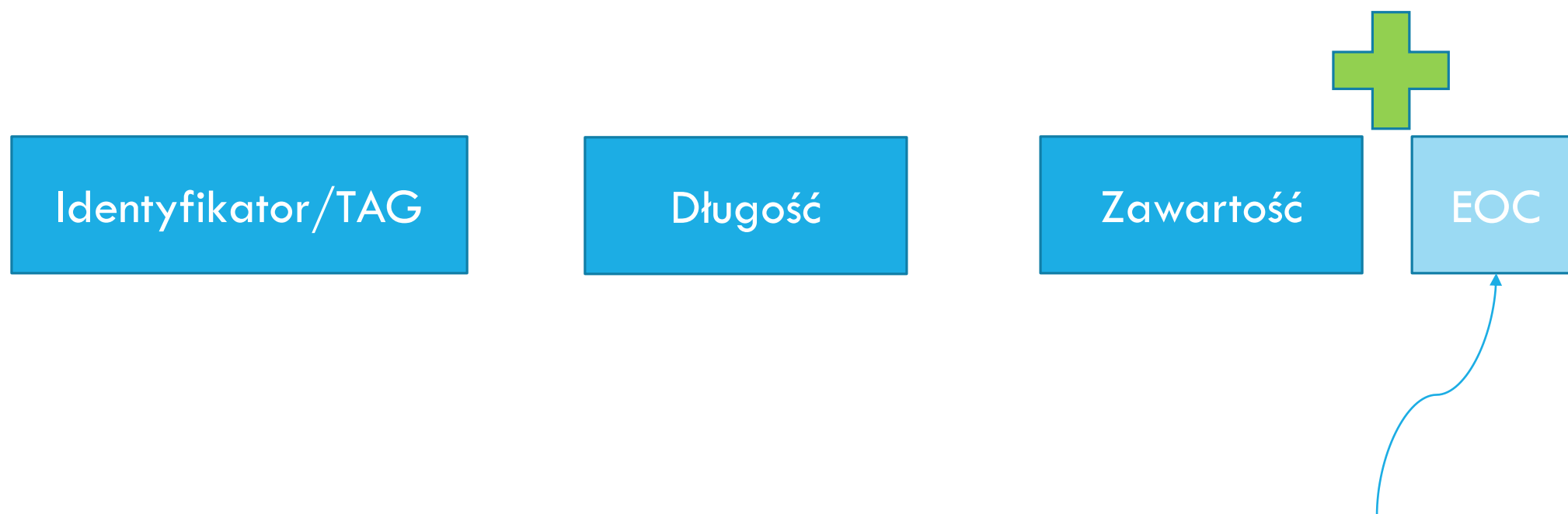
Typowe kodowanie Type-Length-Value

Identyfikator/TAG

Długość

Zawartość

# Aaaale...



W przypadku formy o nieokreślonej długości

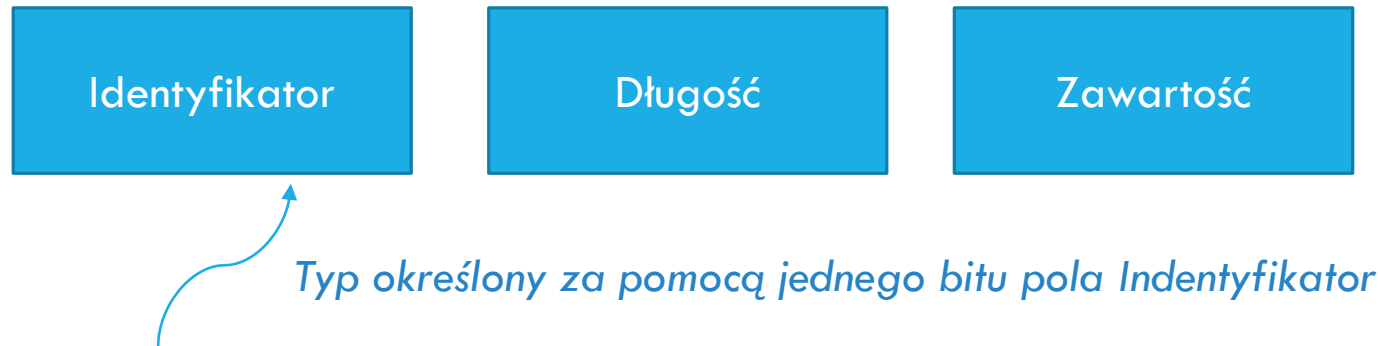
# aaaaale

Kiedy nie ma co zakodować. Jak np. w przypadku typu NULL



W przypadku formy o nieokreślonej długości

# TYPY ZAPISU WARTOŚCI



1. Zapis prosty (Primitive), o określonej długości
  - Do kodowania prostych typów danych
2. Zapis złożony (Constructed), o określonej długości danych  
np. dla SEQUENCE, SEQUENCE [OF], SET
3. Zapis złożony, o nieokreślonej długości  
np. dla SEQUENCE, SEQUENCE [OF], SET

# POLE IDENTYFIKATORA

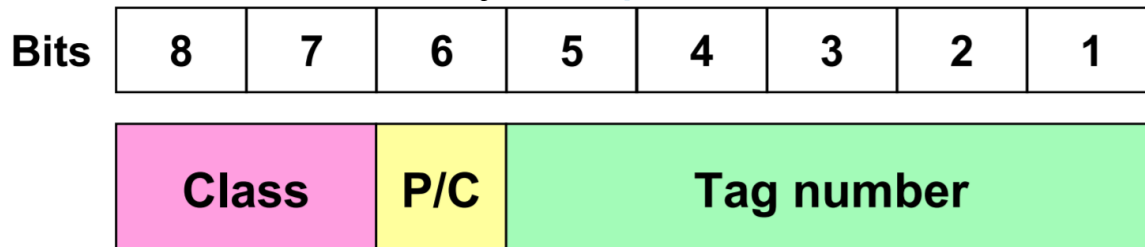
Każdy posiada TAG.

**TAG Składa się z :**

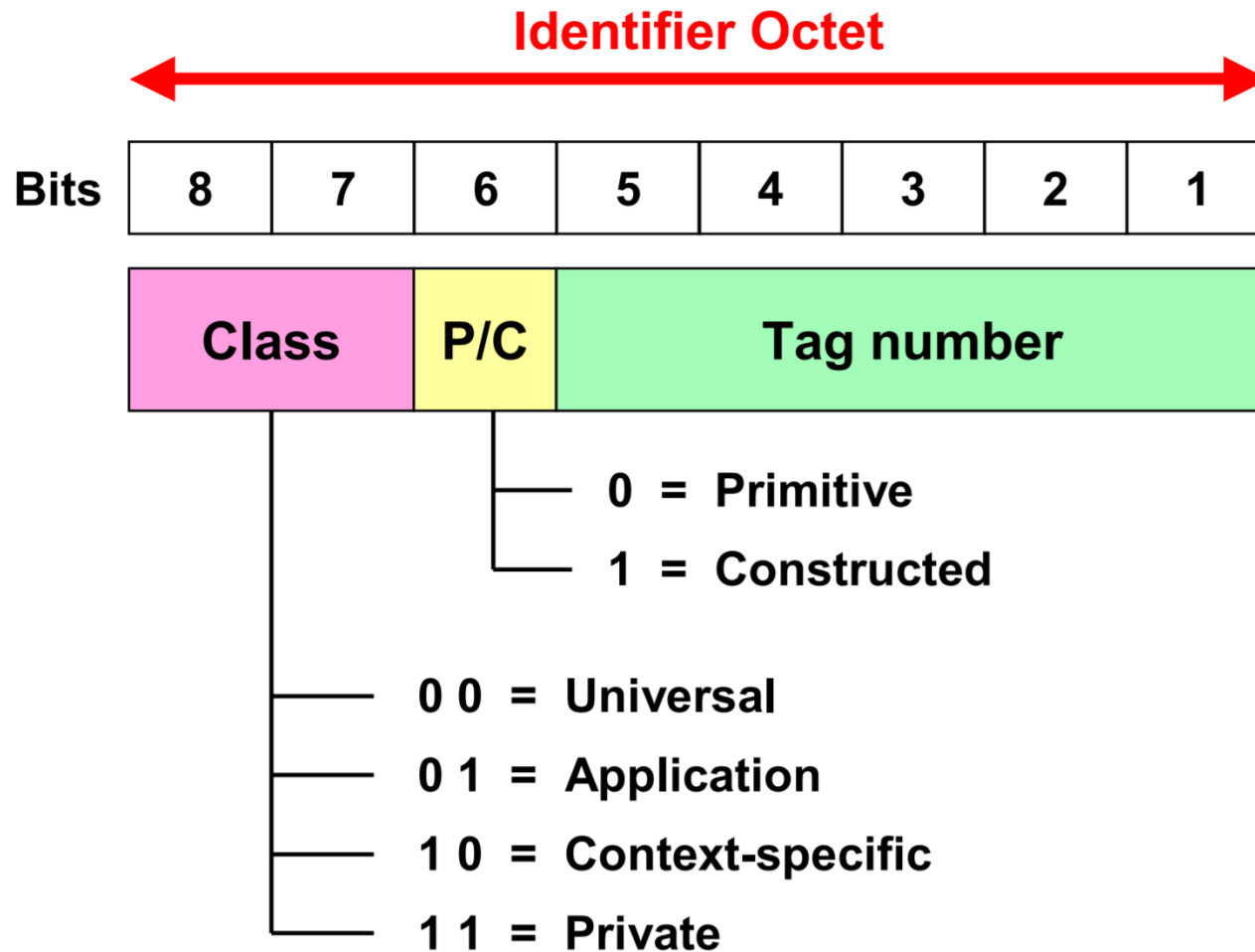
- identyfikatora klasy
- primitive czy constructed
- nieujemnego identyfikatora

**Klasy Tagów:**

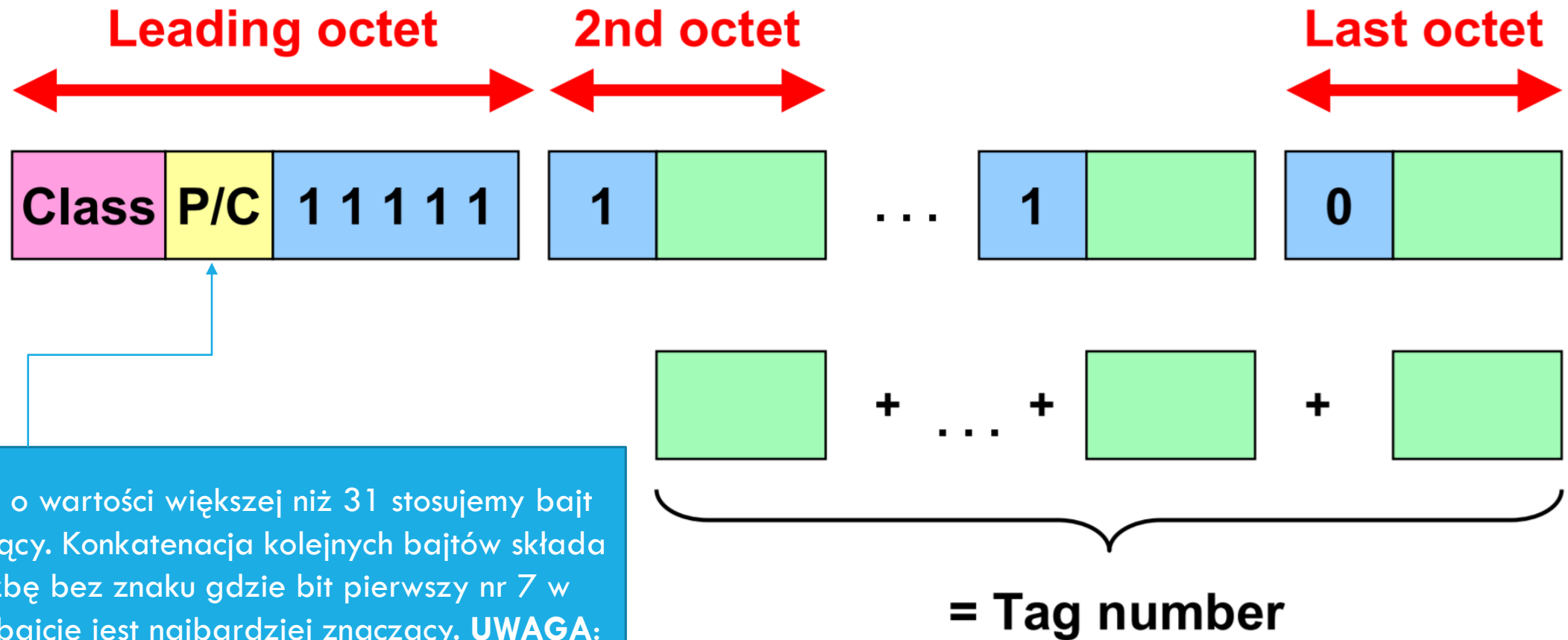
- uniwersalne – zdefiniowane w X.208: 00b
- application-specific : 01b
- prywatne: 10b



# KODOWANIE DLA TAGÓW $< 31$ (POLE IDENTYFIKATORA)



# DLA TAGÓW $> 31$ (POLE IDENTYFIKATORA)



Dla tag'ów o wartości większej niż 31 stosujemy bajt poprzedzający. Konkatenacja kolejnych bajtów składa się na liczbę bez znaku gdzie bit pierwszy nr 7 w pierwszym bajcie jest najbardziej znaczący. **UWAGA:** pierwszy bit w tych bajtach zarezerwowany jest na oznaczanie ostatniego/lub nie; bajtu.

Pierwszy następujący bajt nie może być cały = 0

# KODOWANIE TYPU CHOICE

Typ CHOICE kodujemy używając taga typu danych który rzeczywiście jest przesyłany.



# KODOWANIE TYPU BOOLEAN

- Jeśli wartość TRUE => bity pola WARTOŚĆ różne od zera
- Jeśli wartość FALSE => wszystkie bity ustawione na zero

Używamy kodowania prymitywnego.

# KODOWANIE INTEGER'A

- Używamy kodowania prymitywnego.
- Pole ZAWARTOŚĆ powinno składać się z jednego bądź też wielu bajtów

# KODOWANIE POLA DŁUGOŚĆ (LENGTH)

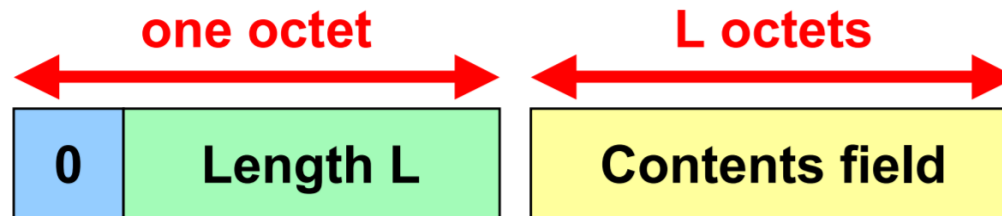
- Krótka forma o określonej ilości ( $< 128$  bajtów)
- Długa forma o określonej ilości ( $128 < \text{ilość danych} < 2^{1008}$ )
  - Nadawca używa reprezentacji o określonej długości jeśli typ danych jest prymitywny
  - Używa określonego lub nie określonego jeśli wszystkie dane do wysłania ma dostępne a typ danych jest złożony.
- Format nieokreślony; koniec danych określony przez EOC
  - Nadawca używa tego typu kodowania jeśli dane nie są od razu dostępne i typ danych jest złożony.

# KODOWANIE POLA *LENGTH*

## UWAGA:

K nie może być równe 127

- Krótka forma o określonej ilości ( $< 128$  bajtów)



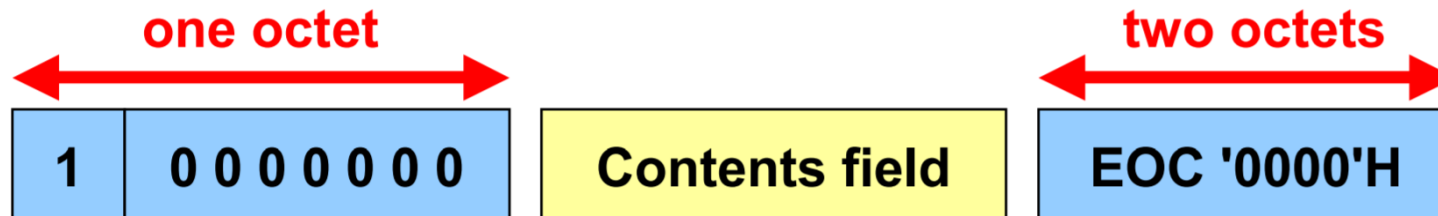
Typy proste lub złożone  $< 128$  bajtów

- Długa forma o określonej ilości ( $128 < \text{ilość danych} < 2^{1008}$ )



Typy proste lub złożone  $> 128$  bajtów

- Format nieokreślony; koniec danych określony przez EOC



Tylko typy złożone

# PYTANIA KONTROLNE

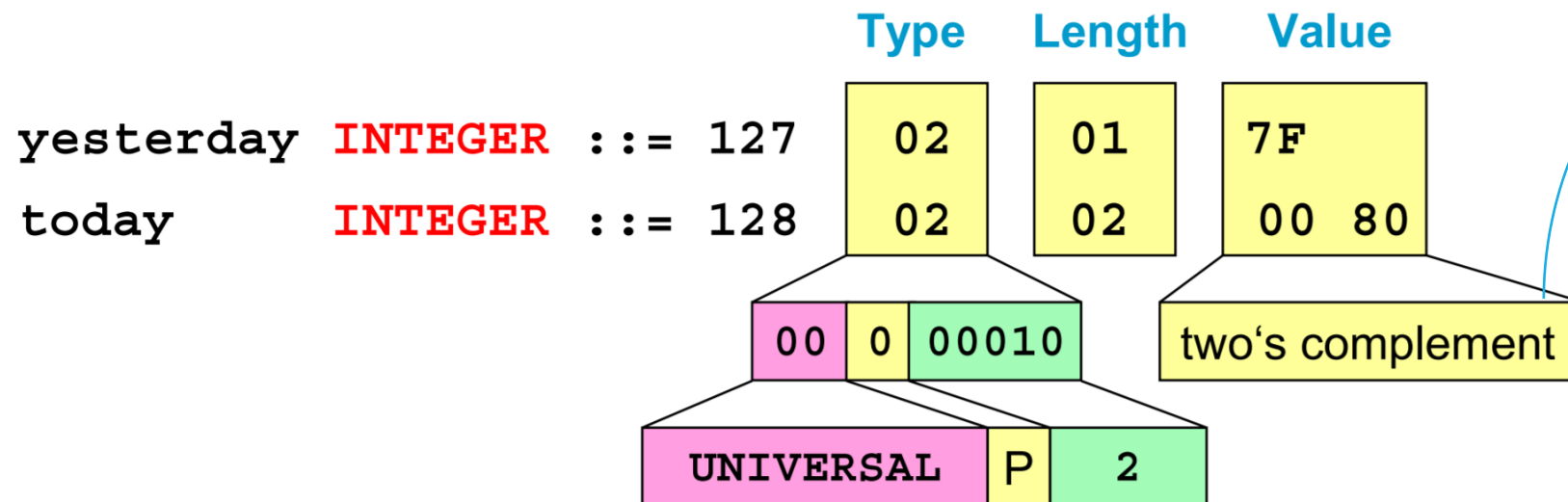
- jak zakodujemy LENGTH typu prostego jeśli długość wynosi 12?
- Jak zakodujemy pole LENGTH typu złożonego jeśli długość wynosi 15?
- Jak zakodujemy pole LENGTH typu złożonego jeśli długość wynosi 215?

# KODOWANIE POLA LENGTH UJĘCIE 2

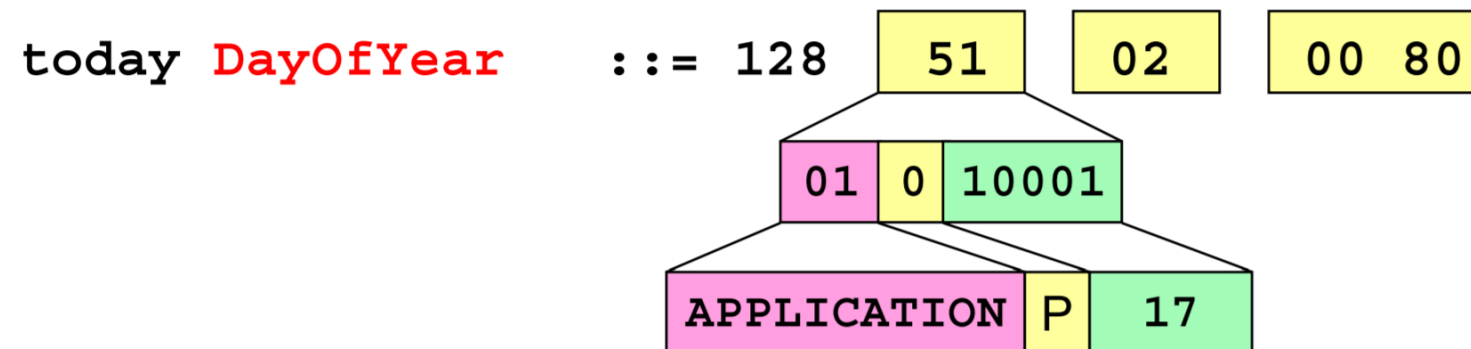


Form	Bits							
	8	7	6	5	4	3	2	1
Definite, short	0	Length (0–127)						
Indefinite	1	0						
Definite, long	1	Number of following octets (1–126)						
Reserved	1	127						

# PRZYKŁAD



DayOfYear ::= [APPLICATION 17] IMPLICIT INTEGER



## PRZYKŁAD 2 – SEQUENCE

```

Birthday ::= SEQUENCE {
    name    VisibleString,
    day     DayOfYear
}

```

Type Definition

UNIVERSAL 16  
00 1 10000

```

myBirthday Birthday ::= {
    name    "Jane",
    day     128
}

```

Value Assignment

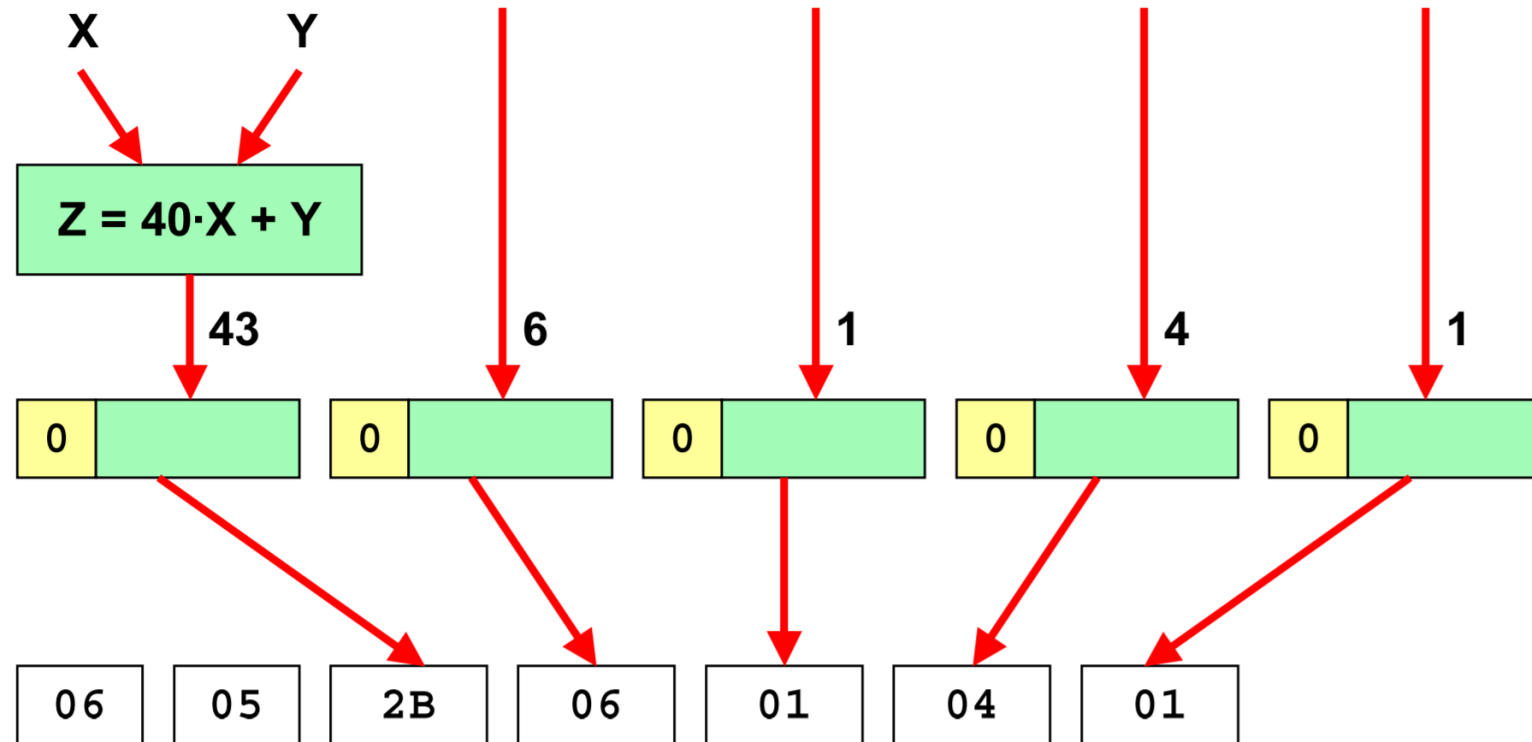
Birthday Length Contents		BER Encoding	
30	0A		
	VisibleString	Length	Contents
	1A	04	"Jane"
	DayOfYear	Length	Contents
	51	02	00 80



# KODOWANIE OBJECT-IDENTIFIER

enterprise OBJECT IDENTIFIER ::=

{iso(1) org(3) dod(6) internet(1) private(4) 1}



# Z PLIKU RFC1155, KTÓRE POTRZEBUJEMY OBSŁUGIWAĆ

- IpAddress ::= [APPLICATION 0] IMPLICIT OCTET STRING (size 4)
- Counter ::= [APPLICATION 1] IMPLICIT INTEGER (0..4294967295)
- Gauge ::= [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)

(..)NetworkAddress, IpAddress, Counter (..)

# PRZYKŁADOWE DANE ZAKODOWANE W BER

- 02, 02, FF 7F (INTEGER, -1 29)
- 04, 04, 01 02 03 04 (OCTET STRING, – wartość to: 01020304)
- 05 00 (NULL)
- 1A 05 4A 6F T3 65 73 (ciąg znaków “Jones” )
- 30 06, 02 01 03, 02 01 08 (sekwencja dwóch liczb INTEGER)

# UNIVERSAL VS IMPLICIT VS EXPLICIT

Wartość Liczbowa = 5

A ::= INTEGER

**Zakodowane:** 02 01 05

B ::= [APPLICATION 4] IMPLICIT INTEGER

**Zakodowane:** 44 01 05

C ::= [APPLICATION 5] EXPLICIT INTEGER

**Zakodowane:** 65 03 02 01 05

1) Reprezentacje danych oznaczonych jako Explicit tworzą się kapsułkując reprezentacje potomka w polu DANE typu macierzystego. Tworząc tym samym typ złożony (constructed) np. o tylko jednej skapsułkowanej wartości. Tj w przykładzie.

2) Bit Constructed (6ty w polu TAG ustawiony na 1).

# UNIVERSAL VS IMPLICIT VS EXPLICIT C.D

B ::= [4] IMPLICIT INTEGER

**Zakodowane:** 84 01 05

C ::= [5] EXPLICIT INTEGER

**Zakodowane :** A5 03 02 01 05

Jeśli nie podamy widoczności typu (Klasy),  
dla IMPLICIT – domyślną kasą będzie  
Context-Specific

# DO TESTOWANIA

<http://asn1-playground.oss.com/>

# PRZYKŁADOWO

Schema: Enter manually ▼

```
World-Schema DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
C ::= [5]EXPLICIT INTEGER

END
```

Compile

Data: HEX text ▼

Decode ▶▶▶

## DATA: ENCODE

Enter a Value (in the ASN.1 Value Notation format) for one of the Types defined in the Schema. Click Encode. Various encoded formats will be available as links for downloading.

Value: JSON ▼

Type: C ▼

5

## CONSOLE OUT

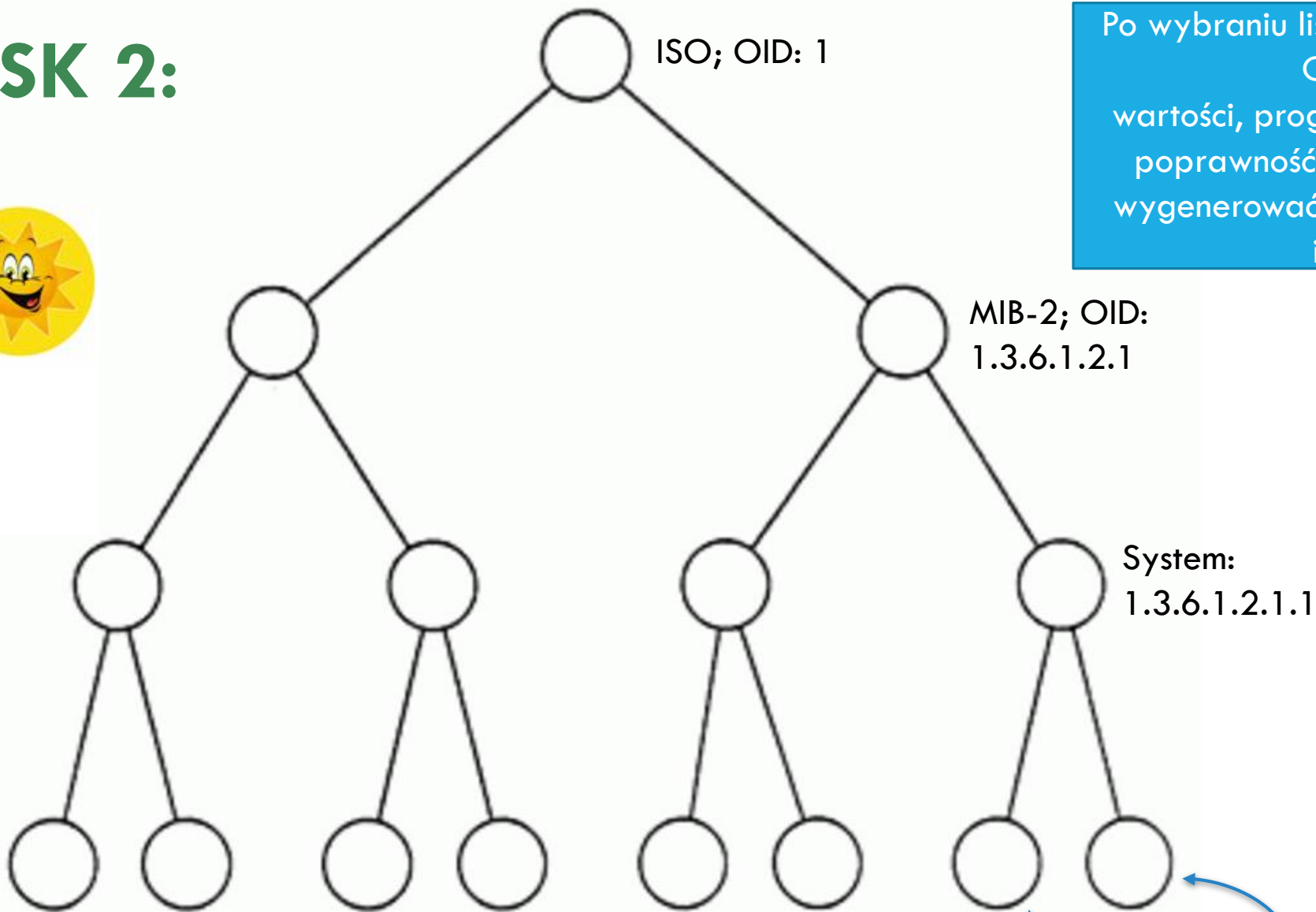
Options:

☐ Strict synt.

```
ASN1STEP: Encoding of
Encoding to the file '
tag = [5] constructed;
  C INTEGER: tag = [UN
    5
Encoded successfully i
A5030201 05
```

Dobrze zakodowaliśmy

## TASK 2:



Po wybraniu liścia w drzewie (za pomocą OID) i podaniu wartości, program potrafi zweryfikować poprawność podanej wartości, ORAZ wygenerować kodowanie BER dla danej instancji typu.

System:  
1.3.6.1.2.1.1

**sysDescr; OID: 1.3.6.1.2.1.1.1**  
**sysUpTime; OID: 1.3.6.1.2.1.1.3**

UWAGA: Maksymalnie można dostać trzy słoneczka.



## TASK 2:



Odczytujemy z nod'a drzewa:

Typ danych – Integer  
do sprawdzenia czy wartość podana jest prawidłowa

Sprawdzamy sposób kodowania oraz zapisany TAG:

typ: Implicit

wartość TAGU: 3, wykorzystamy podczas

kodowania

- Odczytujemy ograniczenia (0...429467295)

Wykorzystane do sprawdzenia czy wartość która chcemy wygenerować  
jest poprawna

- Odczytujemy nazwę

