Appunti delle lezioni di Gestione di Sistemi e Reti
2006-2007

Franco Sirovich

5. L'Informazione di Gestione in SNMP

Come in tutti i sistemi di gestione, la gestione SNMP è fondata su un database che contiene informazione sugli elementi da gestire: *Management Information BASE (MIB).* Ogni risorsa da gestire è rappresentata da un oggetto, e la collezione di questi oggetti costituisce la MIB.

Questa collezione di oggetti ha una struttura (organizzazione) che in SNMP è gerarchica (ad albero): gli oggetti sono "legati" fra di loro da una struttura ad albero. Ogni sistema da gestire (workstation, server, router, bridge, ...) mantiene (possiede ed espone) una MIB che riflette lo stato delle risorse che lo costituiscono. Tutte le operazioni e le funzioni di gestione sono in SNMP offerte tramite operazioni su questa collezione di dati. Le funzionalità di *monitor* sono offerte tramite operazioni di lettura dei valori degli oggetti della MIB. Le funzionalità di controllo si realizzano mediante la modifica di valori degli oggetti della MIB. In SNMP ogni funzionalità di gestione è realizzata mediante accessi in lettura o in scrittura su oggetti della MIB che descrive la risorsa gestita.

Perché una MIB sia funzionale alla gestione però deve soddisfare alcuni requisiti:

1. *Gli oggetti usati per rappresentare una certa risorsa devono essere gli stessi in ogni sistema dello stesso tipo.* Per esempio consideriamo la gestione di una entità TCP che esegue su una certa macchina. L?entità TCP è il "sistema" o la "risorsa" da gestire. Il numero totale di connessione aperte su quella entità è dato dalla somma del numero aperture passive e del numero delle aperture passive. È inutile (anzi nocivo) rappresentare nella MIB tutte e tre queste quantità, che non sono indipendenti: è meglio rappresentarne due e la terza si calcola di conseguenza. Ma quale coppia rappresentiamo? Occorre che ogni agente di gestione memorizzi la stessa coppia di numeri altrimenti l'applicazione per gestire l'entità TCP diventa inutilmente complessa per adattarsi alle (inutili) differenze.
   La specifica della MIB per TCP prescrive di memorizzare il numero di connessioni attive e il numero di passive.
2. *Si deve usare uno schema comune per rappresentare le informazioni: non basta la semantica per assicurare interoperabilità.* Seguendo l'esempio precedente, come rappresentare il numero di connessioni attive? Si può usare un intero, oppure una stringa di cifra decimali. Questo è un

---

1 Quest'opera è stata rilasciata sotto la licenza Creative Commons Attribuzione-Non commerciale-Non opere derivate 2.5 Italia. Per leggere una copia della licenza visita il sito web http://creativecommons.org/licenses/by-nc-nd/2.5/it/ o spedisci una lettera a Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

caso molto semplice, ma in generale occorre definire la rappresentazione (*sintassi*) dell'informazione la cui semantica è stata ben definita. Infine, una volta decisa la rappresentazione occorre definire la *codifica* che useremo per trasmettere questa rappresentazione sulla rete. Notare che non serve definire in che modo la sintassi è codificata nella memoria dell'agente e del manager: possono usare la codifica che desiderano: uno può rappresentare il numero in complemento a due e l'altro in binario senza segno, ma quando si scambiano questa informazione devono usare una ben precisa codifica per il trasferimento di questa informazione.

Per soddisfare il requisito (2) è stata definita la S*truttura della Informazione di Gestione* (*Structure of Management Information, SMI*), che definisce i modi con cui si dovranno specificare le sintassi delle varie informazioni di gestione e i modi con cui queste sintassi dovranno essere codificate per il trasferimento su rete.

Per soddisfare il requisito (1) viene specificato (usando la SMI) lo *schema* della MIB per ogni tipo di sistema/risorsa per il quale devono essere implementati degli agenti di gestione. Nello schema della MIB (che viene però sempre anche esso chiamato, ambiguamente, MIB) viene definita quale informazione deve essere implementata per quel sistema, che semantica ha, e ovviamente come deve essere rappresentata in termini di sintassi. Vediamo ora cosa prescrive la SMI.

5.1. La Struttura della Informazione di Gestione (SMI)

La SMI è uno standard di Internet, definito nell'RFC1155 e poi "migliorato" in RFC1212. Per la seconda versione di SNMP, SNMPv2, è stata fatta una ridefinizione della SMI, *SMIv2* (RFC1442)

La SMI specifica numerosi aspetti delle MIB:

- I tipi di dato che possono essere usati in una MIB
- Come le risorse descritte da una MIB possono essere rappresentate
- Come si può dare un nome alle risorse di una MIB

In sostanza, la SMI specifica come è ammesso fare le specifiche delle MIB che saranno necessarie per gestire le risorse di interesse. Saranno specificate (usando la SMI) tante MIB quanti sono i tipi di sistemi si vogliono gestire in modo standard. Naturalmente, i tipi di sistemi da gestire è andato cambiando nel tempo: con il diffondersi di SNMP sono aumentati i tipi di sistemi per cui sono state specificate le rispettive MIB standard. Se oggi non è stata specificata una MIB standard per un certo tipo di sistema, ad es. per gli acquari di pesci tropicali, ciò significa che ancora non è ritenuto necessario gestire da remoto tramite SNMP questo tipo di risorse.

SMI incoraggia la semplicità delle MIB, e quindi non permette di, o cerca di rendere difficile, definire MIB complesse da implementare. Vedremo una grande quantità di limitazioni della SMI, e sempre queste limitazioni sono state imposte con lo scopo di rendere semplici le MIB che possono essere definite con questa SMI.

La maggior semplificazione riguarda i tipi di dati ammessi nella definizione delle sintassi dei tipi di oggetti. Abbiamo sue solo generi di tipi di dati:

- *Scalari*: sono tipi di dati semplici, senza struttura
- *Tabelle* bidimensionali di scalari

In OSI invece sono ammessi tipi di dati strutturati anche molto complessi e parallelamente sofisticate operazioni di accesso ai dati.

SNMP punta sulla semplicità per due ordini di ragioni:

1. *Semplificare la implementazione degli agenti*. Questo obiettivo ha diversi aspetti positivi. In

primo luogo si riduce il costi della implementazione (e del mantenimento) del software che implementa l'agente di gestione, nonché delle estensione al software della risorsa da gestire necessarie perché l'agente di gestione possa offrire al manager le informazioni richieste dalla MIB. In secondo luogo, la riduzione del software aggiuntivo (dell'agente e della risorsa stessa) necessario alla gestione riduce il costo della sua esecuzione, e permette la realizzazione del prodotto con costi, di CPU e di memoria primaria, inferiori rispetto a quelli implicati da una gestione sofisticata.

2. *Aumentare la interoperabilità*. Quanto più è complessa la MIB di una risorsa tanto maggiori saranno le ragioni (volute o non volute) per introdurre differenze fra una implementazione ed un'altra che riducono la interoperabilità fra le implementazioni. Gli standard che danno luogo a implementazioni che non interoperano sono "cattivi" standard: gli implementatori sono solo parzialmente colpevoli della mancanza di interoperabilità.

Per ottenere questi risultati occorre:

1. Fornire una tecnica standardizzata per definire la *struttura* di ogni MIB particolare
2. Fornire una tecnica standardizzata per *definire i singoli oggetti* che fanno parte di una MIB, incluso il tipo (sintassi) del loro valore
3. Fornire una tecnica standardizzata per *codificare i valori* degli oggetti, in modo che tali valori possano essere scambiati fra entità che implementano SNMP

Sono disponibili molti strumenti ottenere obiettivi simili a (1) e (2), ma nel caso dei protocolli applicativi è necessario raggiungere questi obiettivi senza dare la preferenza ad alcuna caratteristica particolare che i vari costruttori possono mettere in campo. Infatti, uno standard multivendor, come sono Internet o OSI, devono mettere sullo stesso piano i diversi costruttori, senza fare, o essere accusati di fare, favoritismi di sorta, perché in tal caso forniscono l'albi ai costruttori per rifiutare lo standard multivendor o sono "bocciati" su tali basi negli organismi che ratificano gli standard. Quindi nel fare un o standard multivendor non bisogna mai fare riferimento a:

- *Architettura hardware usata dalle implementazioni*: molti linguaggi di programmazione ad alto livello (ad es. il FORTRAN) sono in realtà stati progettati avendo in mente una particolare architettura hardware, per la quale sono ottimi, mentre invece non sfruttano o mettono in difficoltà architetture diverse.
- *Sistema operativo*: i diversi sistemi operativi sono ottimizzati nel trattare certe strutture di dati, oppure per offrire certi servizi alle applicazioni o ai moduli interni addizionali che vengono sviluppati (ricordate che IP e TCP devono essere implementati all'interno del sistema operativo per essere efficienti); occorre non fare preferenze per alcuni e mettere tutti sullo stesso piano.
- *Linguaggio di programmazione*: i linguaggi di programmazione possono incorporare visioni dell'hardware e del sistema operativo che rendono difficile la implementazione di tali linguaggi su diverse architetture o sistemi operativi. Uno standard multivendor e la SMI non devono fare lo stesso "errore" ed essere neutrali verso il linguaggio di programmazione da usare nelle implementazioni, perché spesso il linguaggio di programmazione da usare è dettato dall'architettura hardware e dal sistema operativo che si deve usare nella implementazione.
- Scelte implementative da lasciare all'implementatore: qui veramente tocchiamo il cuore del problema di chi fa gli standard, in particolare quelli multivendor. Chi produce uno standard deve lasciare il massimo spazio possibile alla genialità e alla inventiva degli implementatori, perché in tale modo il "suo" standard avrà successo. Se la specifica lascia la massima libertà agli implementatori, ci saranno tante implementazioni, ognuna delle quali cercherà di sfruttare qualche idea geniale o punto di forza degli implementatori, e queste implementazioni, offerte sul mercato determineranno il successo commerciale dello standard.

Quale è la massima libertà da concedere? Quella che non pregiudica la interoperabilità fra diverse implementazioni dello standard, ma un vincolo in più, non necessario per la interoperabilità, anche se dettato dalle migliori intenzioni ha un effetto negativo sulla accettazione dello standard da parte degli implementatori. È vero che uno standard multivendor è sempre imposto dagli utilizzatori e subito dagli implementatori, ma se gli implementatori si rifiutano di implementarlo gli utilizzatori possono fare poco. Fare uno standard multivendor è quindi un gioco sottile fra minacce (gli utilizzatori te lo imporranno) e promesse (potrai fare una implementazione così buona e così a buon mercato che conquisterai lo stesso il mercato anche se lo standard non è tuo proprietario).

Quindi sui punti (1) e (2) il gioco è quello di dare la massima libertà possibile. Invece, nell'ottenere l'obiettivo (3) occorre invece essere precisi e vincolanti a livello di singolo byte. Si potrebbe allora specificare i punti (1) e (2) in termini di codifica perché tanto alla fine con la codifica si finisce per fare i conti! Ma questo approccio produrrebbe specifiche poco comprensibili, difficili e costose da implementare e alla fine con poca interoperabilità.

La soluzione a queste esigenze contrastanti è di adottare un linguaggio di specifica per esprimere (1) e (2), e delle regole per codificare (trasmettere) quanto definito usando il linguaggio di specifica. In questo modo possiamo esprimere semantica e sintassi in modo astratto, comprensibile e promuovendo implementazioni che sfruttino al massimo tutte la particolarità dell'ambiente di implementazione (incluso il programmatore!). Fornendo poi delle regole di codifica riusciremo a fissare il dettagli della codifica in modo che anche su questo punto finale le implementazioni siano in grado di interoperare. Questa meraviglia si apprezza dopo avere visto in profondità gli aspetti della soluzione: Il *linguaggio di specifica ASN.1* e le *regole di codifica BER*.

Appendice B: Abstract Syntax Notation One – ASN.1

ASN.1 è un linguaggio formale standardizzato da *CCITT* (oggi *ITU*) con la sigla X.208 e da ISO con la sigla ISO 8824. La sigla "1" nel suo acronimo sta a indicare che si pensava di dover definire altri linguaggi formali per gli stessi scopi, ma oramai è molto accettato ed è improbabile che ne verranno definiti altri. L'ASN.1 risponde alle seguenti esigenze:

1. È un linguaggio formale per definire la *sintassi astratta* (il *formato*) dei dati *utilizzati* da applicazioni distribuite: senza tale definizione, non si può definire la elaborazione che devono compiere le entità di protocollo. Come sempre un protocollo è costituito dalla definizione dei dati scambiati dalle entità e dalla elaborazione che conducono alla spedizione di, e alla risposta a, tali dati scambiati

2. ASN.1 serve (anche) per definire la *struttura delle PDU* delle applicazioni, cioè il formato dei dati *affidati al trasporto*, perché le entità applicative devono comunicare i dati elaborati.

Ovviamente, ASN.1 è usato per definire la struttura della Management Information, sia in SNMP che nell'OSI Management Information Service (l'equivalente di SNMP in OSI), perché i dati scambiati sono dati che fanno parte di una MIB e la MIB deve essere specificata usando un qualche linguaggio.

Come si vede, al centro della definizione di ASN.1 è l'esigenza di servire come supporto per la definizione di una sintassi astratta. Ma cosa è una sintassi astratta?

B.1 Sintassi Astratta

Definiamo alcuni termini importanti.

Sintassi astratta (*abstract syntax*) è la descrizione della *struttura dei dati*, usati da un protocollo, indipendentemente da ogni *tecnica di codifica* o *rappresentazione* di tale informazione su un supporto di memoria (primaria o secondaria), o su uno schermo, o su carta.

Si vuole allora disporre di un linguaggio formale per poter specificare i tipi dei dati e indicare (denotare) valori particolari appartenenti a questi tipi. Per *tipo di dato* intendiamo un *insieme di valori*

dotato di *nome*. Il tipo di dato può essere *semplice*, definito specificando i valori che appartengono all'insieme; oppure *strutturato*, definito in termini di altri tipi di dato con una *operazione/relazione* di *composizione* di tali tipi.

Per *codifica* intendiamo la *sequenza di ottetti* che viene usata per rappresentare il valore appartenente ad un tipo di dato. Possiamo avere diverse codifiche, utili in diverse circostanze. Ad esempio, ci interessa certamente la codifica (rappresentazione) della informazione su uno schermo: in tal caso la sequenza di ottetti sarà costituita da ottetti che rappresentano una serie di caratteri stampabili (o digitabili su una tastiera) che siano comprensibili per un operatore. Ci interessa la rappresentazione (codifica) in memoria primaria, altrimenti non potremmo elaborare questi dati; ci interessa la codifica su disco, altrimenti non potremmo memorizzare tali dati per lungo periodo. Una particolare codifica che ci interessa e la seguente, che usiamo quando vogliamo comunicare questi dati ad una applicazione paritaria. *Sintassi di trasferimento* è la generica struttura con cui i dati sono *codificati in termini di ottetti* nel trasferimento fra due entità (applicative) paritarie.

Quando si definisce uno standard occorre definire sia la sintassi astratta che la sintassi di trasferimento. Nulla eviterebbe di intraprendere la definizione della sintassi di trasferimento, una volta definita quella astratta, "inventandosi" una codifica specializzata protocollo per protocollo o addirittura versione per versione dello stesso protocollo. Ma è certamente molto interessante fare invece uso di *regole di codifica* che specificano un particolare *mapping* (*traduzione*) fra sintassi astratta e sintassi di trasferimento. In tale modo, per definire la sintassi di trasferimento di una particolare sintassi astratta è sufficiente specificare quali regole di codifica si devono adotare. La sintassi di trasferimento risulta quella dettata dalla sintassi astratta e dalle regole di codifica specificate. Le regole di codifica devono semplicemente determinare in modo algoritmico per ogni valore che appartiene ad un sintassi astratta la sua codifica di trasferimento.

La architettura di comunicazione di una applicazione può essere pensata come costituita da due componenti essenziali illustrati nella figura che segue.

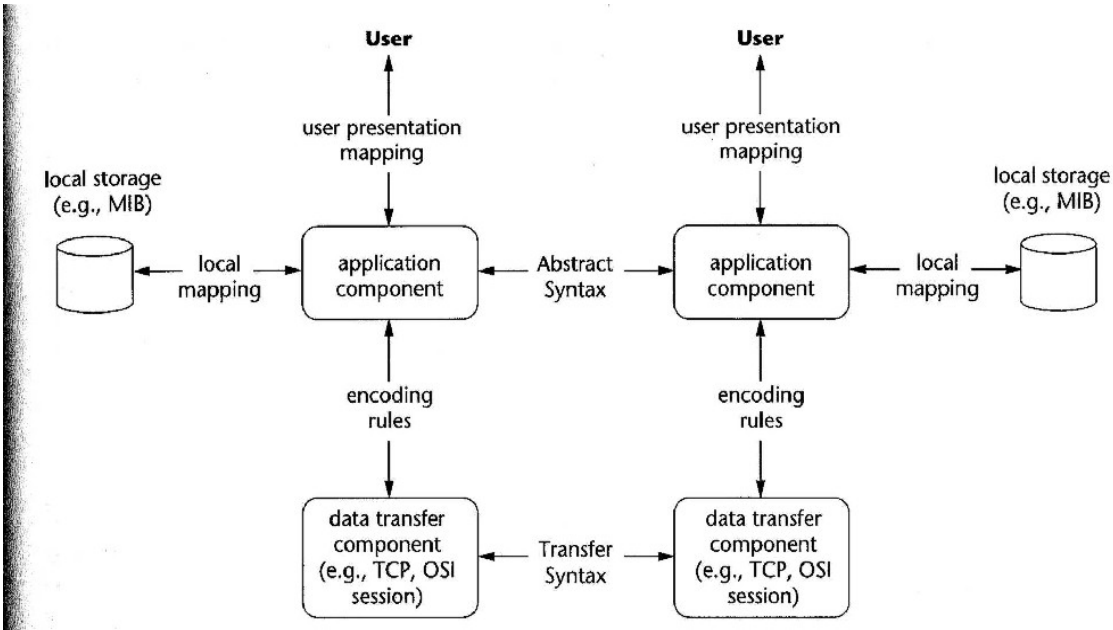


FIGURE B.1 The use of abstract and transfer syntaxes

Il *Data Transfer Component* è quel componente che si occupa del trasferimento dei dati fra i due end system. Questo componente è TCP o UDP, e gli strati sottostanti, in Internet. L'*Application Component*: utilizza il *data transfer component* e *realizza* l'applicazione che l'utente finale (umano) utilizza: ad es. SNMP, FTP, SMTP.

Quando passiamo dal data transfer component all'application component si ha un *cambiamento drastico* nel modo con cui sono considerati i dati:

- *Sequenze di ottetti*, al livello del data transfer component
- *Informazione dotata di tipo e di struttura*, al livello dell'application component

L'application component, dopo avere eseguito le sue elaborazioni, deve deve fornire una rappresentazione di questi dati strutturati che possa essere trasportata dal data transfer component, e interpretata correttamente dall'application component paritario. Questa rappresentazione, da affidare al data transfer component, viene spesso indicata con il nome di *presentation data*.

Il problema viene quindi affrontato nel modo seguente. L'application component tratta una informazione definita da una *sintassi astratta*, che la specifica in termini di *data type* e ove necessario di *data value*, nei casi in cui occorra specificare determinati comportamenti a fronte di determinati valori ricevuti o da inviare. Tale *sintassi* si chiama astratta perché specifica i dati indipendentemente dalla loro rappresentazione in un linguaggio di programmazione, in memoria o a livello di data transfer component. Quindi la notazione usata per descrivere una sintassi astratta (*ASN.1*) ha molte somiglianze con un linguaggio di programmazione che permetta di definire tipi di dato e costanti, o con notazioni che permettono di definire linguaggi di programmazione, come la Backus-Naur Form (BNF).

La sintassi astratta definita usando ASN.1 descrive, in modo formale ma astratto (slegato da una rappresentazione specifica) non solo *i dati trattati* dalle entità applicative paritarie, ma anche *i dati che possono essere scambiati* fra due applicazioni paritarie, perché i dati trattati sono un sovrainsieme di quelli che vengono scambiati. Quindi la sintassi astratta definisce le PDU che le applicazioni si scambiano.

Vi prego di notare al differenza di fondo fra l'ASN.1 e le sintassi astratte: L'ASN.1 è uno strumento, le sintassi astratte sono costruite usando tale strumento. Avremo bisogno di una sintassi astratta per ogni protocollo applicativo che specifichiamo usando ASN.1.

Se la specifica della sintassi di un protocollo viene fatta in modo astratto, all'interno di un sistema che implementa un protocollo queste informazioni, da inviare o ricevute, dovranno essere rappresentate, e in vari modi. Devono essere presentate in qualche modo all'utente umano perché le legga o le dia in input all'applicazione: in tal caso parliamo di *mapping della sintassi astratta verso l'utente*. Ovviamente tali informazioni devono essere memorizzate localmente all'interno del sistema, vuoi in memoria RAM che in memoria secondaria: parliamo allora di *mapping della sintassi astratta in uno o più formati locali di memorizzazione*. Infine, tali informazioni essere mappate in una forma adatta per il trasferimento, usando il data transfer component. Mentre i mapping precedenti sono una questione privata dell'implementatore, il mapping verso il data transfer component non è una questione privata di ciascuna entità paritaria perché i dati passati al data transfer component poi verranno ricevuti tali e quali dall'entità paritaria: quindi la forma di trasferimento deve essere specificata in modo *implementation independent*. Per questo parliamo anche di *sintassi di trasferimento*.

La sintassi astratta definisce la struttura e il significato delle interazioni "astratte" fra le due entità applicative paritarie, mentre lo scambio effettivo dei dati viene effettuato mediante la sintassi di trasferimento. Abbiamo in questo modo raggiunto tre obiettivi:

- C'è una definizione (una *specifica*) comune dei dati da elaborare fra sistemi differenti
- C'è una rappresentazione (*codifica*) comune dei dati da scambiare fra sistemi differenti
- La sintassi astratta permette all'implementatore di scegliere come preferisce una particolare rappresentazione (*sintassi concreta*) dei dati, in memoria, su disco, nell'interazione con l'utente

L'adozione del concetto di sintassi astratta, separata da quella di trasferimento, e a loro volta separate

da quelle concrete (dette anche *locali*), permette di raggiungere questi tre obiettivi.

Sarebbe però molto "noioso" che, dopo aver definito (usando ASN.1) la sintassi astratta delle PDU di un protocollo applicativo, si dovesse ancora fare la fatica (e commettere gli errori eventuali!!) di definire una sintassi di trasferimento per la sintassi astratta definita, cioè dover definire il formato delle PDU che devono "portare" i dati delle PDU astratte. È molto meglio che la sintassi di trasferimento sia automaticamente determinata da *regole di codifica*, basate sull'ASN.1, che prescrivano come convertire in sequenze di ottetti, sintassi di trasferimento, i vari tipi di dato definiti usando l'ASN.1. In questo modo si fa solo la fatica (e gli errori) di definire una sintassi astratta per un protocollo applicativo: le regole di codifica determinano automaticamente la sintassi di trasferimento.

Le prime regole di codifica per l'ASN.1 che sono state specificate da CCITT e ISO si chiamano *Basic Encoding Rules* (*BER*); ne sono poi state definite altre, ma la SMI di SNMP prescrive di utilizzare (un sottoinsieme del) le BER. Nei prossimi paragrafi descriveremo i concetti fondamentali dell'ASN.1 e delle BER.

B.2 Concetti di ASN.1

Il componente fondamentale di una specifica ASN.1 è il modulo. Una sintassi astratta può essere costituita da numerosi moduli che in vario modo fanno riferimento uno all'altro.

B.2.1 Definizione di un modulo

Un modulo ha questa forma

```
<modulereference> DEFINITIONS ::=
     BEGIN
          ExportList
          ImportList
          AssignmentList
     END
```

`<modulereference>` è il nome del modulo, usato per fare riferimento ad esso dall'interno di altri moduli. Il costrutto `ExportList` specifica la lista delle definizioni che vengono esportate dal modulo. Il costrutto `ImportList` specifica la lista delle definizioni che si devono importare da altri moduli, e da quale modulo vengono importate. Il costrutto `AssignmentList` specifica una lista di definizioni che costituiscono il vero e proprio corpo del modulo. Che genere di definizioni troviamo all'interno di un modulo ASN.1?

ASN.1 è un linguaggio progettato per permettere la definizione dei *tipi di dato* che sono necessari nella definizione di un protocollo applicativo. Nella definizione di un protocollo serve anche poter definire dei particolari *valori* (appartenenti e ben determinati tipi) a cui si deve fare riferimento nelle regole di comportamento.

Nella definizione dei tipi di dato è spesso necessario seguire dei *modelli* di definizione ben definiti e strutturati: l'ASN.1 permette di definire *macro*, e naturalmente *invocare macro* nella definizione di tipi di dato.

La `AssignmentList` contiene quindi definizioni di tipi, valori (o costanti) e macro.

L'ASN.1 è un linguaggio formale, che ha quindi una sua grammatica che può essere descritta formalmente (sul libro trovate una parte del definizione). Un modulo ASN.1 può essere analizzato e verificato da un compilatore ASN.1, che potrà generare il codice necessario per tradurre da una sintassi concreta (dettata dal compilatore) alla sintassi di trasferimento, e viceversa.

Le definizioni di tipi e valori hanno la forma

```
<name> ::= <description>
```

Daremo una definizione informale del linguaggio, a partire a alcune convenzioni lessicali.

B.2.2 Convenzioni lessicali

Il linguaggio non impone alcun layout: il separatore fra gli identificatori del linguaggio è il blank, ma molti blank, tab e linee bianche equivalgono ad un unico blank. I commenti sono delimitati da coppie di trattini (--) all'inizio e alla fine del commento; oppure dal primo fine-linea che segue la coppia di trattini che inizia il commento.

Gli *identificatori lessicali* (nomi simbolici di valori e campi), i *riferimenti a tipi* (nomi dei tipi) e i *nomi di moduli* consistono di lettere maiuscole e minuscole, cifre decimali e il trattino (-). Gli identificatori iniziano con lettera minuscola; I riferimenti a tipi e i nomi di modulo iniziano con lettera maiuscola; I tipi built-in (primitivi) e gli identificatori del linguaggio sono in lettere solo maiuscole; I nomi di macro sono il lettere solo maiuscole.

B.2.3 Tipi di Dato Astratti

ASN.1 è una notazione per definire tipi di dati astratti e loro valori (costanti). I tipi di dato possono essere visti come un *insieme di valori*; il numero di elementi di questi insiemi può anche essere infinito, perché non ci si preoccupa della loro effettiva rappresentabilità in una macchina. I tipi possono essere classificati in quattro categorie:

- *Semplici*: tipi i cui valori sono atomici in quanto non hanno componenti
- *Strutturati*: un valore di un tipo strutturato ha dei componenti e delle proprietà che "legano" fra loro questi componenti
- *Tagged*: sono sottotipi di altri, riconoscibili dai tipi "padre", come risulta da restrizioni sul tipo da cui derivano
- *Altri* (o *metatipi* come viene meglio definito in altri testi): quelli definiti con `CHOICE` o con `ANY`

Ogni tipo ASN.1 (con la sola eccezione di CHOICE e di ANY) ha per definizione associato un *tag*: un tag è costituito da un *nome di classe* e da un *valore intero non negativo*. Vi sono quattro possibili classi (nomi di classe):

- *Universal*: sono i tag dei tipi generalmente utili, e quindi al di fuori di una qualunque applicazione, e non relativi ad un contesto; tali tipi e i relativi valori del tag `UNIVERSAL` sono definiti nello standard che definisce ASN.1 (standard ISO e ITU, non un RFC!)
- *Application-wide*: sono tag che hanno validità all'interno solo di uno standard applicativo: quindi due standard applicativi diversi possono usare liberamente lo stesso valore di tag; sono definiti all'interno degli standard che definiscono il protocollo applicativo specifico, quindi troveremo delle definizioni di tag application wide nei documenti che definiscono SNMP
- *Context-specific*: hanno significato in un contesto ristretto, oltre che all'interno di uno standard applicativo; all'interno di due diversi contesti nello stesso protocollo applicativo possono assumere lo stesso valore; sono definiti all'interno degli standard che definiscono il protocollo applicativo specifico, quindi troveremo molti tag context specific nei documenti che definiscono SNMP
- *Private*: sono come quelli Universal, ma definiti da un vendor e che quindi solo lui sa cosa significano; non troveremo la definizione di tag private nei documenti che definiscono SNMP.

I tag devono essere usati nelle specifiche in modo che i valori del tipo siano univocamente identificati dal tag che essi portano con se. L'obiettivo dei tag è permettere il riconoscimento del tipo a cui appartiene un valore. Notate che la identificazione univoca è "aiutata"

- dalla conoscenza dello standard applicativo che è in uso (per gli application-wide), oppure

- del contesto all'interno del quale il valore occorre, oppure
- del vendor all'interno delle applicazioni del quale il valore viene scambiato

I valori dei tag UNIVERSAL sono descritti nella prossima figura:

Universal Tage Value	Type Name
1	BOOLEAN
2	INTEGER
3	BIT STRING
4	OCTET STRING
9	REAL
10	ENUMERATED
6	OBJECT IDENTIFIER
7	Object descriptor
18	NumericString
19	PrintableString
20	TeletexString
21	VideotexString
22	IA5String
25	GraphicString
26	VisibleString
27	GeneralString
5	NULL
8	EXTERNAL
23	UTCTime
24	GeneralizedTime
9-15	Reserved (ma il 9 è già usato)
28-	Reserved (dal 28 in su)
16	SEQUENCE e SEQUENCE OF
17	SET e SET OF

Non dovete "portarli all'esame"!

B.2.3.1. Tipi Semplici

I tipi semplici sono quelli che sono definiti semplicemente dall'insieme dei loro valori: sono in un certo senso atomici, in quanto i loro valori non hanno componenti. Il libro li suddivide in quattro "categorie".

Tipi di base

`BOOLEAN` è costituito da due valori `TRUE` e `FALSE`.

`INTEGER` è costituito da tutti i numeri *interi* della matematica (negativi, positivi e zero); quindi ha un numero infinito di valori: sarà molto difficile da trattare con completezza in una macchina finita.

`BIT STRING` è costituito da valori che consistono in stringhe di bit; ai singoli bit della stringa possono essere associati dei nomi per poterli riferire nella specifica del protocollo.

`OCTET STRING` è costituito da valori che consistono in stringhe di ottetti.

`REAL` è costituito da numeri espressi nella notazione scientifica (mantissa, base, esponente):

$$M * B^E$$

Mantissa `M` ed esponente `E` possono essere un qualunque intero (senza limiti) e la base `B` può essere 2 o 10.

`ENUMERATED` è costituito da un elenco esplicito di interi, ai quali si può associare un nome; ad es. possiamo definire un tipo nel seguente modo:

```
Version ::= ENUMERATED {    version-1 (0),
                            version-2 (1),
                            version-3 (2)
                        }
```

Anche con gli `INTEGER`, si può assegnare dei nomi a particolari valori dell'insieme. Il tipo `ENUMERATED` rende chiaro che i valori non sono interi su cui si possono eseguire operazioni aritmetiche

Il prossimo tipo è estremamente importante: `OBJECT IDENTIFIER` è un tipo i cui valori sono usati per identificare in modo univoco degli oggetti che ricorrono nei protocdli applicativi. Vogliamo che questi valori (*identificatori*) possano essere specificati all'interno degli standard di protocollo senza che ci siano ambiguità, cioè lo stesso valore venga per errore usato da due standard diversi.

Abbiamo quindi bisogno di amministrare lo spazio di questi valori, partizionandolo e assegnandone l'uso via via a gruppi di lavoro diversi. Quindi serve una struttura gerarchica: Un object identifier (spesso abbreviato con *OID*, cioè un valore che appartiene al tipo `OBJECT IDENTIFIER`) è una sequenza di valori interi non negativi. Si noti che, come avviene per le stringhe di bit e di ottetti, il tipo `OBJECT IDENTIFIER` è un tipo semplice perché i suoi valori non sono costituiti di componenti, anche se hanno una certa struttura.

Lo spazio degli object identifier è amministrato da ISO e ITU che se lo sono "spartito" in tre sotto-spazi (o sotto-alberi): il primo a ITU (il primo intero è 0) il secondo a ISO (il primo intero è 1) e il terzo a quei gruppi di standardizzazione in cui lavorano assieme ITU e ISO (il primo intero è 2). Ne parleremo molto perché SNMP usa moltissimo gli object identifier.

`Object descriptor` ha come valori delle semplici stringhe di testo che servono come annotazione da associare ad `OBJECT IDENTIFIER`.

ASN.1 definisce come tipi universali primitivi semplici anche una grande quantità di tipi di *stringhe di caratteri*. Un carattere ha una semantica molto diversa da quella di un ottetto, in quanto mentre una qualunque combinazione di 8 bit è un ottetto, un carattere è una particolare combinazione di 8 bit tale che:

- Vi corrisponde un particolare segno grafico (o di controllo del dispositivo di output), e
- Fa parte di un preciso sottoinsieme delle 256 possibili combinazioni di 8 bit.

Vi sono quindi molti tipi di stringhe di carattere (character set) che sono diverse per

- L'insieme dei segni grafici che li costituiscono, e/o
- La corrispondenza fra segni grafici e specifica combinazione di 8 bit.

Lo stesso segno grafico può essere "codificato" con diversi ottetti in diversi character set, e quindi apparire "sbagliato" sullo schermo o sulla carta quando viene interpretato da un sistema di output diverso da quello del sistema che ha effettuato la codifica.

`NULL` è un tipo particolare che *contiene un solo valore*, che viene chiamato (scritto) `null`. È quindi ancora più semplice del tipo `BOOLEAN` che almeno contiene due valori. Serve come "segna posto" per indicare che qualcosa è assente. Vedremo come lo usa SNMP.

EXTERNAL è un tipo costituito da un qualunque valore di un tipo definito usando ASN.1; viene definito quindi in uno standard diverso da quelb che usa il tipo EXTERNAL.

UTCTime e GeneralizedTime sono due differenti formati per esprimere data e ora in una stringa di caratteri, sia in tempo locale che in tempo "universale".

B.2.3.2. Tipi strutturati

Sono quei tipi i cui valori sono costituiti da *componenti* e da una *relazione* fra di essi: sono valori *strutturati*. Ci sono quattro costruttori di tipi strutturati:

- SEQUENCE
- SEQUENCE OF
- SET
- SET OF

SEQUENCE OF e SEQUENCE sono usati per definire tipi i cui valori sono costituiti da liste ordinate di elementi appartenenti a uno o più tipi di dati. SEQUENCE corrisponde al costruttore di record che si trova in molti linguaggi di programmazione, come la struct del C. Una sequenza consiste di una *lista ordinata* di elementi: ciascuno elemento può essere opzionale, appartiene ad un tipo specificato, può avere un valore di default, e può avere un nome, per poterlo riferire nello standard di protocollo.

La grammatica BNF dei costrutti SEQUENCE è la seguente.

```
SequenceType ::= SEQUENCE {ElementTypeList}|
                 SEQUENCE { }
ElementTypeList ::= ElementType |
           ElementTypeList , ElementType
ElementType ::=
     NamedType                 |
     NamedType OPTIONAL  |
     NamedType DEFAULT Value |
     COMPONENTS OF Type
```

NamedType è un riferimento ad un nome di tipo, con associato o meno un nome. Ogni definizione di elemento della sequenza può essere seguita dalla keyword OPTIONAL oppure DEFAULT. OPTIONAL indica che l'elemento può anche essere assente dalla sequenza. DEFAULT indica che se l'elemento non è presente nella lista allora deve essere assunto valere Value. COMPONENTS OF indica che tutti i componenti di Type, come compaiono nella definizione di Type, devono essere inclusi nella definizione della sequenza, nel punto in cui si trova COMPONENTS OF. Si noti COMPONENTS OF che non è un richiamo ricorsivo di SEQUENCE, ma una vera e propria inclusione di testo.

Invece SEQUENCE OF è apparentemente più semplice:

```
SequenceOfType ::= SEQUENCE OF Type | SEQUENCE
```

I valori del `SequenceOfType` sono costituiti da una lista ordinata (anche vuota) di elementi tutti del tipo `Type`. La notazione `SEQUENCE` è equivalente a `SEQUENCE OF ANY`.

`ANY` sarà descritto nel seguito.

Il costruttore `SET` e xx sono simili ai costruttori `SEQUENCE` e `SEQUENCE OF`, ma gli elementi della lista non devono essere considerati ordinati.

```
SetType ::= SET { ElementTypeList } | SET { }

SetOfType ::= SET OF Type | SET
```

A questo punto dobbiamo ricordare che questi costruttori di tipi di dato sono usati per definire tipi i cui valori devono essere comunicati fra applicazioni, usando un servizio di trasporto. Chi riceve un valore deve essere in grado di analizzarlo e identificare ogni singolo componente. Non ci devono essere ambiguità, ma queste sono introdotte dall'uso dei costrutti `OPTIONAL` e `DEFAULT`, e dalla mancanza di ordine del `SET`. Occorre qualche strumenti per eliminare l'ambiguità, e di nuovo ci vengono in aiuto i tag dell'ASN.1.

B.2.3.3. Tipi tagged

Il termine *tagged* è ingannevole perché in ASN.1 tutti tipi sono dotati di tag. Un *tipo tagged* è un tipo definito associando un nuovo tag ad un tipo già definito. Il tipo tagged è isomorfo al tipo usato nella definizione e a cui si applica il nuovo tag, ma è distinguibile da questo a causa della "aggiunta" del nuovo tag. Viene usato quando voglio distinguere un valore del tipo tagged da un valore del tipo base.

Un tipo tagged è in primo luogo utile a livello applicativo: in questo caso gli specificatori usano i *tag application wide* assegnando

- tag diversi a tipi diversi,
- all'interno dello stesso standard applicativo

quando sono interessati a distinguerli

- fra di loro, e
- rispetto ai tipi base.

All'interno di una SEQUENCE o di un SET si può verificare una ambiguità relativa al contesto: devo poter distinguere i componenti della sequenza o dell'insieme: in tal caso devo usare i *tag context specific*. Occorre usare tag diversi all'interno di un dato contesto: si possono usare gli stessi tag in contesti diversi.

Tagging esplicito e implicito

Quando uso un tag in una definizione posso desiderare che il nuovo tag sia aggiuntivo (esplicito) a quello del tipo a cui si applica, oppure sostitutivo (implicito). ASN.1 fornisce la keyword `IMPLICIT` che permette a chi usa ASN.1 di specificare che il nuovo tag è sostitutivo di quello vecchio. Il tagging implicito produrrà delle codifiche più compatte perché il vecchio tag non deve essere trasmesso, ma si ha una perdita di informazione (le cui conseguenze discuteremo più avanti).

B.2.3.4. `CHOICE` e `ANY`

Non sono veri e propri costruttori di tipo perché non hanno tag (come invece hanno sequenze e insiemi).

`CHOICE` permette di definire un tipo come una alternativa fra un certo numero di tipi indicati nella definizione. Chi usa un tipo definito mediante `CHOICE` potrà usare un valore che appartenga ad uno qualunque dei tipi elencati nella `CHOICE`. Chi riceve questo valore deve poter capire che scelta ha

fatto il mittente. Vi possono essere ambiguità fra le alternative: chi specifica un protocollo con ASN.1 deve usare i tag context specific per risolverle.

La grammatica per usare il tipo CHOICE è

```
ChoiceType ::= CHOICE { AlternativeTypes }

AlternativeTypes ::= NamedType | AlternativeTypes, NamedType
```

Invece per usare ANY è

```
AnyType ::= ANY
```

Il costruttore ANY è utile quando ancora non si sa quale tipo verrà usato in futuro, in altri standard o in evoluzioni dello standard che si sta specificando.

B.2.4. Sottotipi

Un sottotipo si definisce a partire dal tipo padre restringendo in qualche modo l'insieme dei valori che appartengono al sottotipo. Il tipo padre può essere a sua volta un sottotipo. Ci sono sei modi per restringere i valori.

B.2.4.1. Singolo valore

Si elencano esplicitamente i valori che appartengono al sottotipo; es.

```
SmallPrime ::= INTEGER ( 2 | 3 | 5 | 7 )
```

oppure

```
Months ::= ENUMERATED {
               january (1),
               february (2),
               march (3), april (4),
               may (5), june (6),
               july (7), august (8),
               september (9),
               october (10),
               november (11),
               december (12)
               }
First-quarter ::= Months ( january | february | march )
Second-quarter ::= Months ( april | may | june )
```

B.2.4.2. Sottotipo Contenuto

Il nuovo tipo viene definito elencando i tipi i cui valori sono tutti inclusi nel nuovo tipo

```
First-half ::= Months (
          INCLUDES First-quarter |
          INCLUDES Second-quarter )
First-third ::= Months ( INCLUDES First-quarter | april )
```

B.2.4.3. Intervallo di Valori

Si può usare solo per i sottotipi di INTEGER e REAL sui quali ha senso definire un intervallo di valori, indicando l'estremo inferiore e superiore, inclusi o meno: esempi equivalenti

```
PositiveInteger ::= INTEGER (0<..PLUS-INFINITY)
```

```
PositiveInteger ::= INTEGER (1..PLUS-INFINITY)
```

Le costanti MIN e MAX si usano per indicare il valore massimo definito per il tipo padre

```
NegativeInteger ::= INTEGER (MIN..<0)
```

```
NegativeInteger ::= INTEGER (MIN..-1)
```

B.2.4.4. Alfabeto Permesso

Si può usare solo per sottotipi di stringhe di ottetti che contengono codici caratteri: si elencano i caratteri che fanno parte del sottotipo.

```
StringaDiCifreDecimali ::= IA5String ( FROM
     "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" )
```

B.2.4.5. Vincolo sulla dimensione

Si può usare solo per tipi padre sui quali ha senso definire una "dimensione": stringhe, SET, SEQUENCE. Nel caso delle stringhe (bit, ottetti, caratteri) l'unità di misura è il bit, gli ottetti o il carattere rispettivamente. Nel caso di SET e SEQUENCE la dimensione è il numero di componenti

B.2.4.6. Vincolo di Sottotipo Interno

La definizione usando il vincolo di sottotipo interno si può applicare a tipi `SEQUENCE`, `SEQUENCE-OF`, `SET`, `SET-OF` e `CHOICE`. Un sottotipo interno (di un altro) si definisce specificando le condizioni a cui devono soddisfare i valori del tipo padre per essere parte del tipo figlio.

Supponiamo di aver definito un tipo PDU nel modo seguente:

```
PDU ::= SET {  alpha     INTEGER,
               beta      IA5String OPTIONAL,
               gamma     SEQUENCE OF Parameter
               delta     BOOLEAN
          }
```

È una strana PDU in cui l'ordine dei campi non è significativo! Una PDU di test può essere definita come

```
TestPDU ::= Pdu ( WITH COMPONENTS
                          { delta (FALSE), alpha (MIN .. <0) } )
```

è una particolare PDU in cui il campo `delta` vale `FALSE` e il campo `alfa` è compreso fra `MIN` e `0` escluso. Vedremo le definizioni di PDU di SNMP scritte in ASN.1.

B.3 Definizioni di Macro

ASN.1 offre una notazione assai potente per definire macro. La sintassi stessa dell'ASN.1 è modificabile (localmente) mediante la definizione di macro. Vedremo, in modo informale, una macro che è stata definita usando ASN.1 e viene usata in SNMP: vedremo solo l'applicazione della macro e non la sua definizione.

Conclusioni sull'ASN.1

Come "promesso" ASN.1 permette di definire strutture dati, e quindi Protocol Data Unit, senza fare riferimento a

- Architettura hardware usata dalle implementazioni
- Sistema operativo
- Linguaggio di programmazione
- Scelte implementative da lasciare all'implementatore

Ma poi i valori che appartengono ad una *sintassi astratta*, definita usando l'ASN.1, devono essere trasferiti usando una codifica (*sintassi di trasferimento*) che deve essere molto precisa e che non lasci spazio alla inventiva degli implementatori. È molto comodo che la sintassi di trasferimento sia automaticamente determinata da *regole di codifica basate sulla struttura del linguaggio ASN.1*, che prescrivano come convertire in sintassi di trasferimento i valori dei vari tipi di dato definiti usando l'ASN.1: In questo modo la sintassi di trasferimento è certamente corretta e non "costa" nulla una volta definite le regole di codifica.

Quali regole? Le Basic Encoding Rules (BER)

B.4 Basic Encoding Rules

Le BER sono una specifica di codifica sviluppata e standardizzata da CCITT (X.209) e ISO (ISO 8825) che descrive un metodo per codificare (serializzare) i valori di un qualunque tipo di dato definito usando ASN.1. È quindi un "frullino" potentissimo e comodissimo: nello stesso momento in cui definisco una PDU in termini di ASN.1 definisco anche il modo con cui i valori di questo tipo devono/possono essere affidati al data transfer component perché li consegni alla applicazione paritaria.

B.4.1 Struttura della codifica

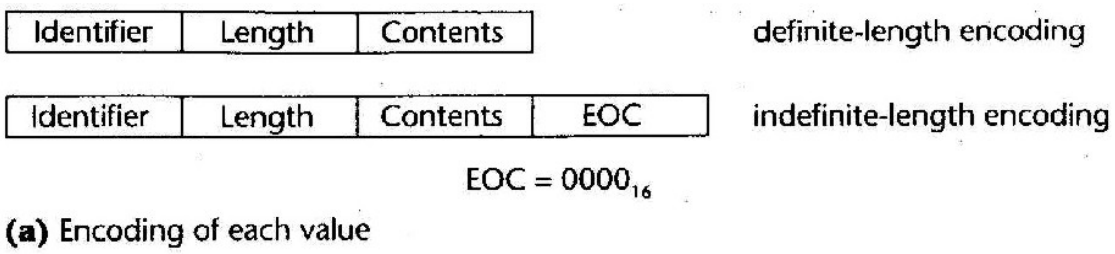
Le BER codificano un valore come una sequenza di ottetti che adotta una struttura chiamata

TLV = Tag-Lunghezza-Valore

(Stallings usa il termine Identifier al posto di Tag.) Quindi una codifica è sempre costituita da "triple" che eventualmente sono ricorsive nel senso che il Valore può essere a sua volta una TLV, se

- il Tag non è di tipi semplici ma di tipi strutturati, oppure
- se ho messo un Tag davanti ad un valore perché sia distinguibile da altri

Ci sono due modi di codificare la Lunghezza. Ma SNMP adotta uno solo dei due, e cioè quello detto a *lunghezza definita*. Ogni tripletta ha la struttura che segue, con due varianti a seconda se la lunghezza è codificata in forma definita o indefinita.



(a) Encoding of each value

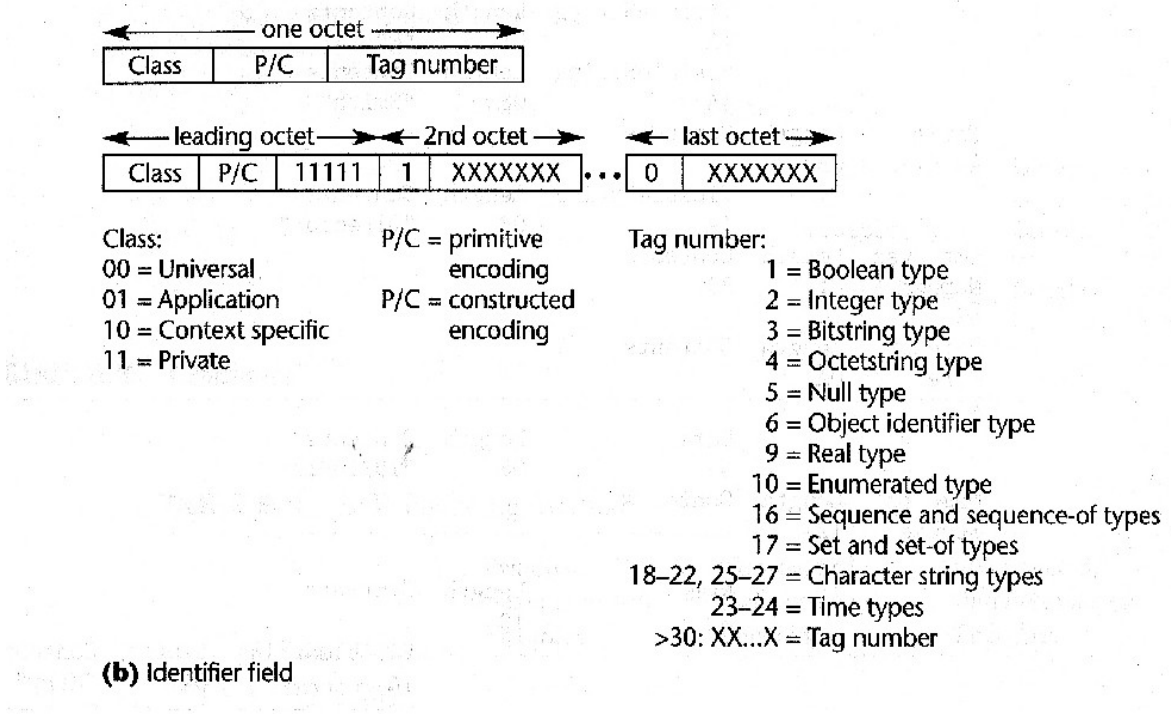
La "marca" di fine (EOC, costituita da due byte) non deve essere presente nel contenuto del valore:

questo sarebbe impossibile da garantire in generale! Quindi la lunghezza indefinita si può usare solo quando il valore è a sua volta costituito da una serie di TLV: in tal caso si può assicurare che la EOC non coincida con alcun Identifier.

Vediamo come sono codificati i TAG dell'ASN.1 ./.

Codifica del tipo/Tag

Il primo campo della tripla viene chiamato da Stallings Identifier perché serve per codificare il Tag dell'ASN.1, ma non coincide con il Tag dell'ASN.1: ha maggiore informazione. Ha una struttura complessa descritta dalla prossima figura.



(b) Identifier field

I primi due bit (da sinistra) codificano la classe del tag. Il terzo bit dice se la codifica (attenzione bene: la *codifica* non la *natura*!) del valore è *Primitiva* o *Costruita*. Perché ci può essere differenza fra natura del valore e codifica? Sono due ragione per la differenza.

Prima ragione: Lo stesso tipo di dati può essere (dover essere) codificato in modi diversi per cui non si capirebbe se c'è da aspettarsi una codifica primitiva o costruita. La codifica primitiva DEVE essere usata per i tipi semplici (come ad esempio `INTEGER`). La codifica costruita DEVE essere usata per i costruttori `SEQUENCE`, `SEQUENCE OF`, `SET` e `SET OF`.

Per i tipi stringa (`BIT STRING` e `OCTET STRING`) si può usare la codifica costruita oppure la codifica primitiva: chi riceve una codifica deve essere informato della scelta effettuata dal mittente, e questa informazione è contenuta nel terzo bit `P/C` dell'Identifier.

Seconda ragione: Quando un tipo è tagged (context-specific o application-wide) e la sintassi astratta prescrive che il tag sia IMPLICIT (quindi il tag originale non viene trasmesso) come si farebbe a sapere se il valore è costituito da una serie di TLV? Il bit `P/C` contiene questa informazione che sarebbe desumibile dall'Identifier del tag omesso.

Il Tag Number

Per i valori piccoli si vuole usare pochi bit e per quelli grandi se ne useranno di più. Quindi abbiamo due forme di rappresentazione.

1. Se il valore di tag è minore o uguale a 30 si mette nei cinque bit che rimangono nel primo

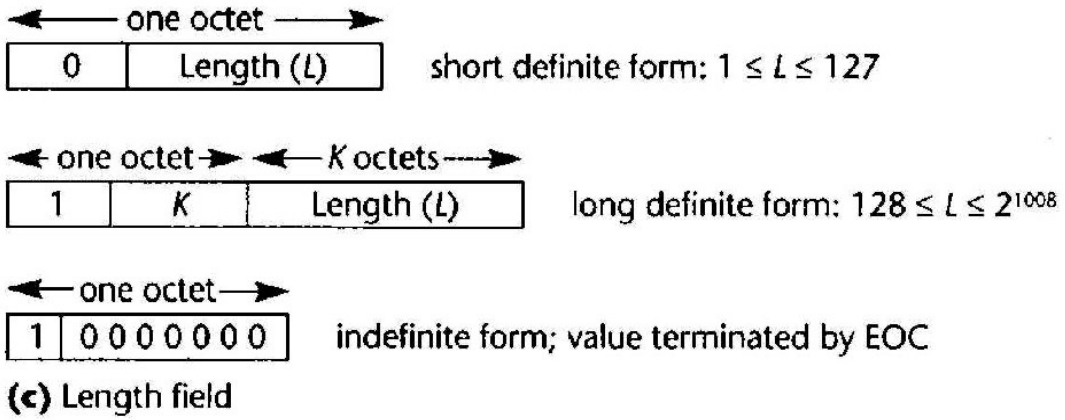
byte. dopo `Class` e `P/C`.

2. Se è maggiore di 30, i cinque bit che rimangono si mettono tutti a 1, e si usano gli ottetti che seguono QUANTO BASTA. Come si fa a sapere se servono più ottetti? Il primo bit di un ottetto che è seguito da un altro ottetto è messo a 1, mentre se l'ottetto è l'ultimo (o l'unico) viene messo a 0.

Quindi fino a valori maggiori di 30 ma minori o uguali a 127 si usa solo il solo secondo byte, altrimenti più byte, quanto è necessario. Valori piccoli di tag sono scambiati in modo efficiente: valori superiori a 30 "costano molto". Si noti che non esiste un limite superiore al valore del tag codificabile

La lunghezza

La lunghezza del valore può essere grande o piccola, e anche in questo caso si adotta una codifica a lunghezza variabile per poter trasmettere lunghezze piccole, che si pensa siano le più frequenti, con pochi bit. La codifica è descritta nella prossima figura.



Quando si usa la codifica indefinita della lunghezza si mette un "segno" che informa che occorre trovare l'EOC che determina la fine del valore. Si può usare solo se l'identificatore (Tag) indica che la codifica è costruita. Se lunghezza è codificata in forma definita ed è minore o uguale a 127, allora deve essere espressa dal solo primo ottetto, il cui primo bit vale 0.

Altrimenti il primo bit del primo ottetto della lunghezza vale 1, i successivi 7 bit dicono di quanti ottetti è fatta la lunghezza, che quindi può occupare fino a 126 ottetti (127 è riservato per future estensioni!) e quindi può valere fino a 2^{1008}.

Codifica dei valori

I valori primitivi sono codificati con una regola che di nuovo cerca di usare meno byte possibili, per cui gli `INTEGER` piccoli usano un solo byte e così via.

Un tipo definito con una regola di subtyping viene codificato come è codificato il tipo padre: quindi un tipo definito come intero senza segno viene sempre comunque codificato con il segno!

Le stringhe possono essere codificate sia in forma primitiva che costruita. Nel caso di codifica primitiva sono costituite da una stringa di byte, la cui lunghezza è data dalla Length della tripla. Nel caso di codifica costruita, il valore è a sua volta costituito da TLV, che hanno come valore del Tag il tag della stringa: in sostanza una stringa appare come una stringa di stringhe. In SNMP non si può usare la codifica costruita per le stringhe.

I tipi costruiti si codificano in modo ricorsivo, con la forma di codifica costruita, ri-applicando ai contenuti la regola TLV. Quindi se dal tag so che il tipo è `SEQUENCE` (e notate che lo stesso valore di tag si usa per `SEQUENCE` e per `SEQUENCE OF`) allora il ricevitore sa che il valore è a sua volta

una sequenza di triple TLV. La lunghezza in byte di tutta la `SEQUENCE` è data dalla Length esterna, e di ciascuna tripla conosco il tipo e la lunghezza del valore perché è codificata nella TLV.

Se nella sintassi si è usato un tag context specific o application wide, preposto ad un tipo ASN.1, una tripla TLV avvolgerà la tripla del valore taggato. Per evitare questo doppio tagging si può usare nella specifica la parola chiave `IMPLICIT`, per eliminare il tag e lunghezza interni: si risparmiano byte! Per questo il bit della forma di codifica (primitiva o costruita) è indispensabile.

Ciascun tipo primitivo ha una sua regola di codifica del valore. Per fortuna in SNMP si usano solo pochi tipi primitivi.

B.4.2.1 Codifica dei valori `INTEGER`

Per prima cosa, gli interi (che come tali hanno un segno!!!) sono codificati in complemento a 2. (andare indietro a ri-studiare il complemento a due perché non può essere ignorato all'esame di GSR). Su un numero finito di bit, i numeri positivi hanno il primo bit (quello più pesante) a Zero, mentre quelli negativi hanno il primo bit a Uno. Quanti bit hanno a Zero (o Uno) dipende dalla dimensione del numero e dal numero di bit che uso per la codifica. Nelle macchine si usa una codifica dei numeri a lunghezza fissa, in genere uguale alla lunghezza della parola della macchina. Ma nel caso delle BER le codifiche avranno lunghezza diversa in modo da risparmiare il numero di byte da trasmettere.

Se il numero è piccolo e/o uso tanti bit per la codifica ci potranno essere tanti bit a Zero (o Uno) per indicare il segno. Se il numero ha un valore assoluto grande serviranno necessariamente un numero grandi di bit per rappresentare il valore assoluto, ma anche qualche bit per rappresentare il suo segno. Usare più di un byte di Zeri o di Uni "iniziali" è inutile e dannoso se si vuole ottenere codifiche compatte, cioè con il numero minimo di byte.

Le BER codificano un intero in complemento a due, e *con il numero minimo di byte necessari*. Si vuole che i bit per la codifica siano il minimo possibile, ma ci deve essere almeno un bit per indicare il segno. Poiché si vuole anche che la codifica sia allineata al byte non possono scegliere di usare un solo bit in più: ci sono situazioni in cui l'aumento di 1 del valore richiede un byte in più per la codifica.

Gli interi senza segno o non negativi sono un sub-type degli interi, e si codificano nello stesso modo; vengono quindi codificati come numeri positivi. Quindi se un numero senza segno necessiterebbe su un host di solo un byte, ad esempio il numero il cui valore assoluto è 255, occorre trasmettere due byte, anche se il byte più “alto” è tutto a Zero, "solo" per segnare che è positivo. Anche il numero –255 ha bisogno di 2 byte, e il byte più pesante è fatto di soli Uno.

- La rappresentazione di un numero nella macchina e nelle BER è diversa:
- La macchina può non usare il complemento a due
- La macchina rappresenta i numeri su parole di lunghezza fissa

Codifica dei valori BOOLEAN

Il valore `FALSE` viene codificato con un byte fatto tutto di Zero; il valore `TRUE` con un qualunque byte diverso dal `FALSE`. In ambedue casi la lunghezza del valore è 1.

Codifica del valore `NULL`

E' un tipo che ha un unico valore: `null`. Dato che il Tag (o la sintassi astratta nel caso di tagging implicito) già dice che questo valore appartiene al tipo `NULL`, non è necessario trasmettere alcuna informazione aggiuntiva. Il valore ha quindi lunghezza 0, e il tutto occupa solo due ottetti: 0500_{16}. 05_{16} è il tag e 00_{16} è la lunghezza.

B.4.2.2. Codifica dei valori `OCTET STRING`

In SNMP la codifica deve esser sempre primitiva, cioè le stringhe non possono essere

rappresentate da concatenazione di sotto-stringhe, come si potrebbe fare nelle BER generali. Il valore contiene esattamente gli ottetti della stringa.

Le stringhe di caratteri sono sub-type di `OCTET STRING` e quindi si codificano nello stesso modo, anche se la tabella dei tag `UNIVERSAL` assegna per queste stringhe numerosi valori di tag. La differenza fra stringhe di ottetti e stringhe di caratteri è che in una stringa di caratteri possono solo comparire valori di byte a cui corrispondono segni visibili o caratteri di controllo (quali "l'a capo"). La corrispondenza fra valore del byte e segno visibile (o comando di controllo) è diversa nelle diverse stringhe e si indica spesso con il nome character set o tabella dei caratteri.

B.4.2.3. Codifica dei valori `OBJECT IDENTIFIER`

Un OID è, concettualmente, una sequenza di interi senza segno; ma in ASN.1 è un tipo di dato primitivo. I primi due interi sono amministrati, cioè possono avere solo valori molto piccoli e ben definiti (0, 1, 2). Il secondo intero, anche esso amministrato, non può superare 39. Allora le BER combinano i primi due identificatori X e Y in un unico identificatore dato dalla formula:

$$Z=(X \times 40) + Y$$

Come mai funziona? Perché il primo intero può essere solo 0, 1 o 2, mentre il secondo intero non supera mai 39. Per decodificare si deve applicare il calcolo seguente:

```
Se 0 <= Z <= 39 allora X=0, Y=Z
Se 40 <= Z <= 79 allora X=1, Y=Z-40
Se 80 <= Z <= 119 allora X=2, Y=Z-80
Altrimenti ERRORE
```

In questo modo si compattano due interi in un unico intero, e si ottiene un risparmio in termini di byte trasmessi, perché gli interi della sequenza vengono trasmessi nel modo seguente.

Ogni intero (e i primi due sono combinati in uno solo) è rappresentato in base 2 da una successione di gruppi di 7 bit, usandone il numero minore possibile. Ogni gruppo di 7 bit è inserito nella parte bassa di un byte, e il primo bit del byte mi dice se segue un altro byte (il primobit vale 1) oppure se è l'ultimo (o l'unico) della successione di gruppi (il primo bit vale 0). La lunghezza totale della sequenza di interi è data dalla tripla TLV. Poiché i primi due interi combinati in uno producono al massimo il valore 119, compattandoli si risparmia un byte.

A questo punto, abbiamo visto i due potenti strumenti (progettati e definito da ISO e ITU) che la SMI di SNMP usa. Possiamo tornare a vedere come è definita la

Structure of Management Information (SMI)

di SNMP.