EDI-C - ett odramatiskt programspråk

Bakgrund

Ett XML-dokument kanske inleds:

```
<?xml version="1.0"?>
<Invoice>
  <Header>
    <InvoiceDate>2003-11-24</InvoiceDate>
```

Det enklaste och mest naturliga sättet att ange datum i elementet `<InvoiceDate>` borde väl vara:

```
Invoice.Header.InvoiceDate.pcdata = "2012-10-24";
```

EDI-C är kanske det enda programspråket i världen som adresserar data i ett XML-dokument på detta sätt. Komponenter i EDI-meddelanden (EDIFACT, X12, TRADACOM), JSON-meddelanden och strukturerade filer adresseras med samma metod. EDI-C är mycket mer än ett språk för att hantera XML, EDI, JSON och andra typer av transaktionsfiler. Det innehåller dessutom allt det som en utvecklare av en integrationslösning behöver: SQL, COM, mängder av praktiska standardfunktioner och mycket annat.

Varför heter språket EDI-C och vad anspelar boktaven C på? Intentionen har inte varit att uppfinna ett språk med en egen "hemmagjord" syntax utan i stället använda den kanske mest spridda och kända syntaxstilen – den som används i C, C++, Java och C#.

En grundläggande strävan i design-arbetet har varit: undvik att hitta på något "eget" om det redan finns konstruktioner och metoder som man kan "låna" från välkända språk. Vi har valt att ta lämpliga delar från C, C++, Java, C#, Visual Basic och till och med från Cobol och PL/1. Det "yviga" i dessa språk och framför allt det "petiga" i C och C++ har vi försökt att undvika. EDI-C är anpassat till de som kan C, C++, C#, Java osv. – men även för de som bara har kommit i kontakt med de programspråken som hastigast. Många integrationskonsulter har kanske inte tid eller lust att lära sig ännu ett nytt språk – de har troligtvis fullt upp med att tänka på andra saker.

För en ambitiös programspråkskonstruktör kanske detta verkar fegt. Vi har inte varit ute efter att få erkännade för uppfinnadet av nya syntax-konstruktioner utan endast att underlätta för den som programmerar i EDI-C. Vår belöning är den aha-upplevelse som många programmerare upplever när de kommer i kontakt med EDI-C för första gången.

Någon har sagt att det fiffiga med EDI-C är att det inte är så fiffigt.

Ett interpreterande språk

Programkoden kompileras och en optimerad ”exekveringsbar” fil skapas. Denna fil kan sedan läsas av interpretatorn (virtual machine) som exekverar programmet. Metoden är ungefär densamma som används för Java.

Språkets grundläggande syntax

Alla de vanliga satserna som finns i C, C++ och Java finns också i EDI-C
Tilldelning

```
a = b + c / d;
```

If-satsen

```
if (e == ”123”)
  f = g();
else
  ++h;
```

For-satsen

```
for (i = 1; i < 100; ++i)
  {
  if (j(i) > 100)
    continue;
  k();
  }
```

While-satsen

```
while (l < m)
  if (n(l + m) == 1)
    break;
```

Do-while-satsen

```
do
  o += abc ();
while (o < 100);
```

Switch-satsen är något utökad

```
switch p
  {
  case “ABC” :
    q = 2;
```

```
    break;
  // Notera uttrycket
  case (123 + i) :
    q = 3;
    break;
  // Notera funktionsanropet
  case (xyz ()) :
    q = 4;
    break;
  default :
    break;
  }
```

En enklare for-each sats är implementerad

```
integer iArray [4]= {3, 5, 8, 13};
foreach (integer iValue in iArray)
  {
  println (iValue);
  }
```

Goto-satsen finns ej – någon (E.W.Dijkstra) lär ju ha ansett att den är skadlig.

De grundläggande datatyperna är `integer` och `string`. Automatisk konvertering sker mellan dessa typer av variabler. Det är alltså fullt möjligt att skriva:

```
integer i = 4711;
string s = ”10000”;
s = s + i; // s innehåller nu ”14711”
i = s + 1; // i innehåller nu vardet 14712
```

Sträng-konkatenering görs med & operatorn.

```
string s = ”12345”;
s = s & ”ABC”; // s är nu ”12345ABC”
```

Funktioner kan deklareras i valfri ordning. Inga funktionsprototyper behövs. Nedan visas ett exempel på en funktion.

```
// Funktion som adderar till eller subtraherar
// från ett datum angivet som år och dagnummer
string AddDate (string sDate, integer iDays)
  {
  // Konvertera datum till formatet: YYYYMMDD
  sDate = dateformat (sDate, ”YYYYMMDD”, ”YYDDD”);

  // Addera/subtrahera ett antal dagar
  sDate = addate (sDate, iDays);

  // Returnera resultatet på formatet: YYDDD
  return dateformat (sDate, “YYDDD”, “YYYYMMDD”);
  }
```

Komplexa datatyper i EDI-C – en bakgrund

En av de grundläggande behoven vid integration av system är att konvertera data från ett format till ett annat. Ett system kan lämna ifrån sig transaktioner med ett specifikt format. Dessa transaktioner kan inte direkt importeras in i ett annat system – det mottagande systemet kräver troligtvis sitt ”eget” format. Det behövs alltså en mekanism som omformar eller med ett annat ord, konverterar transaktionen till ett lämpligt importformat. Detta har sedan länge varit känt i EDI-sammanhang. I detta fall brukar konverteringen ske i två steg:

”Mitt system”	Konvertera ”mitt” systems transaktioner till ett överenskommet EDI-format (ex. EDIFACT) och skicka därefter EDI-filen till mottagande system.
”Mottagarens system”	Konvertera det mottagna EDI-meddelandet till ett format som kan hanteras av det mottagande systemet.

Program som omvandlar data till och från EDI-format brukar kallas syntax-konverterare eller ”EDI-translators”. Dessa program är oftast parameterstyrda och de innehåller vanligtvis även ett enklare programmeringsspråk. Med hjälp av programkod och speciella funktioner kan man till exempel konvertera ett inkommande datum:

```
2003-11-24
```

till ett datumformat:

```
20031124
```

som passar ”mitt” affärsystem.

I en syntaxkonverterare beskrivs oftast transaktionsfilernas format grafiskt. Informationen lagras kanske i en SQL-databas eller i ett XML-schema. Datastrukturerna och deras komponenter är sedan åtkomliga genom diverse funktionsanrop i programmeringsspråket. Vi har tidigare utvecklat program som arbetar enligt ovanstående modell.

Det finns dock en baksida med ett sådant tillvägagångssätt. Programspråket och filstrukturerna pratar inte riktigt samma språk. Strukturerna och programkoden finns i två olika miljöer och utvecklaren är tvungen att koppla ihop dem via funktionsanrop.

Vad är då det speciella med EDI-C ?

Filstrukturerna och dess komponenter är integrerade delar i programspråket – de är egna DATATYPER i språket.

Vad innebär detta ?

Varje komponent i en filstruktur är direkt adresserbar. Nedan visas återigen det inledande exemplet:

```
Invoice.Header.InvoiceDate.pcdata = ”2003-11-24”;
```

Eller om elementnamnen inte är valida enligt C-syntaxen:

```
Invoice.”Inv-Header”.”Inv.Date”.pcdata = ”2003-11-24”;
```

Ovanstående notation medför att namn-kontroller kan göras redan vid kompilering. Detta medför i sin tur både större säkerhet och snabbhet vid exekvering.

I EDIFACT-sammanhang kan data adresseras:

```
if (Ic.INVOIC.DTM[i].C507.2005 == “137”)
  invoiceDate = Ic.INVOIC.DTM[i].C507.2380;
```

Notera att inga funktionsanrop behöver göras för att ”komma åt” dataelementen.

Datatyper för XML

Datatyperna för XML är `xmldoc`, `xmlelem` och `xmlpcdata`. De två datatyperna `xmldoc` och `xmlelem` kan även innehålla definitioner för attribut.

```
xmlpcdata pcdata;

xmlelem Orderhuvud
  attribute (name=”Koepare”)
  attribute (name=”Saeljare”)
  {
  pcdata;
  };

xmlelem Artikel
  attribute (name=”Artikelnummer”)
  {
  pcdata;
  };

xmldoc Order
  {
  Orderhuvud;
  Artikel [1000];
```

```
};
```

XML-grupper (`choice, all, group, sequence`) deklareras som element med speciella attribut. De fördefinierade attributen `version, space, lang, encoding` och `standalone` är adresserbara. Namespace hanteras. Utvecklaren kan komma åt alla element och attribut i ett komplett XML-dokument.

```
Order.Orderhuvud.pcdata = ”123456”;
Order.Orderhuvud.#Koepare = ”ABC AB”;

// Så länge det finns artikeldata
for (i = 1; xmldata (Order.Artikel[i]); ++i)
  println (Order.Artikel[i].pcdata);
```

Adressering av komponenter i ett XML-träd kan göras genom att använda XML-pekare. Ovanstående programsatser kan då skrivas:

```
// Så länge det finns artikeldata
for (xmlptr pArtikel = Order.Artikel[1]; xmldata (pArtikel); ++pArtikel)
  println (pArtikel.pcdata);
```

Inläsning och validering av ett komplett XML-dokument sker med ett anrop till en standardfunktion.

```
xmlread (Order, ”testfil.xml”);
```

Eftersom XML-strukturen (XML-dokumentets schema) beskrivs i källkoden och kompileras medför detta att läsning och validering blir synnerligen effektiv och snabb. Denna teknik gör det möjligt att tolka ett XML-dokument med en hastighet av 10 Mb per sekund.

Skrivning av ett XML-dokument görs med hjälp av en annan standardfunktion.

```
xmlwrite (Order, ”testfil2.xml”);
```

Datatyper för JSON

Ett JSON dokument beskrivs genom att använda XML variabler på ett speciellt sätt – mer om detta i dokumentation för JSON.

Datatyper för EDIFACT

Datatyper för EDIFACT är `element, composite, segment, segmentgrp, message` och `interchange`. De deklareras som:

```
element 0051 a..2;
```

```
element 0052 an..3;
.
.

composite S009
  {
  0065;
  0052;
  .
  .
  };

segment UNH
  {
  0062;
  S009;
  .
  .
  };

message ORDERS
  {
  UNH;
  BGM;
  DTM[35];
  LIN [10000]
    {
    QTY;
    .
    .
    };
  };

interchange icORDERS
  {
  UNB;
  ORDERS [999];
  UNZ;
  };
```

Alla komponenter kan adresseras direkt utan att behöva läsa delar av EDIFACT-filen.

```
icORDERS.ORDERS.UNH.S009.0065 = ”ORDERS”;
```

Adressering av komponenter i ett EDIFACT-träd kan göras genom att använda EDIFACT-pekare. Se exempel nedan:

```
// Så länge det finns meddelanden
for (ediptr pMsg = icOREDRS.ORDERS; edidata (pMsg); ++pMsg)
  // Gör något för varje meddelande
```

Inläsning och validering av en EDIFACT-fil sker med ett anrop till en standardfunktion.

```
ediread (icOrders, ”testfil.edi”);
```

och skrivning görs med en annan standardfunktion.

```
ediwrite (icORDERS, ”testfil2.edi”);
```

Datatyper för flatfiler

Med flatfiler menar vi strukturerade ASCII-filer. Till denna kategori hör även filer enligt den Amerikanska EDI-standarden X12. Datatyperna för flatfiler är `flatfile`, `flatrecgrp` och `flatrec`.

```
flatrec Orderhuvud recid (”001”)
  {
  Koepare an..35;
  Ordernummer n..10;
  };
flatrec Artikel recid (”002”)
  {
  Artikelnummer n..15;
  Benaemning an..50;
  };

flatfile Order variable style (Undefined) recordsep (”\r\n”)
  {
  Orderhuvud;
  Artikel [10000];
  };
```

Notera attributet `style` i `flatfile` deklarationen. Detta attribut anger hur en flatfil generellt kan vara uppbyggd och hur filen skall läsas och skrivas. Ett exempel är `SAP_IDOC`. För närvarande finns ett tiotal varianter definierade. Nya typer kan lätt läggas till.

Utvecklaren kan komma åt alla fält i alla poster i en komplett flatfil.

```
Order.Orderhuvud.Ordernummer = 123456;
Order.Orderhuvud.Koepare = ”ABC AB”;

for (i = 1; flatdata (Order.Artikel[i]); ++i)
  println (Order.Artikel[i].Artikelnummer);
```

Adressering av komponenter i ett faltfils-träd kan göras genom att använda flatfils-pekare. Ovanstående programsatser kan då skrivas:

```
// Så länge det finns artikeldata
for (flatptr p = Order.Artikel[1]; flatdata (p); ++p)
  println (p.Artikelnummer);
```

Inläsning och validering av en flatfil sker med ett anrop till en standardfunktion.

```
flatread (Order, ”testfil.inh”);
```

och skrivning görs med en annan standardfunktion.

```
flatwrite (Order, ”testfil2.inh”);
```

Full SQL ingår

Programspråket hanterar standard ”embedded” SQL. Interpretatorn ”mappar” SQL-satserna mot lämpliga ODBC-funktioner, vilket betyder att alla databaser som har en ODBC-driver kan användas.

```
EXEC SQL SELECT orderno, seller, buyer
  INTO :icORDERS.ORDERS.BGM.1004, :icORDERS.UNB.S002.007,
    :icORDERS.UNB.S002.004
  FROM orderhead
  WHERE orderno = :MittOrdernummer;
if (SQLSTATE != ”00000”)
  // Icke funnen
```

Alla SQL kommandon kan användas men även anrop till “stored procedures” kan göras.

```
EXEC SQL CALL StoredProcAbc (:Arg1, :Arg2);
while (SQLSTATE == “00000”)
  EXEC SQL GET StoredProcAbc INTO :RecSetCol1, :RecSetCol2, :RecSetCol3;
```

COM hantering

Microsoft’s COM-teknik är implementerad. Hanteringen påminner mycket om den som används i Visual Basic. Lämpliga typkonverteringar sker automatiskt. Argument av typen ”User defined” tillåts.

Ett objekt deklareras

```
object myObj;
```

och initieras (instansieras)

```
myObj = createobject (”someCOMmodule.exe”;
```

därefter är objektets funktioner och variabler direkt åtkomliga.

```
myObj.myMethod (123, ”ABC”);
myVar1 = myObj.myProp1;
myObj.myProp2 = myVar2;
```

För s.k. ”get/put” variabler är det fullt möjligt att skriva:

```
++myObj.myProp3;
myObj.myProp4 &= ”ABC”;
```

Standardfunktioner

För närvarande finns det över 400 inbyggda standardfunktioner. Det tillkommer kontinueligt nya, främst efter önskemål från utvecklare/användare.

Gemensamt med alla standardfunktioner är att vi har försökt göra dem så ”high-level” som möjligt. I exemplet nedan översätts alla små bokstäver i en ASCII-fil till stora bokstäver och därefter konverteras alla tecken till EBCDIC.

```
string s = readfile (”t.txt”)
s = toupper (s);
s = convertmem (s, 1, 2)
writefile (s, “t.txt”);
```

Ovanstående programsatser kan skrivas på en rad om man så önskar:

```
writefile (convertmem (toupper (readfile (”t.txt”)), 1, 2), “t.txt”);
```

För de komplexa datatyperna (XML, EDIFACT, flatfiler) finns många kraftfulla funktioner.

Det finns en mängd strängfunktioner för alla upptäkliga behov (`substr, trim, replace, search, regexp, translate` osv).

Decimalaritmetik och numerisk editering görs med ett antal funktioner. Tal med up till 25 heltalssiffror och 10 decimaler kan hanteras.

ASCII-filer kan läsas och skrivas (`open, read, readfile, readnl, write, writefile, writeln, close, tell, seek, unread osv.`).

Ett antal ”file and directory” funktioner finns `(chdir, direxist, dirname, fileappend, filecopy, fíndnext, findnextdir` osv).

Andra program (.EXE, .DLL) kan anropas både synkront och asynkront. Andra EDI-C program kan anropas – till och med enskilda funktioner i dessa.

Datum och tid kan manipuleras på ett flertal sätt.

Funktioner finns för loggning, trace, zip osv.

Det senaste tillskottet är ett antal funktioner för tolkning av ”fri text”. Dessa s.k. ”token”-funktioner gör det enkelt att tolka till och med programkod.

Implementation

Detta avsnitt beskriver kortfattat hur implementationen av EDI-C har gjorts. Först något om kompilatorn och därefter lite information om interpretatorn.

Språket innehåller SQL-satser med helt andra operatorer än de i C-syntaxen. Bland annat på grund av detta, använder kompilatorn ”Recursive descent” teknik för att analysera programkoden. Denna metod ger också större frihet vid semantisk analys. Felhanteringen kan även göras bättre med fylligare felmeddelanden.

Kompilatorn utgörs av en C++ klass med ett antal ”hjälp”-klasser såsom lexikalisk analys, felhantering, symbol-tabell, ”statement tree manager” samt några till. Klasser för de komplexa datatyperna XML, EDIFACT och flatfiler finns också. En speciell klass hanterar COM-objekt och dess variabler. Eftersom kompilatorn är en egen klass har det varit enkelt att bygga in den i utvecklingsverktyget Inobiz DS samt att kapsla in den i en COM-komponent.

Resultatet av en kompilering är en binärfil innehållande bl.a. symboltabellen och ett komprimerat statement-träd. Statement-trädet är optimerat för snabb exekvering i interpretatorn. Resultatfilen innehåller även prototyp-träd för alla komplexa datatyper som har använts.

Interpretatorn är implementerad som en egen C++ klass med ett antal ”hjälp”-klasser. Liksom för kompilatorn har det varit enkelt att bygga in interpretatorn i Inobiz DS men även i Inobiz IS och Inobiz RT. Interpretatorn finns även som en .NET komponent och används i denna form, som en OEM-produkt, av ett antal ”konkurrerande” integrations- och EDI-system.

Interpretatorn är optimerad för snabb exekvering. Direkt adressering av variabler och funktioner görs. Variabler hanteras ungefär som VARIANT metoden i COM-sammanhang.

Funktionerna för läsning och validering av XML-dokument, EDI-filer och flatfiler är implementerade enligt den s.k. DOM-metoden. Det finns dock varianter av dessa läsfunktioner som är implementerade med en hybrid av DOM och SAX metoderna. Detta medför att exempelvis mycket stora XML-filer (upp till 10GB) kan hanteras.

Både kompilatorn och interpretatorn är effektivt implementerade i ”låg nivå” C++ vilket har medfört kompakthet och snabbhet. Storleken på kompilatorn är ca: 300 KB och interpretatorn är på ca: 1MB. Notera att i interpretatorn ingår förutom alla standardfunktioner även alla ”parsers” (XML, EDIFACT, X12, JSON, TRADACOM, CSV, EXCEL, Flatfil) och all SQL-hantering.