

iPEX

Version 2.0



1 iPEX2.0 を利用した XML 文書処理	1
1.1 XML パーサ.....	1
1.2 DOM.....	1
1.3 SAX.....	1
2 DOM を用いた XML 文書処理	2
2.1 DOM パーサの特徴.....	2
2.2 DOM ツリーの構成.....	2
2.3 iPEX を用いた XML 文書のパース.....	6
2.4 DOM から XML 文書を出力する.....	12
2.5 DOM から特定のデータを抽出する.....	15
2.6 DOM オブジェクトを構築する.....	20
3 SAX パーサによる XML 文書処理	25
3.1 SAX パーサ.....	25
3.2 SAX パーサの特徴.....	26
3.3 SAX パーサの構成.....	26
3.3.1 ContentHandler.....	27
3.3.2 ErrorHandler.....	27
3.3.3 DTDHandler.....	27
3.3.4 EntityResolver.....	27
3.4 SAX パーサによる XML 文書のパース.....	28
4 XSLT を利用した XML 文書変換	33
4.1 XSLT プロセッサ.....	33
4.2 XML 文書とスタイルシート.....	34
4.3 XSLT プロセッサによる文書変換.....	37
4.4 xsl:param を使ってデータを抽出する.....	42

1 iPEX2.0 を利用した XML 文書処理

本書では、XML 文書をアプリケーションで処理する際に使用する標準インタフェースの利用方法を、iPEX2.0 のプログラミング例を用いて紹介します。

iPEX は、インフォテリア株式会社が提供する XML プロセッサです。iPEX は、XML パーサ機能のほかに、XSLT プロセッサの機能も内包しています。XSLT は、XML 文書に表示スタイルを付加する際の変換規則を記述するための標準仕様です。

本章では、XML パーサの標準 API である DOM と SAX の概要を説明します。次章以降では、DOM、SAX、XSLT のそれぞれについて、iPEX を利用したアプリケーションの例として、XML 文書の解析、XML 文書からのデータ抽出、XML 文書の整形などのプログラムを作成します。

1.1 XML パーサ

XML(eXtensible Markup Language)文書をプログラムから利用する際には、一般に、テキスト形式で記述された XML 文書を読み込んで、XML のタグで指定された文書要素や属性を解析する必要があります。こうした XML 文書の解析処理を「パース」といいます。XML 文書のパースを目的とした XML プロセッサを「XML パーサ」といいます。

XML パーサは、アプリケーションプログラムの中で XML 文書を入出力したり XML 文書の要素へアクセスしたりする際のもろもろの手続きをカプセル化するソフトウェアモジュールです。XML パーサを通して XML 文書进行操作するための API(Application Programming Interface)には、DOM と SAX という 2 種類の標準インタフェースがあります。iPEX は、これら 2 種類の標準インタフェースに対応しています。

1.2 DOM

DOM(Document Object Mode)は、階層構造を持つ XML 文書をメモリ上に展開した文書オブジェクトへアクセスする際のインタフェースを規定したものです。DOM インタフェースは、W3C で策定している API であり、XML 文書を内部的に表現する論理的な木構造のオブジェクトモデルを定義しています。この木構造のオブジェクトモデルを「DOM ツリー」といいます。

DOM Level 1 仕様では、文書構造および内容を表現および操作するために必要とされるメソッドに関する仕様を規定しており、XML 文書を内部表現した各文書要素へアクセスする際のインタフェースが定義されています。DOM の仕様では、アプリケーションをコーディングする際の言語マッピングとして OMG CORBA 2.2 IDL、Java、ECMAScript によるインタフェースが定義されています。

DOM を利用することにより、XML 文書へアクセスするプログラムを開発する際に、標準インタフェースに基づいたプログラムを作成できます。iPEX は、W3C の勧告となっている DOM の Level 1 と Level 2 に対応しています。iPEX における DOM インタフェースの詳細については、マニュアルを参照してください。

1.3 SAX

SAX(Simple API for XML)は、XML 文書をパースするプログラムを作成する際のインタフェースを規定したものです。SAX インタフェースは、David Megginson が中心となって策定している API であり、XML 文書の各要素(タグ、属性、テキストなど)を解析するためのメソッドとのインタフェースを規定しています。SAX は、業界のデファクトスタンダード(事実上の標準)となっています。

SAX インタフェースに基づいて XML 文書の各要素を処理するためのメソッドを定義したプログラムを「ハンドラ」といいます。SAX を利用することにより、アプリケーション

固有に開発したハンドラを SAX パーサに登録することにより、独自の文書処理プログラムを開発できます。iPEX は、SAX Version 2 の仕様に対応しています。iPEX における SAX インタフェースの詳細については、マニュアルを参照してください。

一般に、XML 構造の複雑な操作を必要としないアプリケーションには SAX が、複雑な操作を必要とする場合は DOM が適しています。

2 DOM を用いた XML 文書処理

本章では、iPEX の DOM API を利用してアプリケーションプログラムから XML 文書を処理する方法を説明します。最初に XML パーサの基本的な機能を説明し、iPEX の DOM を用いたデータ処理の利用例を解説しながら、XML 文書からの DOM の構築、DOM の各オブジェクトへのアクセス、DOM ツリーの生成と XML データ出力といったプログラムを作成します。

2.1 DOM パーサの特徴

DOM 用 XML パーサは、W3C で勧告となっている Document Object Model で規定されたインタフェースによる DOM ツリーへのアクセスを提供します。

DOM 用 XML パーサは、指定された XML 文書を順次入力しながら、DOM ツリーを階層的に構築します。構築された DOM のドキュメントオブジェクトは、パース完了後にアプリケーションから取得できます。アプリケーションでは、引き渡された DOM オブジェクトの各メソッドを呼び出すことによって XML 文書へアクセスできます(図 2.1)。

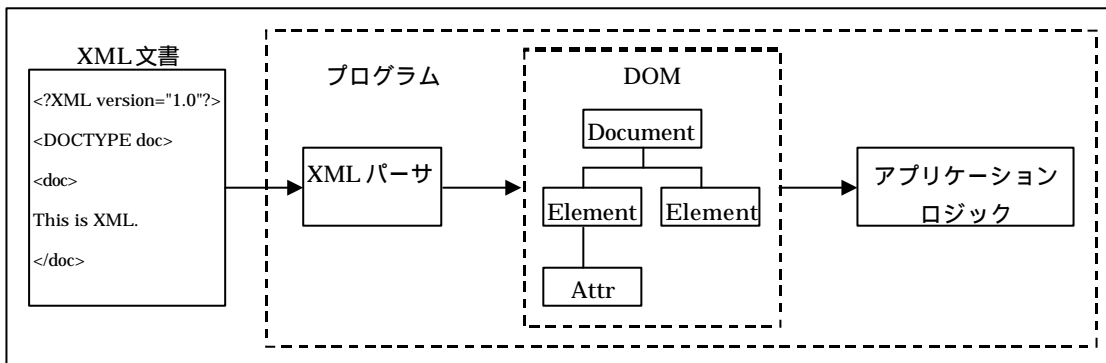


図 2.1 XML パーサと DOM

W3C による DOM の仕様では、XML 文書をパースして DOM を構築するための API を定義していないため、利用するパーサの実装に依存したクラスが提供するインタフェースを利用することになります。つまり、XML 文書から DOM を生成する部分は、パーサに依存したコードになります。

また、DOM 仕様は、XML 文書のパースだけでなく、DOM ツリーから XML 文書への出力も規定していないため、独自に出力プログラムを作成する必要があります。iPEX の DOM 用 XML パーサは、DOM の形式で格納されたドキュメントデータを特定の出力媒体へ整形して出力する機能を標準で提供しています。この機能を利用することにより、XML のタグ付け規則やエンコーディングに対応したプログラムを独自にコーディングすることなく、DOM 形式のドキュメントオブジェクトから簡単に XML 文書を出力できます。

2.2 DOM ツリーの構成

iPEX の DOM API を用いたプログラミングを解説する前に、DOM ツリーの各ドキュメ

ントオブジェクトについて概要を説明します(表 2.1)。

表 2.1 DOM インタフェース

インタフェース名	説明
Attr	Element の属性
Attribute	Element の属性(Java のみ)
CDATASection	CDATA セクション内の文字データ
CharacterData	PCDATA などの文字データ
Comment	コメントタグ内の文字データ
Document	XML または HTML 文書全体
DocumentFragment	文書の一部分を Document よりも簡便に扱うためのオブジェクト
DocumentType	DOCTYPE 宣言
DOMImplementation	特定の文書のインスタンスに依存しない操作を実行するためのメソッドを提供
Element	文書要素
Entity	文書内の実体
EntityReference	DOCTYPE 宣言内の実体参照
NamedNodeMap	Node のコレクションを名前で作るためのオブジェクト
Node	DOM 内の各ノードに対する基本データ型
NodeList	文書内の下位要素のリスト
Notation	DOCTYPE 宣言内の記法宣言
ProcessingInstruction	処理命令
Text	文書要素または属性の文字データ
DOMException	処理中に発生する例外

iPEX における DOM の各インタフェースは C++ のクラスとして定義されています。詳細については、マニュアルを参照してください。

DOM のドキュメントオブジェクトでは、XML 文書の構成要素ごとにオブジェクトが割り当てられます(図 2.2)。この文書構成要素を表わすオブジェクトを「ノード」といいます。ノードは、ノード自身を表わす Node というインタフェースが基底になりますが、さらに、文書の構成要素の種類によって異なるインタフェースのオブジェクトが割り当てられます。たとえば、文書要素を表わす Element というインタフェースは Element 固有の属性や操作を定義しているだけですが、DOM API を利用するプログラムからは Node インタフェースが持つ属性や操作を Element で定義されているのと同様に利用できます。

DOM で定義されたインタフェースの中には、Node インタフェースを基底インタフェースとしていないものがあります。DOMImplement インタフェースは、XML 文書の構成要素というよりは処理系の実装を隠蔽するための情報を提供するためのインタフェースです。また、NodeList は並び順に意味のあるノードを収容するためのコレクションを表現しています。あるノードに属する下位の文書要素や文字データを XML 文書内での出現順に管理するために使用され、Element や Text などのインスタンスを要素に持ちます。そして、NamedNodeMap は並び順に意味はないが名前をキーにして値を参照する必要があるノードを収容するためのコレクションです。たとえば、文書要素を表わすタグ内に記述された属性はタグ内での並び順に意味はないので、名前をキーにして Attribute のインスタンスへアクセスすることになります。

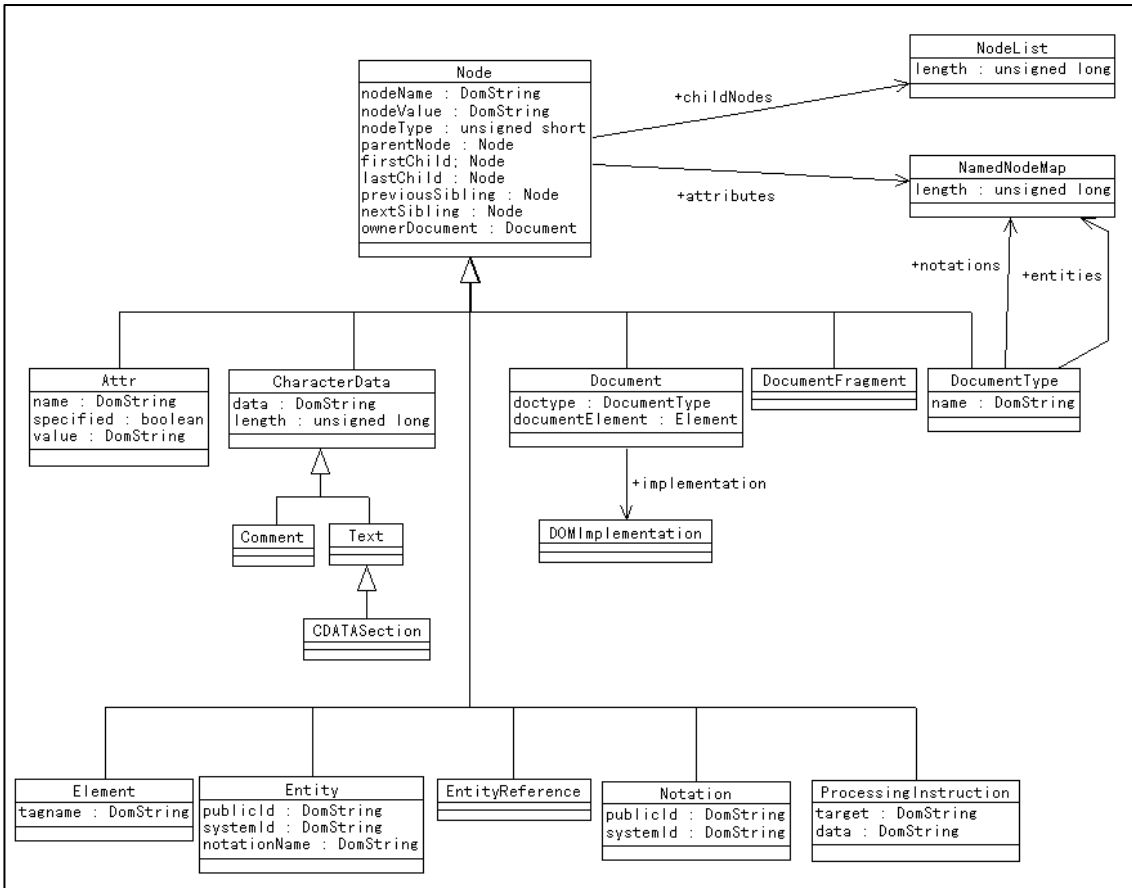


図 2.2 DOM のインタフェース構成

XML 文書全体を表現するノードは Document クラスのインスタンスであり、XML 文書の文書構造を反映した形で、各構成要素を表わすノードへリンクする構造となっています。Document のインスタンスを入り口としたノード間のリンク関係を図 2.3 に示します。

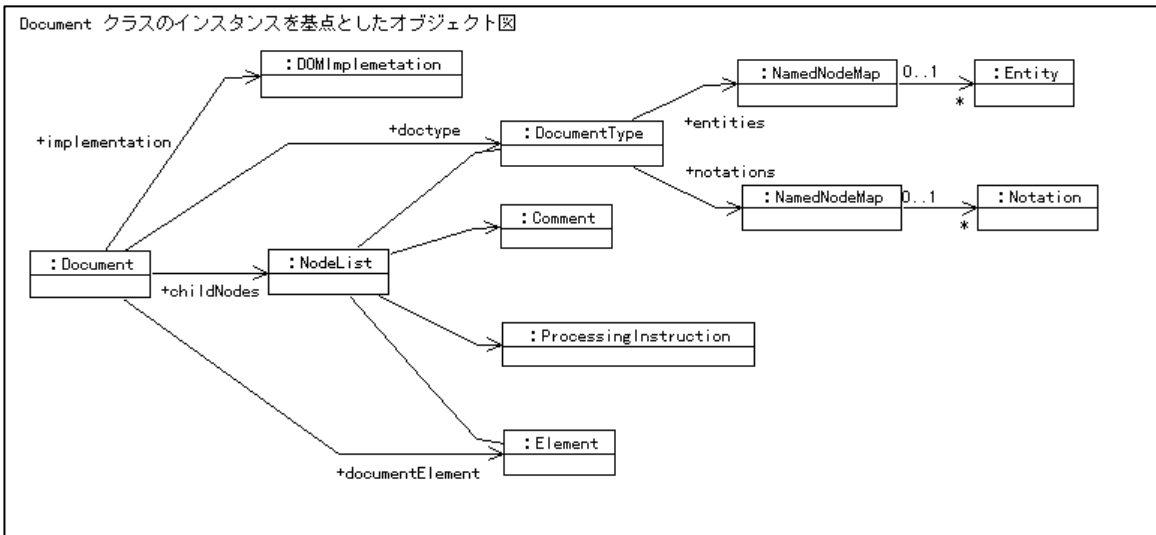


図 2.3 Document を入り口としたノード間の関連

Document インタフェースは、XML 文書全体を表わしています。Document には、DOCTYPE 宣言を表わす DocumentType、ルートの文書要素を表わす Element、処理命令を表わす ProcessInstruction、コメントを表わす Comment という 4 種類のノードが下位要

素として属しています。これらのノードは、Documentの子要素として NodeList というコレクションに收容されます。DocumentType と Element は、Document から直接アクセスできます。ルートレベルの処理命令やコメントへアクセスするためには、Document から documentElement へのリンクをたどるのではなく、childNodes というコレクションへアクセスする必要があります。

この図では、便宜的に、Document から文書要素の Element へのリンクを、IDL の定義にしたがって documentElement という属性として描いていますが、iPEX でのマッピングでは getDocuementElement() という関数に対応します。

ある文書要素を表わす Element のインスタンスを処理している際には、Element だけでなく基底インタフェースの Node で定義しているリンクをたどることによって関連する他の構成要素へアクセスできます(図 2.4)。

ある文書要素を処理しているときに XML 文書全体を表わす Document へ戻る場合には ownerDocument メソッドに対応する getOwnerDocument() 関数を呼び出します。すぐ上位の文書要素を取得する場合には parentNode メソッドに対応する getParentNode() 関数を呼び出します。

この図では、ある文書要素に属する文字データに対応するノードが描かれていませんが、getChildNodes() 関数で取得した NodeList には基底インタフェースの Node として收容されています。NodeList に收容されているノードの実際のインタフェースは、Element であることもあるし、Text や Comment であることもあります。Node にキャストされたインタフェースのタイプが分からなければ、ノード固有のメソッドを利用できません。ノードのタイプを取得するためには、iPEX では、Node のメソッドとして getNodeType() という関数を呼び出します。この関数を用いてノードのタイプを調べてから、タイプ別の処理を実行する必要があります。

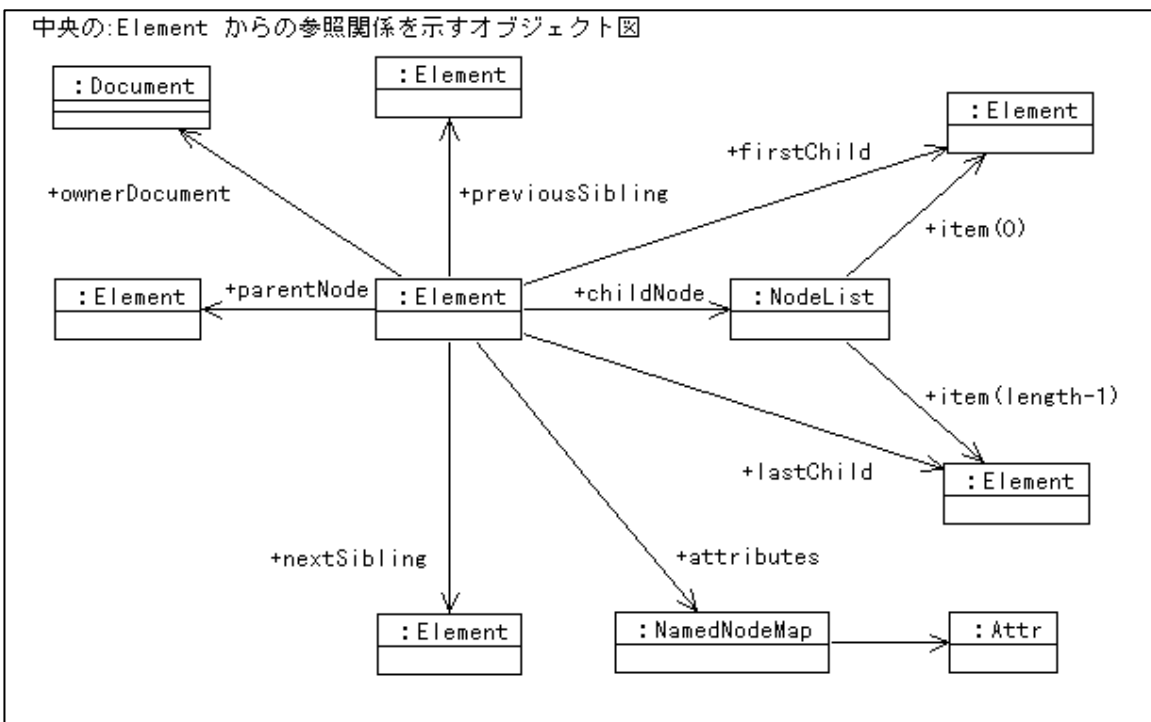


図 2.4 Element からのリンク関係

2.3 iPEX を用いた XML 文書のパース

XML 文書を実際に読み込んで DOM のドキュメントオブジェクトを生成して、iPEX の DOM を利用して DOM の構造を理解するための C++ アプリケーションを作成します。DOM オブジェクトを構築して、DOM のドキュメントオブジェクトを上位の文書要素から順にトランプース(走査)しながら各文書要素の内容を出力するプログラムを作成します。

(1) iPEX クラスのインクルード

iPEX で定義されているクラスを使用するためには、以下のようにヘッダファイル `ipexdom.h` をインクルードする必要があります。ここで定義されているクラスなどは、すべて iPEX ネームスペースの中で定義されているので、これらのクラスを使用するためには、iPEX:: を明示的に使うか、using 宣言をする必要があります。ここでは std ネームスペースと共に using 宣言を使うことにします。

```
#include <ipexdom.h>
using namespace iPEX;
using namespace std;
```

(2) メインプログラムの実装

これから `DOMEcho1.cpp` という名前のファイルでプログラムを作成します。プログラムで処理すべき XML 文書は、コマンドライン引数によって指定することになります。そのため、引数の数が 1 でない場合には、標準エラー出力へメッセージを出力してプログラムを終了させることにします。

```
...
int main(int argc, char* argv[])
{
    ...
    if (argc != 2) {
        cerr << "Usage: DOMEcho1 filename" << endl;
        return 1;
    }
    ...
}
```

(3) XML 文書のパース

XML パーサに XML 文書をパースさせるためには、まず iPEX で提供されている非検証 DOM 用 XML パーサのインスタンス(XMLReader)を生成します。

次に、コマンドラインの第一引数として指定された XML 文書のファイルパス名をにより、InputStreamByFile クラスを利用して入力ストリーム ifs を作成します。DOM 使用時には文字列は DOMString クラスを用いて表現されます。そのため、EUC コード文字列 ->DOMString、DOMString->EUC コード文字列変換のためのマクロ、WSTR と MBSTR を定義しておきます。また、EUC コードサポートを有効にするために、EUCJPConverterFactory のインスタンスを定義します。

```
...
#defineWSTR(x) iPEX::Character::toWString(x,L"euc-jp")
#defineMBSTR(x) iPEX::Character::toMString(x,L"euc-jp")
...
int main(int argc, char* argv[])
{
    EUCJPConverterFactory fEUCJP;
    ...
    DOMString pathname =WSTR(argv[1]);
    InputStreamByFile ifs(pathname.c_str());
```

```

if (!ifs) {
    cerr << "Cannot open file: " << MBSTR(pathname) << endl;
    return 1;
}
XMLReader reader(&ifs);
. . .
}

```

次に、IPEXDocument::createDocumentObject()により、Document ノード pDoc を作成し、XMLReader::read()により、pDoc を Document ノードとする DOM ツリーを構築します。

```

auto_ptr<Document> pDoc(IPEXDocument::createDocumentObject());
if (!reader.read(pDoc.get())) {
    cerr << "Cannot read xml: " << MBSTR(pathname) << endl;
    return 1;
}

```

(4) DOM のトラバース

パースした結果として構築された DOM オブジェクト、つまり Document のインスタンス内にどのように文書要素や属性が格納されているかを調べると同時、DOM オブジェクトを階層順に走査する例を示すために、print()関数を独自に用意して呼び出すことにします。

この print()関数は、DOM の各ノードに応じた処理を実行できるようにします。print()関数の中では、引数として引き渡された Node のタイプを調べて、タイプに応じた処理を記述できるようにします。ここでは、サンプルの XML 文書の解析に必要な Document、Element、Text について処理を分岐させています。

```

. . .
print(pDoc.get());
. . .
void print(Node* pNode)
{
    . . .
    if (pNode == 0) {
        return;
    }
    switch (pNode->getNodeTypeId()) {
    case Node::DOCUMENT_NODE:
        . . .
        break;
    case Node::ELEMENT_NODE:
        cout << "Element start: " << MBSTR(pNode->getNodeName()) << endl;
        . . .
        break;
    case Node::TEXT_NODE:
        . . .
        break;
    }
}

```

(5) Node 別の処理

Document のインスタンスが引き渡された場合、Document の開始と終了を示すメッセージを出力させます。開始と終了のメッセージを出力する間で、ルート of 文書要素を処理させるために、getDocumentElement()で取得したルート要素(最上位の文書要素)について、再帰的に print メソッドを呼び出します。

```
cout << "Document start:" << endl;
print((static_cast<Document*>(pNode))->getDocumentElement());
cout << "Document end:" << endl;
```

次は少々複雑になりますが、Element のインスタンスを処理するコードを作成します。Element についても、Element の処理全体の最初と最後に、それぞれ開始と終了を示すメッセージを出力させます。このとき、開始メッセージと終了メッセージには、getNodeName() で取得した文書要素名(タグ名)を付加することにします。

Element に対する処理では、getAttributes() 用いて属性の指定があるかをチェックします。属性がある場合には、NamedNodeMap に登録されている属性について、順にメッセージを出力します。DOM では属性の並び順を意識しないようになっているので、XML 文書に記述された順に出力されるとはかぎりません。属性のメッセージについては、属性名に続けて、属性値を二重引用符でくくって出力させます。

さらに、getChildNodes() で子要素の有無をチェックします。子要素がある場合には、NodeList に登録されている子要素をリストに登録されている順に、再帰的に print() を呼び出してメッセージを出力させます。print() が再帰的に呼び出されるので、子要素が Element であっても Text であっても順番にメッセージを出力できます。

```
cout << "Element start: " << MBSTR(pNode->getNodeName()) << endl;
attrs = pNode->getAttributes();
if (attrs != 0) {
    for (int i = 0; i < attrs->getLength(); i++) {
        Node* attr = attrs->item(i);
        cout << " " << MBSTR(attr->getNodeName())
            << "=\"" << MBSTR(attr->getNodeValue()) << "\"" << endl;
    }
}
children = pNode->getChildNodes();
if (children != 0) {
    for (int i = 0; i < children->getLength(); i++) {
        print(children->item(i));
    }
}
cout << "Element end: " << MBSTR(pNode->getNodeName()) << endl;
```

最後に、Text ノードの値を出力するコードを記述します。Text のインスタンスは、文字列のみを値とするので、その文字列を二重引用符でくくって出力します。この時、getNodeValue() で得られる文字列は DOMString 型なので、表示する時には予め定義した MBSTR マクロで EUC コードに変換することに注意してください。

```
cout << "Text: ¥" << MBSTR(pNode->getNodeValue()) << "\"" << endl;
```

(6) プログラムのコンパイル

ここまでで説明したコードの断片を一つのプログラムにまとめたものをリスト 2.1 に掲載します。

リスト 2.1 DOMEcho1.cpp

```
#include <iostream>
#include <memory>
using namespace std;

#include <ipexdom.h>
using namespace iPEX;
```

```

#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBString(x,L"euc-jp")

void print(Node* pNode);

int main(int argc, char* argv[])
{
    EUCJPConverterFactory fEUCJP;

    if (argc != 2) {
        cerr << "Usage: DOMEcho1 filename" << endl;
        return 1;
    }
    // 入力文書をオープン
    DOMString pathname = WSTR(argv[1]);
    InputSteamByFile ifs(pathname.c_str());
    if (!ifs) {
        cerr << "Cannot open file: " << MBSTR(pathname) << endl;
        return 1;
    }
    // 入力文書を読み込み
    XMLReader reader(&ifs);
    auto_ptr<Document> pDoc(IPEXDocument::createDocumentObject());
    if (!reader.read(pDoc.get())) {
        cerr << "Cannot read xml: " << MBSTR(pathname) << endl;
        return 1;
    }
    // 入力文書の内容を標準出力へ出力
    print(pDoc.get());
    return 0;
}

void print(Node* pNode)
{
    NodeList* children = 0;
    NamedNodeMap* attrs = 0;

    if (pNode == 0) {
        return;
    }
    switch (pNode->getNodeTypeInfo()) {

        // ドキュメントノードの出力
        case Node::DOCUMENT_NODE:
            cout << "Document start:" << endl;
            print((static_cast<Document*>(pNode))->getDocumentElement());
            cout << "Document end:" << endl;
            break;

        // 文書要素ノードの出力
        case Node::ELEMENT_NODE:
            cout << "Element start: " << MBSTR(pNode->getNodeName()) << endl;

            // 属性の出力
            attrs = pNode->getAttributes();
            if (attrs != 0) {
                for (int i = 0; i < attrs->getLength(); i++) {
                    Node* attr = attrs->item(i);
                    cout << " " << MBSTR(attr->getNodeName())

```

```

        << "¥" << MBSTR(attr->getNodeValue()) << "' ' << endl;
    }
}
// 子ノードの出力
children = pNode->getChildNodes();
if (children != 0) {
    for (int i = 0; i < children->getLength(); i++) {
        print(children->item(i));
    }
}
cout << "Element end: " << MBSTR(pNode->getNodeName()) << endl;
break;

// テキストノードの出力
case Node::TEXT_NODE:
    cout << "Text: ¥" << MBSTR(pNode->getNodeValue()) << "' ' << endl;
    break;
}
}
}

```

このプログラムをコンパイルする場合には、コンパイラに対して iPEX のインクルードファイルやライブラリの場所を指定するために、次のような makefile を作成します。

DOMEcho1.cpp と makefile を同じディレクトリに置いて、"make"とタイプすれば自動的にコンパイル・リンクが実行されます。

コンパイルした結果としてエラーもしくはワーニングが報告された場合には、出力されたメッセージを参考にプログラムを修正して、コンパイルし直してください。コンパイルした結果、プログラムにエラーがなければ、コンパイルされたバイナリコードを含む DOMEcho1 という実行形式のファイルが生成されます。

```

MAKEFILE = makefile
SHELL = /bin/sh
CCC = g++

CCFLAGS = -O
DEFINES = -DIPEX_LINUX -DNDEBUG
LDFLAGS =

TARGET = DOMEcho1
IPEXDIR = /usr/local/iPEX2_DE
IPEXLIB = -lipex
OBS = DOMEcho1.o

all:
    (MAKE) -f (MAKEFILE) target

target: (TARGET)

clean:
    -rm -f *.o core

rebuild: clean all

%.o : %.cpp
    (CCC) (CCFLAGS) (DEFINES) ¥
    -I(IPEXDIR)/include ¥
    -c -o @ <

(TARGET): (OBS)

```

```
(CCC) (LDFLAGS) -o @ ^ ¥  
-L(IPEXDIR)/lib ¥  
(IPEXLIB) -lstdc++
```

(7) プログラムの実行

コンパイルの結果、実行ファイルが生成されたならば、いよいよプログラムを実行できます。

ここでは、XML 文書とプログラムの実行結果との対応を示すために、リスト 2.2 に示すスケジュール管理データの断片をサンプルデータとして使用します。

リスト 2.2 サンプル・データ(schedule.xml)

```
<?xml version="1.0" encoding="EUC-JP"?>  
<eventlist staff="飯塚富雄">  
  <event type="work">  
    <start/>  
    <info>入社</info>  
    <place>Duo&lt; ;早稲田&gt;</place>  
  </event>  
</eventlist>
```

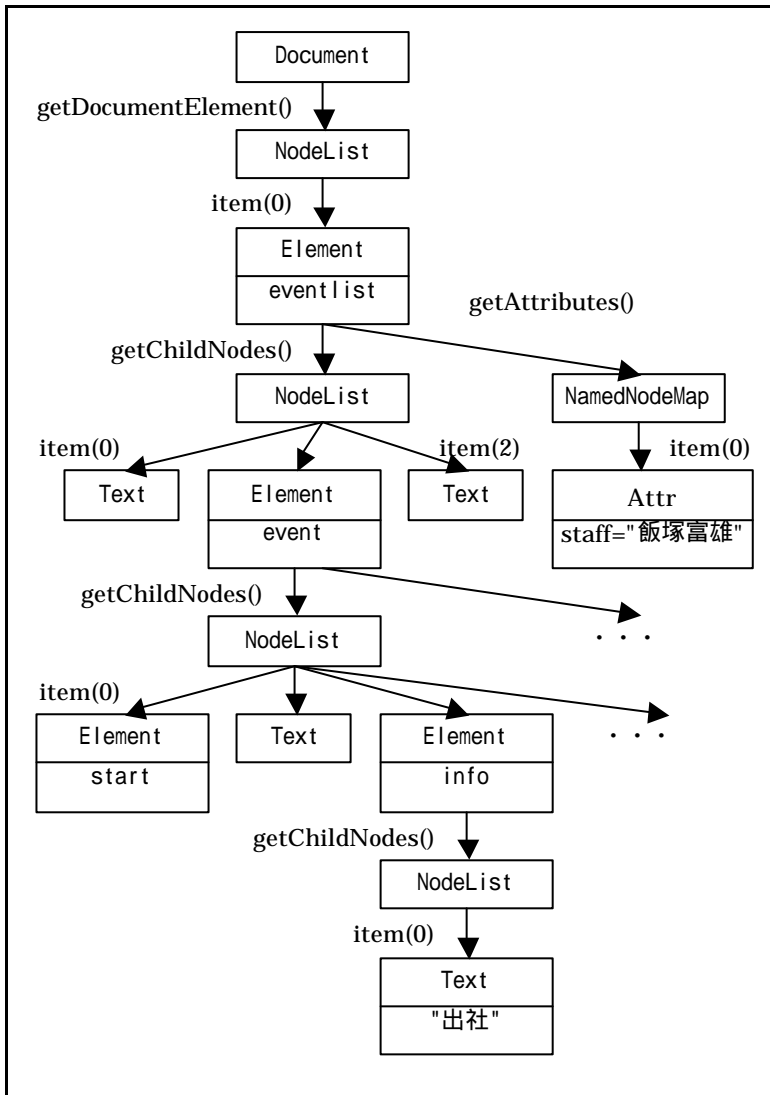
リスト 2.2 に示したサンプルデータのファイル(schedule.xml)が DOMEcho1 ファイルと同じディレクトリにある場合、右のようにプログラムを実行します。

```
./DOMEcho1 schedule.xml
```

実際に出力される結果は、リスト 2.3 に示したようになります。出力された結果のもとになる DOM のドキュメントオブジェクトの一部を図 2.5 に併記します。

出力されるメッセージの順番は、print メソッドで処理した Node の順序になっています。つまり、最上位の文書要素から順に下位の文書要素をたどり、同一の階層にある文書要素については XML 文書内での出現順にメッセージを出力しています。

図 2.5 サンプルデータに対する DOM



リスト 2.3 DOMEcho1 の出力

```

Document start:
Element start: eventlist
  staff="飯塚富雄"
Text: "
"
Element start: event
  type="work"
Text: "
"
Element start: start
Element end: start
Text: "
"
Element start: info
Text: "出社"
Element end: info
Text: "
"
Element start: place
Text: "Duo<早稲田>"
Element end: place
Text: "
"
Element end: event
Text: "
"
Element end: eventlist
Document end:
    
```

2.4 DOM から XML 文書を出力する

この節では、前節のプログラムを拡張して、XML ファイルを読み込んで DOM のオブジェクトを作成し、これに対応した XML 形式のデータを出力するプログラムを作成します。

iPEX には XMLWriter というクラスが定義されており、作成した DOM オブジェクトを簡単に XML 形式のテキストとして出力できます。

(1) XML 出力の呼び出し

DOMEcho2 というプログラムを、前節の DOMEcho1 というプログラムのコードをベースにして作成します。入力された XML 文書のパースまでは DOMEcho1 とほとんど同じです。異なる点は、DOM 用 XML パーサが生成した DOM オブジェクトである Document のインスタンス(pDoc)の内容を XMLWriter に渡して XML 文書を出力していることです。

まず、OutputStreamByFile クラスにより出力ストリーム ofs を作成します。出力先のファイルは簡単のために標準出力(STDOUT)とします。また、出力するエンコーディングは EUC を指定します。次に、XMLWriter のインスタンスとして writer を作成します。引数として、上で作成した出力ストリーム ofs を指定します。最後に、すでに生成している DOM オブジェクト pDoc を write()により XML ファイルとして書き出します。

```

OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
if (!ofs) {
    cerr << "Cannot open stdout" << endl;
    return 1;
}
XMLWriter writer(&ofs);
if (!writer.write(pDoc.get())) {
    cerr << "Cannot write to stdout" << endl;
    return 1;
}

```

ここまでで説明したプログラムの断片と、DOMEcho1 と同様な XML ファイルから DOM オブジェクトを生成するプログラムをまとめたものをリスト 2.4 に掲載します。

リスト 2.4 DOMEcho2.cpp

```

#include <iostream>
#include <memory>
using namespace std;

#include <ipexdom.h>
using namespace iPEX;

#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBSString(x,L"euc-jp")

int main(int argc, char* argv[])
{
    EUCJPCConverterFactory fEUCJP;

    if (argc != 2) {
        cerr << "Usage: DOMEcho2 filename" << endl;
        return 1;
    }
    // 入力文書をオープン
    DOMString pathname = WSTR(argv[1]);
    InputSteamByFile ifs(pathname.c_str());
    if (!ifs) {
        cerr << "Cannot open file: " << MBSTR(pathname) << endl;
        return 1;
    }
    // 入力文書をパース
    XMLReader reader(&ifs);
    auto_ptr<Document> pDoc(iPEXDocument::createDocumentObject());
    if (!reader.read(pDoc.get())) {
        cerr << "Cannot read xml: " << MBSTR(pathname) << endl;
        return 1;
    }
    // 出力文書をオープン
    OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
    if (!ofs) {
        cerr << "Cannot open stdout" << endl;
        return 1;
    }
    // XML 文書に整形して出力
    XMLWriter writer(&ofs);
    if (!writer.write(pDoc.get())) {

```

```
    cerr << "Cannot write to stdout" << endl;
    return 1;
}
return 0;
}
```

(2) プログラムのコンパイル

この例のプログラムをコンパイルするために、次のような makefile を作成します。

DOMEcho1 と同様に、DOMEcho2.cpp と makefile を同じディレクトリにおいて"make" とタイプすれば実行形式のファイル DOMEcho2 が作成されます。

```
MAKEFILE = makefile
SHELL    = /bin/sh
CCC      = g++

CCFLAGS  = -O
DEFINES  = -DIPEX_LINUX -DNDEBUG
LDFLAGS  =

TARGET   = DOMEcho2
IPEXDIR  = /usr/local/iPEX2_DE
IPEXLIB  = -lipex
OBSJ     = DOMEcho2.o

all:
    (MAKE) -f (MAKEFILE) target

target: (TARGET)

clean:
    -rm -f *.o core

rebuild: clean all

%.o : %.cpp
    (CCC) (CCFLAGS) (DEFINES) ¥
    -I(IPEXDIR)/include ¥
    -c -o @ <

(TARGET): (OBSJ)
    (CCC) (LDFLAGS) -o @ ^ ¥
    -L(IPEXDIR)/lib ¥
    (IPEXLIB) -lstdc++
```

(3) プログラムの実行

このプログラムを実行するために、コマンドライン引数として XML 文書のファイルパス名をとるので、次のように起動します。

```
./DOMEcho2 schedule.xml
```

プログラムを実行すると、標準出力に schedule.xml と同じ XML 文書が出力されます。この出力結果をファイルへ出力する場合には、次のように標準出力の出力先をリダイレクトします。

```
./DOMEcho2 schedule.xml > 出力先のファイル名
```

2.5 DOM から特定のデータを抽出する

この節では、DOM のドキュメントオブジェクトから特定の条件に適合する文書要素を抽出し、その結果を XML 文書として出力するプログラムを作成します。

この例では、リスト 2.5 の XML 文書を使用します。このプログラムは、スケジュール管理の対象となっているメンバ全員の情報が格納されている XML 文書から、特定の年月日に予定されているイベントを含む文書要素を抽出します。

リスト 2.5 schedule2.xml

```
<?xml version="1.0" encoding="EUC-JP"?>
<schedule>
<eventlist staff="飯塚富雄">
  <event type="work">
    <start>
      <date>
        <year>2000</year>
        <month>10</month>
        <day>10</day>
        <oftheweek>月</oftheweek>
      </date>
    </start>
    <info>入社</info>
    <place>Duo&lt;早稲田&gt;</place>
  </event>
</eventlist>
<eventlist staff="横山至治">
  <event type="work">
    <start>
      <date>
        <year>2000</year>
        <month>10</month>
        <day>11</day>
        <oftheweek>火</oftheweek>
      </date>
    </start>
    <info>入社</info>
    <place>Duo&lt;早稲田&gt;</place>
  </event>
</eventlist>
</schedule>
```

(1) XML データの抽出

XML パーサのドライバとなる main() 含む DOMEcho3.cpp は、前節で作成した DOMEcho2.cpp に若干のコードを追加します(リスト 2.6)。

まず、抽出するデータを特定するために、コマンドライン引数から年、月、日を指定できるようにします。

そして、DOM 用 XML パーサにより生成された Document のインスタンスを XMLWriter へ出力する直前に、DOMChooser というクラスのメンバ関数 chooser() を起動して、DOM オブジェクトに格納されているデータから、指定された年月日のイベント以外のデータを削除することにします。

リスト 2.6 DOMEcho3.cpp

```
#include <iostream>
#include <memory>
using namespace std;
```

```

#include <ipexdom.h>
using namespace iPEX;

#include "DOMChooser.h"

#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBString(x,L"euc-jp")

int main(int argc, char* argv[])
{
    EUCJPConverterFactory fEUCJP;

    if (argc != 5) {
        cerr << "Usage: DOMEcho3 filename yyyy mm dd" << endl;
        return 1;
    }

    DOMString pathname = WSTR(argv[1]);
    DOMString year = WSTR(argv[2]);
    DOMString month = WSTR(argv[3]);
    DOMString days = WSTR(argv[4]);

    // 入力文書のオープン
    InputSteamByFile ifs(pathname.c_str());
    if (!ifs) {
        cerr << "Cannot open file: " << MBSTR(pathname) << endl;
        return 1;
    }
    // 入力文書のパース
    XMLReader reader(&ifs);
    auto_ptr<Document> pDoc(IPEXDocument::createDocumentObject());
    if (!reader.read(pDoc.get())) {
        cerr << "Cannot read xml: " << MBSTR(pathname) << endl;
        return 1;
    }
    // 指定されたデータの抽出
    DOMChooser chooser;
    chooser.choose(pDoc.get(), year, month, days);

    // 出力文書のオープン
    OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
    if (!ofs) {
        cerr << "Cannot open stdout" << endl;
        return 1;
    }
    // XML 文書に整形して出力
    XMLWriter writer(&ofs);
    if (!writer.write(pDoc.get())) {
        cerr << "Cannot write to stdout" << endl;
        return 1;
    }
    return 0;
}

```

(2) DOMChooser クラスの宣言

ここでは、DOMChooser クラスの宣言を DOMChooser.h という名前のファイルで行います(リスト 2.7)。各メンバの定義は、DOMChooser.cpp ファイルの中で行ないます。その詳細については次節以降で説明します。

リスト 2.7 DOMChooser.h

```
class DOMChooser {
public:
    void choose (Document* pDoc, DOMString& year, DOMString& month, DOMString& days);

private:
    bool choose (Element* eventlist, DOMString& year, DOMString& month, DOMString& days);
    bool isTarget (Element* event, DOMString& year, DOMString& month, DOMString& days);
    DOMString getTargetText (Element* element, DOMString& targetName);
};
```

(3) eventlist 要素の走査

DOMChooser クラス(リスト 2.8)で定義する最初の choose()関数は、Document のインスタンスに含まれる子要素である schedule 要素を順に走査して、eventlist 要素を順に処理していきます。

この例では、前節まで利用していた getChildNodes メソッドではなく、getFirstChild メソッドで先頭の子要素を取得して、順次 getNextSibling メソッドで次の子要素を取得します。指定した日付と一致する event 要素が含まれているかどうかは、二番目の choose メソッドを呼び出した結果として取得します。指定された日付のイベントが eventlist にひとつも含まれていない場合、その eventlist は削除するようにしています。getNextSibling を呼び出して次の要素を順に取得する場合、現在の要素を削除する前に次の要素を取得しておく必要があります。

(4) 文書要素の削除

二番目の choose()では、eventlist 要素に含まれる子要素を順に走査しながら、event の日付が指定された日付と一致しているかを isTarget()を呼び出して判定します。日付が一致する event 要素でなければ、その event 要素を削除します。

(5) 文書要素の判定

個々の event 要素について year、month、day 要素が指定された日付と一致するかどうかは、isTarget()と、getTargetText()でチェックします。isTarget()では、getTargetText()を呼び出して、event 要素に含まれる year、month、day 要素のテキスト部分の文字列を取得し、それぞれ指示された値に一致するかをチェックしています。

getTarget()では、引数で指定されたタグ名を持つ文書要素を取得し、対応するテキストを抽出して返します。DOM には、ある要素の下位要素に特定の要素名を持つ文書要素をすべてリストアップしてくれる getElementByName() という関数が用意されています。この関数を使えば、文書要素の階層関係を意識せずにダイレクトに特定の文書要素に対応したデータを取得できます。つまり、event 要素において、start 要素 date 要素 year 要素といった階層を考慮する必要がなくなります。ただ、このメソッドは指定された要素以下のすべての要素に対するデータのタグ名を比較していくため多少のオーバーヘッドをとまいます。

year、month、あるいは days に相当するデータを取得できたならば、その子要素として格納されている Text のインスタンスが保持しているテキストを返します。その際に、該当する文書要素に子要素がひとつだけ含まれていて、それが Text のインスタンスであるかどうかをチェックしています。

リスト 2.8 DOMChooser.cpp

```
#include <memory>
```

```

#include <string>
using namespace std;

#include <ipexdom.h>
using namespace iPEX;

#include "DOMChooser.h"

#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBString(x,L"euc-jp")

void DOMChooser::choose (Document* pDoc, DOMString& year, DOMString& month, DOMString& days)
{
    // ルート要素を取得
    Element* schedule = pDoc->getDocumentElement();

    // ルート要素の子要素を順に走査
    Node* eventlist = schedule->getFirstChild();
    while (eventlist != 0) {

        // eventlist ノードの場合、該当データかを判断
        bool found = false;
        if ((eventlist->getNodeType() == Node::ELEMENT_NODE)
            && (eventlist->getNodeName() == L"eventlist")) {
            found = choose(static_cast<Element*>(eventlist), year, month, days);
        }
        // 次のノードを取得
        Node* nextNode = eventlist->getNextSibling();

        // 該当外のノードを削除
        if (!found) {
            try {
                schedule->removeChild(eventlist);
            }
            catch (DOMException& e) {
                cerr << "Cannot remove eventlist" << endl;
            }
        }
        eventlist = nextNode;
    }
}

bool DOMChooser::choose (Element* eventlist, DOMString& year, DOMString& month, DOMString& days)
{
    // eventlist 内の子要素を順に走査
    bool result = false;
    Node* event = eventlist->getFirstChild();
    while (event != 0) {

        // event ノードの場合、該当データかを判定
        bool found = false;
        if ((event->getNodeType() == Node::ELEMENT_NODE)
            && (event->getNodeName() == L"event")) {
            found = isTarget(static_cast<Element*>(event), year, month, days);
            if (found) {
                result = true;
            }
        }
    }
    // 次ノードを取得

```

```

Node* nextNode = event->getNextSibling();

// 該当外のノードを削除
if (!found) {
    try {
        eventlist->removeChild(event);
    }
    catch (DOMException& e) {
        cerr << "Cannot remove event" << endl;
    }
}
event = nextNode;
}
return result;
}

bool DOMChooser::isTarget (Element* event, DOMString& year, DOMString& month, DOMString& days)
{
    // 日が不一致の場合、該当買いを返却
    DOMString daysText = getTargetText(event, L"day");
    if (daysText.empty() || (days != daysText)) {
        return false;
    }
    // 月が不一致の場合、該当買いを返却
    DOMString monthText = getTargetText(event, L"month");
    if (monthText.empty() || (month != monthText)) {
        return false;
    }
    // 年が不一致の場合、該当買いを返却
    DOMString yearText = getTargetText(event, L"year");
    if (yearText.empty() || (year != yearText)) {
        return false;
    }
    return true;
}

DOMString DOMChooser::getTargetText (Element* element, DOMString& targetName)
{
    // 指定タグ名のノードを取得
    NodeList* nodes = element->getElementsByTagName(targetName);
    if ((nodes == 0) || (nodes->getLength() != 1)) {
        return L"";
    }
    // 先頭の子ノードを取得
    Element* targetNode = static_cast<Element*>(nodes->item(0));
    if (targetNode == 0) {
        return L"";
    }
    // 子ノードを取得
    NodeList* children = targetNode->getChildNodes();
    if ((children == 0) || (children->getLength() != 1)) {
        return L"";
    }
    // 先頭の子ノードのテキストを返却
    Node* child = children->item(0);
    if ((child == 0) || (child->getNodeType() != Node::TEXT_NODE)) {
        return L"";
    }
    return child->getNodeValue();
}

```

```
}
```

(6) プログラムのコンパイル

この例のプログラムをコンパイルして実行するために、以下の makefile を作成します。

```
MAKEFILE = makefile
SHELL    = /bin/sh
CCC      = g++

CCFLAGS  = -O
DEFINES  = -DIPEX_LINUX -DNDEBUG
LDFLAGS  =

TARGET   = DOMEcho3
IPEXDIR  = /usr/local/iPEX2_DE
IPEXLIB  = -lipex
OBJJS    = DOMEcho3.o DOMChooser.o

all:
    (MAKE) -f (MAKEFILE) target

target: (TARGET)

clean:
    -rm -f *.o core

rebuild: clean all

%.o : %.cpp
    (CCC) (CCFLAGS) (DEFINES) ¥
    -I(IPEXDIR)/include ¥
    -c -o @ <

(TARGET): (OBJJS)
    (CCC) (LDFLAGS) -o @ ^ ¥
    -L(IPEXDIR)/lib ¥
    (IPEXLIB) -lstdc++
```

DOMEcho3.cpp、DOMChooser.h、DOMChooser.cpp、makefile の各ファイルと同じディレクトリに置き、"make"とタイプすれば実行ファイル DOMEcho3 が作成されます。

(7) プログラムの実行

プログラムがコンパイルできたならば、次のようにプログラムを実行します。

```
./DOMEcho3 schedule2.xml 2000 10 11
```

2.6 DOM オブジェクトを構築する

この節では DOM のドキュメントオブジェクトを構築し、XML 文書として出力するプログラムを作成します。DOM の構築を紹介するという目的で、リスト 2.2 のサンプルデータと同じ DOM を構築し、XML 文書を生成するプログラムを作成します。

(1) Document の生成

main()関数では、XMLGenerator クラスのインスタンスを生成して、makedom()というメンバ関数を呼び出して生成された Document のインスタンスを取得します。XML 形式での出力は前の節で説明した XMLWriter を利用しています。ただし、ここでは XMLWriter への第 3 引数に true を渡し、出力時にインデントを行なうように指定しています。

```

int main(int argc, char* argv[])
. . .
XMLGenerator xmlGenerator;
auto_ptr<Document> pDoc(xmlGenerator.makedom());
OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
if (!ofs) {
    cerr << "Cannot open stdout" << endl;
    return 1;
}
XMLWriter writer(&ofs, L"", true);
if (!writer.write(pDoc.get())) {
    cerr << "Cannot write to stdout" << endl;
    return 1;
}
. . .

```

XMLGenerator のコンストラクタでは、IPEXDocument::createDocumentObject()で生成した Document へのポインタをプライベート変数 pDoc に格納します。

```

class XMLGenerator
{
public:
XMLGenerator()
{
    pDoc = IPEXDocument::createDocumentObject();
}
. . .

```

(2) 文書要素と属性の追加

Document を生成したならば、ルート要素である eventlist 要素を子要素として追加します。この際にプライベート関数の append() を使用します。

さらに、setAttribute() 関数により作業者名を eventlist の staff 属性として設定しています。

```

. . .
Element* eventlist = append(pDoc, WSTR("eventlist"));
try {
    eventlist->setAttribute(WSTR("staff"), WSTR("飯塚富雄"));
}
catch (DOMException& e) {
    cerr << "Cannot set attribute to eventlist" << endl;
}
. . .

```

次に、同様にして eventlist の子要素として event 要素を追加し、type 属性の値を設定します。

```

Element* event = append(eventlist, WSTR("event"));
try {
    event->setAttribute(WSTR("type"), WSTR("work"));
}
catch (DOMException& e) {
    cerr << "Cannot set attribute to event" << endl;
}

```

さらに、event の子要素として、start、info、place の各要素を追加します。

```

Element* start = append(event, WSTR("start"));
Element* info = append(event, WSTR("info"), WSTR("会社"));

```

```
Element* place = append(event, WSTR("place"), WSTR("<早稲田>"));
```

下位の要素を生成するために利用する `append()` は、二種類の関数を用意しています。ひとつは、追加先の要素と追加する要素名を引数としており、あとで子要素が追加される要素を生成します。もうひとつは、2 個の引数に加えて文書要素のテキストを引数とした関数です。

具体的には、最初の `append()` では `createElement()` で新たな要素を生成し、`appendChild()` で、指定された要素の子供として追加します。2 番目の `append()` では、最初の `append()` を呼び出して子要素を生成し、さらに `createTextNode()` でテキストノードを生成して、子要素の子ノードとして追加します。

```
Element* append(Node* pNode, const DOMString& tagName)
{
    Element* pElement;
    try {
        pElement = pDoc->createElement(tagName);
    }
    catch (DOMException& e) {
        cerr << "Cannot create element: " << MBSTR(tagName) << endl;
        return 0;
    }
    try {
        pNode->appendChild(pElement);
    }
    catch (DOMException& e) {
        cerr << "Cannot append element: " << MBSTR(tagName) << endl;
        return 0;
    }
    return pElement;
}

Element* append(Node* pNode, const DOMString& name, const DOMString& text)
{
    Element* pElement = append(pNode, name);
    Text* textNode;
    try {
        textNode = pDoc->createTextNode(text);
    }
    catch (DOMException& e) {
        cerr << "Cannot create text node: " << MBSTR(name) << endl;
        return 0;
    }
    try {
        pElement->appendChild(textNode);
    }
    catch (DOMException& e) {
        cerr << "Cannot append text node: " << MBSTR(name) << endl;
        return 0;
    }
    return pElement;
}
```

以上で説明したコードの断片をまとめたのが `XMLGenerator.cpp` です(リスト 2.9)。

リスト 2.9 XMLGenerator.cpp

```
#include <iostream>
```

```

#include <memory>
using namespace std;

#include <ipexdom.h>
using namespace IPEX;

#define WSTR(x) IPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) IPEX::Character::toMBString(x,L"euc-jp")

class XMLGenerator
{
public:
    XMLGenerator()
    {
        // 新規の文書オブジェクトを生成
        pDoc = IPEXDocument::createDocumentObject();
    }

    Document* makedom()
    {
        // ルート要素の eventlist ノードを追加
        Element* eventlist = append(pDoc, WSTR("eventlist"));
        try {
            eventlist->setAttribute(WSTR("staff"), WSTR("飯塚富雄"));
        }
        catch (DOMException& e) {
            cerr << "Cannot set attribute to eventlist" << endl;
        }
        // event ノードを追加
        Element* event = append(eventlist, WSTR("event"));
        try {
            event->setAttribute(WSTR("type"), WSTR("work"));
        }
        catch (DOMException& e) {
            cerr << "Cannot set attribute to event" << endl;
        }
        // event の各下位要素を追加
        Element* start = append(event, WSTR("start"));
        Element* info = append(event, WSTR("info"), WSTR("出社"));
        Element* place = append(event, WSTR("place"), WSTR("<早稲田>"));
        return pDoc;
    }

private:
    Document* pDoc;

    Element* append(Node* pNode, const DOMString& tagName)
    {
        // 指定のタグ名を持つノードを生成
        Element* pElement;
        try {
            pElement = pDoc->createElement(tagName);
        }
        catch (DOMException& e) {
            cerr << "Cannot create element: " << MBSTR(tagName) << endl;
            return 0;
        }
        // 生成したノードを子ノードとして追加
        try {

```

```

    pNode->appendChild(pElement);
}
catch (DOMException& e) {
    cerr << "Cannot append element: " << MBSTR(tagName) << endl;
    return 0;
}
return pElement;
}

Element* append(Node* pNode, const DOMString& name, const DOMString& text)
{
    // 指定のテキストノードを生成
    Element* pElement = append(pNode, name);
    Text* textNode;
    try {
        textNode = pDoc->createTextNode(text);
    }
    catch (DOMException& e) {
        cerr << "Cannot create text node: " << MBSTR(name) << endl;
        return 0;
    }
    // 生成したノードを子ノードとして追加
    try {
        pElement->appendChild(textNode);
    }
    catch (DOMException& e) {
        cerr << "Cannot append text node: " << MBSTR(name) << endl;
        return 0;
    }
    return pElement;
}
};

int main(int argc, char* argv[])
{
    EUCJPCConverterFactory fEUCJP;

    // 新規の DOM を生成
    XMLGenerator xmlGenerator;
    auto_ptr<Document> pDoc(xmlGenerator.makedom());

    // 出力文書をオープン
    OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
    if (!ofs) {
        cerr << "Cannot open stdout" << endl;
        return 1;
    }
    // XML 文書を整形して出力
    XMLWriter writer(&ofs, L"", true);
    if (!writer.write(pDoc.get())) {
        cerr << "Cannot write to stdout" << endl;
        return 1;
    }
    return 0;
}

```

(3) プログラムのコンパイル

この XML 生成プログラムをコンパイルするために、以下の makefile を用意します。

```

MAKEFILE = makefile
SHELL    = /bin/sh
CCC      = g++

CCFLAGS  = -O
DEFINES  = -DIPEX_LINUX -DNDEBUG
LDFLAGS  =

TARGET   = XMLGenerator
IPEXDIR  = /usr/local/iPEX2_DE
IPEXLIB  = -lipex
OBJJS    = XMLGenerator.o

all:
    (MAKE) -f (MAKEFILE) target

target: (TARGET)

clean:
    -rm -f *.o core

rebuild: clean all

%.o : %.cpp
    (CCC) (CCFLAGS) (DEFINES) ¥
    -I(IPEXDIR)/include ¥
    -c -o @ <

(TARGET): (OBJJS)
    (CCC) (LDFLAGS) -o @ ^ ¥
    -L(IPEXDIR)/lib ¥
    (IPEXLIB) -lstdc++

```

この makefile と XMLGenerator.cpp を同じディレクトリに置き、"make"とタイプすれば実行ファイル XMLGenerator が作成されます。

(4) プログラムの実行

プログラムを正常にコンパイルできたならば、以下のようにコマンドを実行してください。

```
./XMLGenerator
```

標準出力にリスト 2.2 と同じ XML ファイルが出力されます。XMLWriter インデントはタブで行なわれるので、インデントの深さが違って見えます。

3 SAX パーサによる XML 文書処理

前章では、DOM に準拠したドキュメントオブジェクトを扱える XML パーサを使って XML 文書処理を概説しました。本章では、XML 文書を処理するためのもう一つのインタフェースである SAX を用いた XML 文書処理の概要を説明します。DOM に対応したパーサを利用して作成したプログラムと同様な機能を実現するプログラムを作成し、SAX インタフェースとの違いを対比させながら利用例を示します。

3.1 SAX パーサ

SAX は、DOM と同様に、XML 文書をパースして解析処理を行うアプリケーションプロ

グラムで利用する標準インタフェースです。SAX パーサは、DOM に対応したパーサとは異なり、XML 文書を順次シーケンシャルに読み込みながら、XML のタグ(開始タグ、終了タグ、空要素タグ)を検出するごとにユーザがアドインした各種ハンドラを起動します。アプリケーションからは、SAX インタフェースで規定されたメソッドを実装したハンドラを用意することによって、XML 文書进行处理できます。

3.2 SAX パーサの特徴

SAX パーサの特徴を理解するために、DOM 準拠の XML パーサとを対比させて整理してみましょう。

DOM に準拠した XML パーサを使用する場合には、XML パーサが XML 文書全体をパースして DOM を構築してからでないと DOM のオブジェクトを操作できませんでした。つまり、XML 文書进行处理する時点では、XML 文書全体が DOM のドキュメントオブジェクトとして構築されているわけです。そのため、XML 文書の物理的な順序に制約を受けずに他の文書要素を参照できるので、複雑な文書処理を効率的にプログラミングできます。もちろん、独自に DOM のツリーを検索したり更新したりするためのコードを書かなければなりませんが、文書要素の階層構造にしたがったデータの管理は DOM を通じて提供されているので、比較的簡単にコードを書くことができます。

それに対して SAX パーサを使用した場合に、アプリケーションプログラムは、XML パーサが XML 文書全体のパースを完了するのを待たずに、XML 文書の処理を開始できます。というよりは、XML パーサのパース中に処理を実行することになります。つまり、XML パーサは、XML 文書を順次シーケンシャルに読み込みながら、XML のタグ(開始タグ、終了タグ、空要素タグ)を検出するごとに、ユーザが XML パーサに登録した各ハンドラのメソッドを逐次起動します。

ユーザが作成したハンドラは、パース中にタグが出現することがトリガーとなって XML パーサから呼び出されるので、イベントドリブン型で処理を記述する必要があります。ユーザが登録したハンドラは、XML 文書の物理的な並びにしたがって呼び出されるため、XML 文書の文書要素を先頭から順に処理していく場合には効率的です。

また、SAX パーサは、DOM パーサが DOM を構築する際のオーバヘッドが無いため、DOM を利用した場合に比べて処理速度的にも記憶領域的にもコストが低くなるといった利点があります。その一方で、文書要素の物理的な出現順序を無視した前方参照や後方参照といった操作を行おうとすると、そのための情報を独自に管理するコードを書かなければならないため、プログラムが複雑になる傾向があります。

以上のように、DOM と SAX では一長一短な特徴があるので、どちらを利用すべきか判断に迷うことがあるかもしれません。

次の場合には、DOM を利用せずに、SAX を利用したほうが良いかもしれません。

- ・ 文書要素を物理的な出現順にしたがって処理する場合
- ・ 特定の文書要素しか処理の対象にしない場合
- ・ DOM を構築するための記憶領域を節約する必要がある場合

3.3 SAX パーサの構成

SAX インタフェースを利用する場合に独自のハンドラを用意しなければならないことを述べましたが、SAX のパーサは次のようなクラスから構成されます(図 3.1)。各クラスの詳細はマニュアルを参照してください。

- ・ XMLReader
- ・ ContentHandler
- ・ DTDHandler
- ・ EntityResolver
- ・ ErrorHandler

SAX インタフェースを利用する場合、最初に XMLReader を用いてパーサのインスタンスを生成します。パーサを起動すると、パーサは XML 文書を先頭から順にパースしながら、XML 文書の記述内容にしたがって、ContentHandler、ErrorHandler、DTDEventHandler、EntityResolver といったインタフェースを実装したコールバック関数を呼び出します。これらのハンドラを独自に用意しない場合でも、DefaultHandler クラスによって実装されているデフォルトのハンドラが起動されるようになっています。

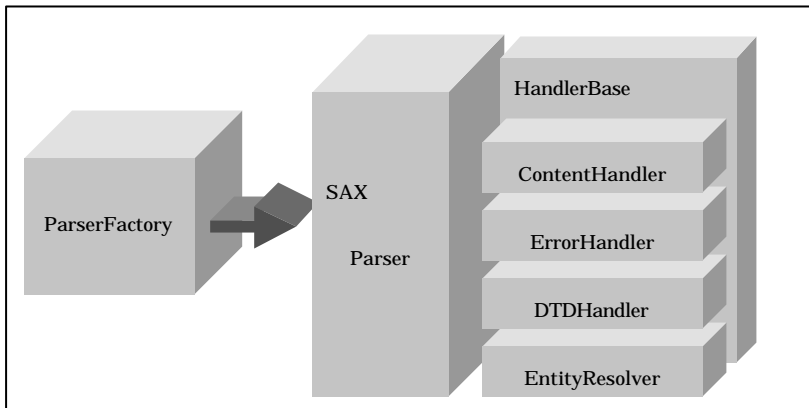


図 3.1 SAX パーサの基本構成

3.3.1 ContentHandler

ContentHandler クラスは、startDocument()、endDocument()、startElement()、endElement()、characters()、processingInstruction()というメンバ関数を定義しています。startDocument()と endDocument()は、それぞれ XML 文書の最初と最後に起動されます。startElement()と endElement()は、それぞれ XML のタグが検出された場合に、文書要素の最初と最後に起動されます。characters()は、XML 文書内のテキストを検出することに起動されます。processingInstruction()は、処理命令を検出することに起動されます(図 3.2)。

3.3.2 ErrorHandler

ErrorHandler クラスは、error()、fatalError()、warnig()というメンバ関数を定義しています。これらの関数は、パース中に発生したさまざまなエラーを処理するために起動されます。

3.3.3 DTDHandler

DTDHandler クラスは、DTD 内の各種定義を処理する際に起動されます。

3.3.4 EntityResolver

EntityResolver クラスは、URI によって指定されたデータを特定する際に起動される resolveEntity()というメンバ関数定義しています。URI は、たいていの場合 URL で指定されますが、WEB 空間内でユニークな公開識別子や名前によって指定されることもあります。

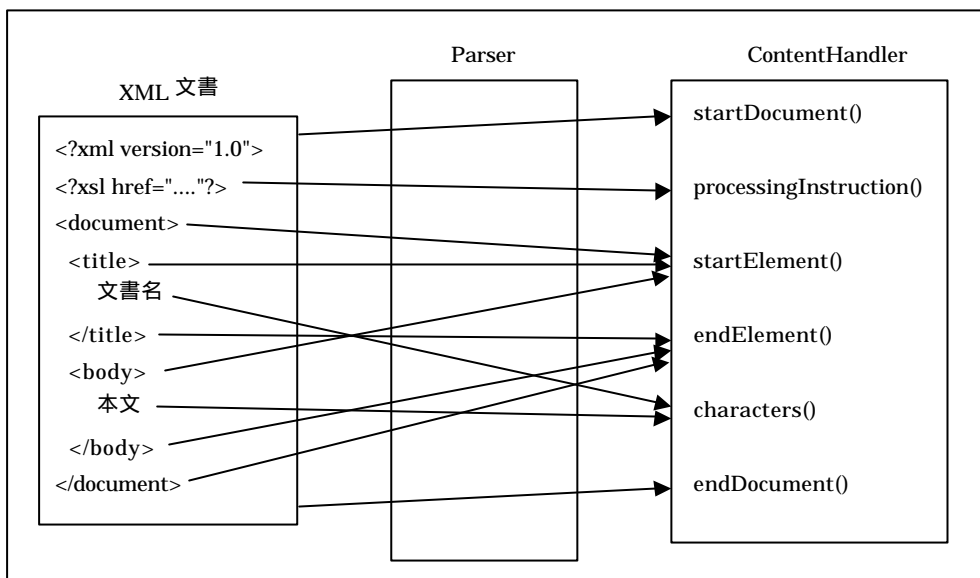


図 3.2 SAX パーサと ContentHandler

3.4 SAX パーサによる XML 文書のパース

SAX API を利用した XML 文書の処理方法を理解するために、非検証パーサを利用して、各ハンドラを起動させて、各ハンドラへ受け渡される情報を表示するプログラムを作成します。

(1) SAX クラスのインクルード

SAX API を定義したインタフェース、またはそのインタフェースを実装したクラスを利用する場合には、以下のようにヘッダファイル ipexsax.h をインクルードする必要があります。ここで定義されているクラスなどは、すべて iPEX::SAX ネームスペースの中で定義されているので、これらのクラスを使用するためには、iPEX::SAX::を明示的に使うか、using 宣言をする必要があります。ここでは std および iPEX ネームスペースと共に using 宣言を使うことにします。

```
#include <ipexsax.h>
using namespace iPEX;
using namespace iPEX::SAX;
using namespace std;
```

(2) ContentHandler インタフェースの実装

SAX パーサに独自の処理を追加するために、ContentHandler インタフェースを実装したクラスを作成します。ここでは、ContentHandler インタフェースのデフォルトの振る舞いを実装した DefaultHandler クラスから継承することによってインタフェースを継承させます。

DefaultHandler は、ContentHandler インタフェースで定義されたメンバ関数をすべて実装しているので、必要なコールバック関数のみを実装するだけでよいのですが、ContentHandler インタフェースで定義されているメソッドの起動を確認するために、DefaultHandler の各メンバ関数をオーバーライドする関数を用意します。リスト 2.2 に示したサンプルデータ(schedule.xml)では PI(処理命令)を処理する必要がないので、processingInstruction()関数は省略しています(リスト 3.1)。個々の関数の内容については、あとの説明を参照してください。

リスト 3.1 SAXEcho1.cpp の基本構造

```
class SAXHandler : public DefaultHandler
{
public:
    virtual void startDocument()
    {
    }
    virtual void endDocument()
    {
    }
    virtual void startElement(const SAXString& strNamespaceURI,
                              const SAXString& strLocalName,
                              const SAXString& strQName,
                              const Attributes* pAttributes)
    {
    }
    virtual void endElement(const SAXString& strNamespaceURI,
                             const SAXString& strLocalName,
                             const SAXString& strQName)
    {
    }
    virtual void characters(const SAXString& strChar)
    {
    }
};
```

(3) main()の実装

このプログラムを C++アプリケーションとして実行できるようにするために、main()関数を実装します。また、プログラムで処理すべき XML 文書をコマンドライン引数から取得するため、引数の数が 1 でない場合に標準エラー出力へメッセージを出力してプログラムを終了させることにします。

XML 文書処理に先立って、XMLReader のインスタンスを生成します。

XMLReader のインスタンスを生成したならば、それに各イベントを処理するハンドラ SAXHandler を登録します。このクラスは、デフォルトのハンドラを定義している DefaultHandler クラスから継承させているので、このクラスのインスタンスを生成して XMLReader に登録します。

```
XMLReader reader;
...
SAXHandler handler;
reader.setContentHandler(&handler);
```

(4) XML 文書のパース

XML 文書のファイル名は、コマンドラインの第一引数としてを取得するので、XMLReader の parse()関数を呼び出す際に、このファイル名をもとに InputStreamByFile クラスのインスタンス ifs を生成して引数として引き渡します。なお、SAX 使用時には文字列を、SAXString クラスを用いて表現します。そのため、DOM の場合と同様に、EUC コード文字列->SAXString、SAXString->EUC コード文字列変換のためのマクロ、WSTR と MBSTR を定義しておきます。また、EUC コードサポートを有効にするために、EUCJPConverterFactory のインスタンスを定義します。

```
...
#defineWSTR(x) iPEX::Character::toWString(x,L"euc-jp")
#defineMBSTR(x) iPEX::Character::toMString(x,L"euc-jp")
```

```

. . .
int main (int argc, char* argv[])
{
    EUCJPCConverterFactory fEUCJP;
    . . .
    SAXString filename = WSTR(argv[1]);
    InputStreamByFile is(filename.c_str());
    if (!is) {
        cerr << "Cannot open file: " << MBSTR(filename) << endl;
        return 1;
    }
    . . .
    reader.parse(&is);
}

```

parse()を呼び出してパースを開始させると、ハンドラとして登録したインスタンスの関数が順次呼び出されることになります。

(5) ハンドラで実装するメンバ関数の機能

startDocument()と endDocument()については、関数が呼び出されたことを示すメッセージを出力させます。

startElement()については、関数が呼び出されるトリガーになった文書要素のタグ名を出力させ、pAttributes に登録された属性の名前と値を出力させます。

endElement()については、タグの名前を出力させます。

characters()については、パーサから引数として引き渡されたデータを文字列に変換し、二重引用符でくくって出力させます。

(6) プログラムのコンパイル

ここまでで説明したコードの断片を一つのプログラムにまとめたものをリスト 3.2 に掲じます。

リスト 3.2 SAXEcho1.cpp

```

#include <ipexsax.h>
#include <ipexchar.h>
#include <ipexstream.h>
using namespace iPEX;
using namespace iPEX::SAX;

#include <iostream>
using namespace std;

#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBString(x,L"euc-jp")

class SAXHandler : public DefaultHandler
{
public:
    virtual void startDocument()
    {
        // 文書開始メッセージを出力
        cout << "startDocument." << endl;
    }

    virtual void endDocument()

```

```

{
    // 文書終了メッセージを出力
    cout << "endDocument." << endl;
}

virtual void startElement(const SAXString& strNamespaceURI,
    const SAXString& strLocalName,
    const SAXString& strQName,
    const Attributes* pAttributes)
{
    // 要素開始メッセージを出力
    cout << "startElement: " << MBSTR(strLocalName) << "." << endl;
    int len = pAttributes->getLength();

    // 属性を出力
    for (int i = 0; i < len; i++) {
        cout << " " << MBSTR(pAttributes->getLocalName(i))
            << "=\"" << MBSTR(pAttributes->getValue(i)) << "\"." << endl;
    }
}

virtual void endElement(const SAXString& strNamespaceURI,
    const SAXString& strLocalName,
    const SAXString& strQName)
{
    // 要素終了を出力
    cout << "endElement: " << MBSTR(strLocalName) << "." << endl;
}

virtual void characters(const SAXString& strChar)
{
    // 文字列を出力
    cout << "characters: ¥" << MBSTR(strChar) << "¥." << endl;
}
};

int main (int argc, char* argv[])
{
    EUCJPCConverterFactory fEUCJP;

    if (argc != 2) {
        cerr << "Usage: SAXEcho1 filename" << endl;
        return 1;
    }
    // 入力文書をオープン
    SAXString filename = WSTR(argv[1]);
    InputSteamByFile is(filename.c_str());
    if (!is) {
        cerr << "Cannot open file: " << MBSTR(filename) << endl;
        return 1;
    }
    // ハンドラを登録
    XMLReader reader;
    SAXHandler handler;
    reader.setContentHandler(&handler);

    // XML 文書のパース
    reader.parse(&is);
}

```

このプログラムをコンパイルする場合には、下のような makefile を作成します。

```
MAKEFILE = makefile
SHELL    = /bin/sh
CCC      = g++

CCFLAGS  = -O
DEFINES  = -DIPEX_LINUX -DNDEBUG
LDFLAGS  =

TARGET   = SAXEcho1
IPEXDIR  = /usr/local/iPEX2_DE
IPEXLIB  = -lipex
OBS      = SAXEcho1.o

all:
    (MAKE) -f (MAKEFILE) target

target: (TARGET)

clean:
    -rm -f *.o core

rebuild: clean all

%.o : %.cpp
    (CCC) (CCFLAGS) (DEFINES) ¥
    -I(IPEXDIR)/include ¥
    -c -o @ <

(TARGET): (OBS)
    (CCC) (LDFLAGS) -o @ ^ ¥
    -L(IPEXDIR)/lib ¥
    (IPEXLIB) -lstdc++
```

SAXEcho1.cpp と makefile を同じディレクトリに置いて、"make"とタイプすれば自動的にコンパイル・リンクが実行されます。コンパイルした結果プログラムにエラーがなければ、SAXEcho1 という実行ファイルが生成されます。

(7) プログラムの実行

プログラムを実行するには、XML 文書とプログラムの実行結果との対応を示すために、スケジュール管理データの断片(リスト 2.5)を使用します。

schedule.xml ファイルが SAXEcho1 ファイルと同じディレクトリにある場合、次のようにプログラムを実行します。

```
./SAXEcho1 schedule.xml
```

このプログラムの実行結果は、リスト 3.3 のようになります。

SAX API を利用したプログラムの出力結果で注目していただきたいのは、次の 2 点です。

1 点目は、<start/>という空要素タグの場合です。start タグは XML 文書内で一つしかありませんが、SAX パーサのハンドラでは startElement()と endElement()が呼び出されている点です。<start/>は、<start></start>と同じ意味を持っているので、ハンドラの呼び出され方が統一されているということは、両方の場合に対応させるために別のコードを書く必要がないということです。2 点目は、開始タグとその子要素の開始タグとのあいだ、終了

タグとその次の要素の開始タグとのあいだ、終了タグとその親要素の終了タグとのあいだに含まれているテキストについても、空白文字であるかどうかに関わらず、characters メソッドが呼び出されている点です。このサンプルデータの場合にはデータを視覚的に構造化しているだけなので無意味に思える空白文字であっても、自然言語で文章などを記述してある場合には空白文字以外の有意なデータが記録されていることがあります。startElement と endElement 間の空白文字についても、なんらかの処理がアプリケーションに要求されることがあります。

リスト 3.3 SAXEcho1 の実行結果

```
startDocument.
startElement: eventlist.
  staff="飯塚富雄.
characters: "
  ".
startElement: event.
  type="work.
characters: "
  ".
startElement: start.
endElement: start.
characters: "
  ".
startElement: info.
characters: "出社".
endElement: info.
characters: "
  ".
startElement: place.
characters: "Duo<早稲田>".
endElement: place.
characters: "
  ".
endElement: event.
characters: "
  ".
endElement: eventlist.
endDocument.
```

4 XSLT を利用した XML 文書変換

本章では、XSL によるスタイルシートを利用して XML 文書を HTML 文書に変換するプログラム、およびプログラムからスタイルシートにパラメータを渡し、そのパラメータにより XML 文書を変換するプログラムの例を紹介します。

4.1 XSLT プロセッサ

多くの場合、XML 文書には表示形式のための情報が含まれていません。XML 文書をブラウザなどで閲覧するためには、表示する際のスタイルを付加する必要があります。表示スタイルを定義した文書を「スタイルシート」といいます。W3C は、XML 文書のスタイルシート記述言語を XSL(Extensible Stylesheet Language)という仕様で規定しています。XSL 自体は、表示スタイル自体と文書変換言語を規定しており、文書変換言語部分は XSLT(XSL Transformations)として勧告仕様となっています。文書の変換方法のみを定義したスタイルシートを「トランスフォーム」ともいいます。また、XSL 仕様の関連仕様として、XML 文書内の文書要素や属性を特定するための XPath(XML Path Language)が勧

告仕様となっています。

XSLT プロセッサは、XML 文書にスタイルシートを適用して、スタイルシートに基づいた文書変換を処理します。XSLT プロセッサを利用して、同一の XML 文書に複数の異なるスタイルシートを適用することにより、同一の文書から異なる形式の文書へ変換できます。たとえば、HTML へ変換するスタイルシートと Compact-HTML へ変換するスタイルシートを用意すれば、同一の XML 文書を WEB ブラウザと i-モード対応の携帯電話で閲覧できます。iPEX は、XSLT プロセッサの機能も内包しており、XSLT version 1.0 と XPath Version 1.0 のすべての仕様をサポートしています。

4.2 XML 文書とスタイルシート

本節では、数名から数十名程度の比較的小さなグループのスケジュール管理を想定したアプリケーションを例に、スケジュール管理のための XML 文書とスタイルシートを説明します。

(1) スケジュール管理用 XML 文書

スケジュール管理の対象となるデータは XML 文書で作成されているものとします。プログラムでは、リスト 4.1 の DTD で定義される XML ファイルを扱うことにします。

ルート要素は `schedule` とします。各メンバのイベントの集合を表わす要素を `eventlist` とし、`schedule` の子とします。この `eventlist` には `staff` という属性を設けてメンバの名前を表わすことにします。`eventlist` には複数の `event` が含まれます。`event` には `type` という属性を設けてイベントの種類を表わすことにします。今回はイベントの種類として、`meeting`、`work`、`study`、`lecture`、`party` の 5 種類を考えます。また、`event` は `start`(開始時間)、`end`(終了時間)、`project`(関連プロジェクト)、`info`(内容、種別)、`place`(場所)の子要素を持つことができることにします。さらに、`start` や `end` の時間を表わす要素はさらに細かく、`date`(日付)や `time`(時間)で表わすことにします。`date` と `time` の詳細な定義についてはリスト 4.1 を参照してください。

リスト 4.1 `schedule.dtd`

```
<!ELEMENT schedule (eventlist*)>
<!ELEMENT eventlist (event*)>
<!ATTLIST eventlist
  staff CDATA #REQUIRED
>
<!ELEMENT event (start,end?,project?,info,place)>
<!ATTLIST event
  type (meeting|work|study|lecture|party) "meeting"
>
<!ELEMENT start (date,time)>
<!ELEMENT end (date?,time)>
<!ELEMENT date (year,month,day,oftheweek?)>
<!ELEMENT time (hour,minute?)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT place (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT month (#PCDATA)>
<!ELEMENT day (#PCDATA)>
<!ELEMENT oftheweek (#PCDATA)>
<!ELEMENT hour (#PCDATA)>
<!ELEMENT minute (#PCDATA)>
<!ELEMENT project (#PCDATA)>
```

本章では、この DTD に準拠した XML 文書として、リスト 4.2 のようなファイル

schedule1.xml を取り扱うことにします。

リスト 4.2 schedule1.xml

```
<?xml version="1.0" encoding="EUC-JP"?>
<!DOCTYPE schedule SYSTEM "schedule.dtd" >
<schedule>
  <eventlist staff="青木保一">
    <event type="work">
      <start>
        <date><year>1999</year><month>10</month><day>18</day><oftheweek>月</oftheweek></date>
        <time><hour>10</hour><minute>00</minute></time>
      </start>
      <end>
        <time><hour>12</hour><minute>00</minute></time>
      </end>
      <info>入社</info>
      <place>Duo(早稲田)</place>
    </event>
    <event type="meeting">
      <start>
        <date><year>1999</year><month>10</month><day>18</day><oftheweek>月</oftheweek></date>
        <time><hour>13</hour><minute>00</minute></time>
      </start>
      <end>
        <date><year>1999</year><month>10</month><day>18</day><oftheweek>月</oftheweek></date>
        <time><hour>18</hour><minute>00</minute></time>
      </end>
      <project>XML</project>
      <info>打合せ</info>
      <place>CQ 出版(巣鴨)</place>
    </event>
  </eventlist>
```

(中略)

```
</eventlist>
<eventlist staff="飯塚富雄">
  <event type="work">
    <start>
      <date><year>1999</year><month>10</month><day>18</day><oftheweek>月</oftheweek></date>
      <time><hour>10</hour><minute>00</minute></time>
    </start>
    <end>
      <time><hour>12</hour><minute>00</minute></time>
    </end>
    <info>入社</info>
    <place>Duo(早稲田)</place>
  </event>
  <event type="meeting">
    <start>
      <date><year>1999</year><month>10</month><day>18</day><oftheweek>月</oftheweek></date>
      <time><hour>13</hour><minute>00</minute></time>
    </start>
    <end>
      <date><year>1999</year><month>10</month><day>18</day><oftheweek>月</oftheweek></date>
      <time><hour>18</hour><minute>00</minute></time>
    </end>
    <project>XML</project>
    <info>打合せ</info>
```

```
<place>CQ 出版(楽鴨)</place>
</event>

(中略)

</eventlist>

(中略)

</schedule>
```

(2) スケジュール一覧用スタイルシート

スケジュール管理用の XML 文書を HTML 形式に変換して出力するためのスタイルシートを、リスト 4.3 の schedule1.xsl に示します。このスタイルシートでは、元の XML 文書に記録されているスケジュール情報をもとに、event 要素を開始日順に整列するために start 要素を昇順に整列して、一覧形式でスケジュール一覧を作成するための変換規則を記述しています。

リスト 4.3 schedule1.xsl

```
<?xml version="1.0" encoding="EUC-JP"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <HTML>
    <HEAD>
      <TITLE>Title</TITLE>
    </HEAD>
    <BODY>
      <TABLE BORDER="1">
        <TR>
          <TD>日付</TD>
          <TD>名前</TD>
          <TD>開始</TD>
          <TD>終了</TD>
          <TD>内容</TD>
          <TD>場所</TD>
          <TD>プロジェクト</TD>
        </TR>
        <xsl:for-each select="/schedule/eventlist/event">
          <xsl:sort select="start"/>
          <TR>
            <TD>
              <xsl:apply-templates select="start/date"/>
            </TD>
            <TD>
              <xsl:value-of select="../@staff"/>
            </TD>
            <TD>
              <xsl:apply-templates select="start/time"/>
            </TD>
            <TD>
              <xsl:apply-templates select="end/date"/>
              <xsl:apply-templates select="end/time"/>
            </TD>
            <TD>
              <xsl:apply-templates select="info"/>
            </TD>
            <TD>
              <xsl:apply-templates select="place"/>
            </TD>
          </TR>
        </xsl:for-each>
      </TABLE>
    </BODY>
  </HTML>
</template>
```

```

        </TD>
        <TD>
            <xsl:apply-templates select="project" />
        </TD>
    </TR>
</xsl:for-each>
</TABLE>
</BODY>
</HTML>
</xsl:template>

<xsl:template match="*">
    <xsl:apply-templates />
</xsl:template>

<xsl:template match="text(">
    <xsl:value-of select="."/ />
</xsl:template>

<xsl:template match="text()|@"><xsl:value-of select="."/ /></xsl:template>

<xsl:template match="date">
    <xsl:value-of select="month" /><xsl:value-of select="day" />
    <xsl:if test="oftheweek"><xsl:value-of select="oftheweek" /></xsl:if>
</xsl:template>

<xsl:template match="time">
    <xsl:value-of select="hour" />:<xsl:value-of select="minute" />
</xsl:template>

</xsl:stylesheet>

```

4.3 XSLT プロセッサによる文書変換

本節では、リスト 4.2 の schedule1.xml とリスト 4.3 の schedule1.xsl を使用して、スタイルシートを使って HTML 文書へ変換するプログラムを作成します。iPEX で XSLT を処理するには、XSLT プロセッサとして XSLTReader を使用します。

(1) iPEX クラスのインクルード

iPEX の XSLT 関連のクラスを使用するためには、以下のようにヘッダファイル ipexslt.h をインクルードする必要があります。ここで定義されているクラスなどは、すべて iPEX ネームスペースの中で定義されているので、これらのクラスを使用するためには、iPEX::を明示的に使うか、using 宣言をする必要があります。ここでは std ネームスペースと共に using 宣言を使うことにします。

```

#include <iostream>
#include <memory>
using namespace std;

#include <ipexslt.h>
using namespace iPEX;

```

(2) 入力ストリームの作成

XSLSample1.cpp という名前のファイルでプログラムを作成します。プログラムで処理すべき XML 文書とスタイルシートは、コマンドライン引数で指定することにします。そのため、引数の数が 2 でない場合には、標準エラー出力へメッセージを出力してプログラムを終了させることにします。

InputStreamByFile クラスにより、XML 文書とスタイルシートの入力ストリーム isXml と isXslt を作成します。なお、この際に前章までの例と同様にコード変換のために WSTR、MBSTR マクロを使用します。

```

. . .
#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBString(x,L"euc-jp")
. . .
int main(int argc, char* argv[])
{
    EUCJPCConverterFactory fEUCJP;
    . . .
    DOMString xmlfilename = WSTR(argv[1]);
    InputStreamByFile isXml(xmlfilename.c_str());
    if (!isXml) {
        cerr << "Cannot open file: " << MBSTR(xmlfilename) << endl;
        return 1;
    }
    DOMString xslfilename = WSTR(argv[2]);
    InputStreamByFile isXslt(xslfilename.c_str());
    if (!isXslt) {
        cerr << "Cannot open file: " << MBSTR(xslfilename) << endl;
        return 1;
    }
}
. . .

```

(3) 入力ファイルのパーズ

XSLTReader に入力ファイルをパーズさせるために、まず XSLTReader のインスタンス reader を、入力ストリームを引数として渡し作成します。

```

XSLTReader reader(&isXslt, &isXml);
if (!reader) {
    cerr << "Error while reading xslt: " << MBSTR(xslfilename) << endl;
    return 1;
}

```

(4) 出力用 DOM ツリーの作成

IPEXDocument::createDocumentObject により出力用の Document オブジェクト pDocOut を作成し、createDocumentFragment()により DocumentFragment オブジェクト pdf を作成します。次に、これらを引数として XSLTReader のメンバ関数 read()を呼んで、出力用の DOM ツリーを作成します。

```

auto_ptr<Document> pDocOut(IPEXDocument::createDocumentObject());
auto_ptr<DocumentFragment> pdf(pDocOut->createDocumentFragment());
if (!reader.read(pDocOut.get(), pdf.get())) {
    cerr << "Error occured while processing xslt" << endl;
    return 1;
}

```

(5) 結果の出力

OutputStreamByFile によって標準出力への出力ストリーム ofs を作成します。次に、reader のメンバ関数 getWriter()によって、変換後の DOM ツリーを出力できる Writer のインスタンス pWriter を作成します。最後は、setStream()によって pWriter の出力ストリームを ofs に指定し、write()によって出力を実行します。

```

OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
if (!ofs) {

```

```

    cerr << "Cannot open stdout" << endl;
    return 1;
}
auto_ptr<Writer> pWriter(reader.getWriter(pdf.get()));
if (!pWriter.get()) {
    cerr << "Cannot create writer" << endl;
    return 1;
}
if (!pWriter->setStream(&ofs)) {
    cerr << "Cannot bind stream" << endl;
    return 1;
}
if (!pWriter->write(pdf.get())) {
    cerr << "Cannot output" << endl;
    return 1;
}
}

```

(6) プログラムのコンパイル

ここまでで説明したコードの断片を一つのプログラムにまとめたものをリスト 4.4 に掲載します。

リスト 4.4 XMLSample1.cpp

```

#include <iostream>
#include <memory>
using namespace std;

#include <ipexslt.h>
using namespace iPEX;

#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBString(x,L"euc-jp")

int main(int argc, char* argv[])
{
    EUCJPCConverterFactory fEUCJP;

    if (argc != 3) {
        cerr << "Usage: XSLSample1 xmlfile xslfile" << endl;
        return 1;
    }
    // 入力の XML 文書をオープン
    DOMString xmlfilename = WSTR(argv[1]);
    InputSteamByFile isXml(xmlfilename.c_str());
    if (!isXml) {
        cerr << "Cannot open file: " << MBSTR(xmlfilename) << endl;
        return 1;
    }
    // 入力のスタイルシートをオープン
    DOMString xslfilename = WSTR(argv[2]);
    InputSteamByFile isXslt(xslfilename.c_str());
    if (!isXslt) {
        cerr << "Cannot open file: " << MBSTR(xslfilename) << endl;
        return 1;
    }
    // XSLT プロセッサを生成
    XSLTReader reader(&isXslt, &isXml);
    if (!reader) {

```

```

    cerr << "Error while reading xslt: " << MBSTR(xslfilename) << endl;
    return 1;
}
// XSLT プロセッサによる変換
auto_ptr<Document> pDocOut(IPEXDocument::createDocumentObject());
auto_ptr<DocumentFragment> pdf(pDocOut->createDocumentFragment());
if (!reader.read(pDocOut.get(), pdf.get())) {
    cerr << "Error occured while processing xslt" << endl;
    return 1;
}
// 出力文書をオープン
OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
if (!ofs) {
    cerr << "Cannot open stdout" << endl;
    return 1;
}
// XML 文書を出力
auto_ptr<Writer> pWriter(reader.getWriter(pdf.get()));
if (!pWriter.get()) {
    cerr << "Cannot create writer" << endl;
    return 1;
}
if (!pWriter->setStream(&ofs)) {
    cerr << "Cannot bind stream" << endl;
    return 1;
}
if (!pWriter->write(pdf.get())) {
    cerr << "Cannot output" << endl;
    return 1;
}
return 0;
}
}

```

この例のプログラムをコンパイルするために、次のような makefile を作成します。

```

MAKEFILE = makefile
SHELL    = /bin/sh
CCC      = g++

CCFLAGS  = -O
DEFINES  = -DIPEX_LINUX -DNDEBUG
LDFLAGS  =

TARGET   = XSLSample1
IPEXDIR  = /usr/local/iPEX2_DE
IPEXLIB  = -lipex
OBJJS    = XSLSample1.o

all:
    (MAKE) -f (MAKEFILE) target

target: (TARGET)

clean:
    -rm -f *.o core

rebuild: clean all

%.o : %.cpp
    (CCC) (CCFLAGS) (DEFINES) ¥

```

```
-I(IPEXDIR)/include ¥
-c -o @ <

(TARGET): (OBJS)
(CCC) (LDFLAGS) -o @ ^ ¥
-L(IPEXDIR)/lib ¥
(IPEXLIB) -lstdc++
```

XSLSample1.cpp と makefile を同じディレクトリに置いて、"make"とタイプすれば自動的にコンパイル・リンクが実行され、実行ファイル XSLSample1 が作成されます。

(7) プログラムの実行

リスト 4.1 に示した DTD のファイル(schedule.dtd)、リスト 4.2 に示した XML 文書のファイル(schedule1.xml)、リスト 4.3 に示したスタイルシート(schedule1.xsl)、 および XSLSample1 が同じディレクトリにある場合、次のようにプログラムを実行します。

```
./XSLSample1 schedule1.xml schedule1.xsl
```

実際に出力される結果は、リスト 4.5 に示したようになります。出力されたテキストをファイルに保存して任意の HTML ブラウザで閲覧すれば、スケジュール表が表示されます。

リスト 4.5 XSLSample1 の実行結果

```
<HTML>
<HEAD>
  <TITLE>Title</TITLE>
</HEAD>
<BODY>
  <TABLE BORDER="1">
    <TR>
      <TD>日付</TD>
      <TD>名前</TD>
      <TD>開始</TD>
      <TD>終了</TD>
      <TD>内容</TD>
      <TD>場所</TD>
      <TD>プロジェクト</TD>
    </TR>
    <TR>
      <TD>10/18(月)</TD>
      <TD>笠原洋子</TD>
      <TD>09:00</TD>
      <TD>12:00</TD>
      <TD>打合せ</TD>
      <TD>Duo(早稲田)</TD>
      <TD>DB</TD>
    </TR>
    <TR>
      <TD>10/18(月)</TD>
      <TD>青木保一</TD>
      <TD>10:00</TD>
      <TD>12:00</TD>
      <TD>入社</TD>
      <TD>Duo(早稲田)</TD>
      <TD></TD>
    </TR>
```

(中略)

```
</TABLE>
</BODY>
</HTML>
```

4.4 xsl:param を使ってデータを抽出する

XSL プロセッサを利用してできる簡単なデータ抽出例として、xsl:param を使ったデータの抽出を紹介します。

データの抽出は、テーブルやツリーなどの一覧として表示されたデータの中から特定のデータを選択するといった単純な利用がほとんどです。この場合には、XSL スタイルシートにパラメータとして抽出するデータのキーを渡して、XSL プロセッサにデータを抽出させることができます。

スケジュール管理の例で、特定の年月日のスケジュールのみを抽出させる場合、指定された日付に一致する場合にだけデータをテンプレートの対象とるようにします(リスト 4.6)。

リスト 4.6 schedule2.xsl

```
<?xml version="1.0" encoding="EUC-JP"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:param name="year"/>
  <xsl:param name="month" select="''"/>
  <xsl:param name="days"/></xsl:param>

  <xsl:template match="/">
    <HTML>
      <HEAD><TITLE>Title</TITLE></HEAD>
      <BODY>
        <TABLE BORDER="1">
          <TR>
            <TD>日付</TD><TD>名前</TD><TD>開始</TD><TD>終了</TD>
            <TD>内容</TD><TD>場所</TD><TD>プロジェクト</TD>
          </TR>
          <xsl:for-each select="/schedule/eventlist/event">
            <xsl:sort select="./start/date"/>
            <xsl:sort select="../@staff"/>
            <xsl:sort select="./start/time"/>
            <xsl:if test="start/date/year = year
              and start/date/month = month
              and start/date/day = days">
              <xsl:apply-templates select="."/>
            </xsl:if>
          </xsl:for-each>
        </TABLE>
      </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="event">
    <TR>
      <TD><xsl:apply-templates select="start/date"/></TD>
      <TD><xsl:value-of select="../@staff"/></TD>
      <TD><xsl:apply-templates select="start/time"/></TD>
      <TD><xsl:apply-templates select="end/date"/>
```

```

        <xsl:apply-templates select="end/time"/></TD>
    <TD><xsl:apply-templates select="info"/></TD>
    <TD><xsl:apply-templates select="place"/></TD>
    <TD><xsl:apply-templates select="project"/></TD>
</TR>
</xsl:template>

<xsl:template match="*">
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()">
    <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="date">
    <xsl:value-of select="month"/><xsl:value-of select="day"/>
    <xsl:if test="oftheweek">
        (<xsl:value-of select="oftheweek"/>)
    </xsl:if>
</xsl:template>

<xsl:template match="time">
    <xsl:value-of select="hour"/>:<xsl:value-of select="minute"/>
</xsl:template>

</xsl:stylesheet>

```

(1) パラメータの定義と条件分岐

XSLスタイルシートのコードを変更することなくXSLスタイルシートへパラメータを引き渡すためには、xsl:param タグを使用します。

まず、スタイルシート外部からのパラメータを受け取れるようにするために、次の形式で xsl:param を定義します。

```

<xsl:param name="パラメータ名" select="既定値"/>
または
<xsl:param name="パラメータ名">既定値</xsl:param>

```

既定値を設定する場合、select 属性を指定します。既定値を設定しない場合、select 属性には空文字列(' ')が設定されたこととなります。ここでは、年、月、日に対応したパラメータとして、year、month、days というパラメータを定義しており、いずれも既定値は空文字列となっています。

次に、特定の条件が成立しているかどうかによってスタイルシートを適用させる場合には、xsl:if タグを利用します。event 要素ごとに、start/date 要素の year、month、day 要素がパラメータと一致した場合にだけテンプレートが適用されるように修正します。

実際の条件式は xsl:if タグの test 属性に記述しています。条件式などでパラメータを参照する場合には、ダラー記号(\$)をパラメータの前に付加することになっています。文書要素のテキストとパラメータの値を比較する際には、ダラー記号のついたパラメータ名全体を単一引用符(')でくくります。ここでは、日付を比較するために年月日それぞれについて文書要素のテキストとパラメータの値を比較し、各条件式を and 演算子で接続していますが、xsl:if を入れ子にした要素とすることもできます。

(2) プログラムの構成

特定のデータの抽出を XSL スタイルシートで実現できるようになるので、プログラムが

ら特定の年月日を指定して、それに合致するデータだけを抽出するようにします。

ここでのプログラム XSLSample2.cpp は、前節の XSLSample1.cpp とほとんど同じですが、上記のように年月日のパラメータを指定する部分だけ異なります。

XSLTReader がスタイルシートにパラメータを渡すようにするためには、setParameter() メンバ関数を使います。setParameter() の引数型は、std::map<std::pair<DOMString, DOMString>, Value*>であり、最初の DOMString のペアはネームスペース URI とローカルネーム、最後の Value* はパラメータの値にそれぞれなります。

今回は、スタイルシートの xsl:param として宣言された year、month、days にそれぞれ"1999"、"10"、"18"という値を渡し、1999年10月18日のデータを抽出することにします。

そのためにまず、setParameter()の引数のインスタンス mapParameter を作成します。次に、std::map クラスの insert()メンバ関数により、パラメータ名と値を代入します。なお、ネームスペース URI は空文字列とし、既定のネームスペースを使うよう指示します。最後に、mapParameter を引数として XSLTReader の setParameter()関数を呼び、上で代入したパラメータ名と値をスタイルシートに渡します。

```
std::map<std::pair<DOMString, DOMString>, Value*> mapParameter;
mapParameter.insert(make_pair(make_pair(L"", L"year"),
                               new ValueString(L"1999")));
mapParameter.insert(make_pair(make_pair(L"", L"month"),
                               new ValueString(L"10")));
mapParameter.insert(make_pair(make_pair(L"", L"days"),
                               new ValueString(L"18")));
reader.setParameter(&mapParameter);
```

(3) プログラムのコンパイル

ここまでに説明したコードの断片を一つのプログラムにまとめたものをリスト 4.7 に掲載します。

リスト 4.7 XSLSample2.cpp

```
#include <iostream>
#include <memory>
using namespace std;

#include <ipexslt.h>
using namespace iPEX;

#define WSTR(x) iPEX::Character::toWCString(x,L"euc-jp")
#define MBSTR(x) iPEX::Character::toMBString(x,L"euc-jp")

int main(int argc, char* argv[])
{
    EUCJPCConverterFactory fEUCJP;

    if (argc != 3) {
        cerr << "Usage: XSLSample2 xmlfile xslfile" << endl;
        return 1;
    }

    DOMString xmlfilename = WSTR(argv[1]);
    InputSteamByFile isXml(xmlfilename.c_str());
    if (!isXml) {
        cerr << "Cannot open file: " << MBSTR(xmlfilename) << endl;
    }
}
```

```

    return 1;
}
DOMString xslfilename =WSTR(argv[2]);
InputStreamByFile isXslt(xslfilename.c_str());
if (!isXslt) {
    cerr << "Cannot open file: " << MBSTR(xslfilename) << endl;
    return 1;
}
XSLTReader reader(&isXslt, &isXml);
if (!reader) {
    cerr << "Error while reading xslt: " << MBSTR(xslfilename) << endl;
    return 1;
}

std::map<std::pair<DOMString, DOMString>, Value*> mapParameter;
mapParameter.insert(make_pair(make_pair(L"", L"year"),
    new ValueString(L"1999")));
mapParameter.insert(make_pair(make_pair(L"", L"month"),
    new ValueString(L"10")));
mapParameter.insert(make_pair(make_pair(L"", L"days"),
    new ValueString(L"18")));
reader.setParameter(&mapParameter);

auto_ptr<Document> pDocOut(IPEXDocument::createDocumentObject());
auto_ptr<DocumentFragment> pdf(pDocOut->createDocumentFragment());
if (!reader.read(pDocOut.get(), pdf.get())) {
    cerr << "Error occured while processing xslt" << endl;
    return 1;
}
OutputStreamByFile ofs(OutputStreamByFile::STDOUT, L"euc-jp");
if (!ofs) {
    cerr << "Cannot open stdout" << endl;
    return 1;
}
auto_ptr<Writer> pWriter(reader.getWriter(pdf.get()));
if (!pWriter.get()) {
    cerr << "Cannot create writer" << endl;
    return 1;
}
if (!pWriter->setStream(&ofs)) {
    cerr << "Cannot bind stream" << endl;
    return 1;
}
if (!pWriter->write(pdf.get())) {
    cerr << "Cannot output" << endl;
    return 1;
}
return 0;
}
}

```

この例のプログラムをコンパイルするために、次のような makefile を作成します。

```

MAKEFILE = makefile
SHELL   = /bin/sh
CCC     = g++

CCFLAGS = -O
DEFINES = -DIPEX_LINUX -DNDEBUG
LD_FLAGS =

```

```

TARGET    = XSLSample2
IPEXDIR   = /usr/local/iPEX2_DE
IPEXLIB   = -lipex
OBJJS     = XSLSample2.o

all:
    (MAKE) -f (MAKEFILE) target

target: (TARGET)

clean:
    -rm -f *.o core

rebuild: clean all

%.o : %.cpp
    (CCC) (CCFLAGS) (DEFINES) ¥
    -I(IPEXDIR)/include ¥
    -c -o @ <

(TARGET): (OBJJS)
    (CCC) (LDFLAGS) -o @ ^ ¥
    -L(IPEXDIR)/lib ¥
    (IPEXLIB) -lstdc++

```

XSLSample2.cpp と makefile を同じディレクトリに置いて、"make"とタイプすれば自動的にコンパイル・リンクが実行され、実行ファイル XSLSample2 が作成されます。

(4) プログラムの実行

リスト 4.1 に示した DTD のファイル(schedule.dtd)、リスト 4.2 に示した XML 文書のファイル(schedule1.xml)、リスト 4.6 に示したスタイルシート(schedule2.xsl)、および XSLSample2 が同じディレクトリにある場合、次のようにプログラムを実行します。

```
./XSLSample2 schedule1.xml schedule2.xsl
```

実際に出力される結果は、リスト 4.8 に示したようになります。1999 年 10 月 18 日のデータだけが抽出されます。出力されたデータをファイルに保存してを任意の HTML ブラウザで閲覧すれば、1999 年 10 月 18 日のデータだけが表示されます。

リスト 4.8 XSLSample2 の実行結果

```

<HTML>
  <HEAD>
    <TITLE>Title</TITLE>
  </HEAD>
  <BODY>
    <TABLE BORDER="1">
      <TR>
        <TD>日付</TD>
        <TD>名前</TD>
        <TD>開始</TD>
        <TD>終了</TD>
        <TD>内容</TD>
        <TD>場所</TD>
        <TD>プロジェクト</TD>
      </TR>
      <TR>

```

```
<TD>10/18(月)</TD>
<TD>笠原洋子</TD>
<TD>09:00</TD>
<TD>12:00</TD>
<TD>打合せ</TD>
<TD>Duo(早稲田)</TD>
<TD>DB</TD>
</TR>
```

(中略)

```
<TR>
<TD>10/18(月)</TD>
<TD>飯塚富雄</TD>
<TD>13:00</TD>
<TD>10/18(月) 18:00</TD>
<TD>打合せ</TD>
<TD>CQ 出版(巣鴨)</TD>
<TD>XML</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```