テキスト処理 最終課題 2009/12/15 出題 - 2010/1/17 提出

高井 実

最終課題内容

対話形式の XML パーザを実装せよ。

＊ プログラムを実行するとコマンドプロンプトが表示される。

例: $perl xml_parse.pl

> _

＊ 以下のコマンドを実装する。

1. parse〈file〉: 指定されたファイルをパーズし、抽象構文木として内部に記憶。
2. print : 記憶している抽象構文を XML の形で出力。
3. get コマンド群 : 指定された要素を得る。
4. set コマンド群 : 要素の内容を置き換える。
5. add コマンド群 : 新たな要素を追加する。

1、2 のコマンドは必須、3 以降のコマンドはひとつ以上実装できて 50 点とする。残りのコマンドはチャレンジ問題とし、できる限りの工夫をすること。オリジナリティのある工夫をする方が点数は高い。(もちろん、1 や 2 のコマンドで改良の意味でオリジナリティを出すことも可。)

＊ パーザが扱う XML は、

＊ XML 宣言がない場合も許す。宣言がなされた場合は、バージョン情報がなければならない。

＊ DTD (スキーマ) は考えない。(妥当性のチェックは行わない。)

＊ 開始タグにはタグ名以外に属性のリストを許す。

例: 〈text id="1" type="greeting"〉

Hello, World!

〈/text〉

＊ コメントを許す。コメントの形式は 〈!-- --〉 。

＊ コマンドやその機能、パーザの仕様を自分で考えて実装し、その内容をレポートにまとめること。

レポート形式について

＊ 最初に自分自身の定義による XML の抽象構文の説明。わかりやすく図説し、それをどのように Perl で表現したかの解説。

＊ パーザの仕組みを解説すること。全体の関数構造、関数ひとつひとつの役割、またコード中の正規表現の意味やループ構造があればそれの継続・終了条件など。

＊ プログラムを独立した pl ファイルで提出すること。ファイル名は「xml_neXXXXXX.pl」(XXXXXX は自分の学籍番号) とすること。

目次 (6 章構成)

1. やりたいことは何か

(1)コマンドプロンプトを表示し、(2)parse コマンドで XML を解析（パーズ）し、(3)print コマンドやその他のコマンドを受け付けてコマンドに対応した操作を行うプログラムを作る。

1-1. (1)コマンドプロンプトを表示する

コマンドプロンプトとは、システムが命令入力を受け付けられる状態にあることを示すために表示される記号のことをいう。つまり、ユーザからコマンドを受け付ける形のプログラムを作ればよい。コマンドを 1 回受け付けるだけでプログラムが終了してしまっては様々なコマンドによる操作ができないので、コマンドは何度も受け付け、特定のコマンドが入力されたらプログラムを終了する形にする。

コマンド入力の受け付け(Read)、その入力されたコマンドの評価・実行(Evaluation)、その実行結果を出力(print)する一連の流れを何度も行うループ(Loop)の構造を Read-Eval-Print-Loop(REPL)という。この REPL に近いものを実装すれば「対話形式」のプログラムを作ることができる。

1-2. (2) parse コマンドで XML を解析する

ファイルに書かれた XML をプログラム上で操作（例えば person 要素が何個あるか調べるなど）するには、プログラム上で XML の情報を管理しなければならない。ファイルに書かれた XML というのは、テキストファイルであり、そのテキストによって XML のデータが表現されている。

```
<people>
    <person>
        <name>Tanaka</name>
    </person>
    <person>
        <name>Yamada</name>
    </person>
</people>
```

これは一種の XML であるが、人が見れば people の中に Tanaka という name を持った person と、Yamada という name を持った person があるという情報を読み取ることができる。これは XML というものが（コンピュータだけでなく）人にも読めるように設計されているからであるが、逆にこの情報をコンピュータに読ませるには設計に基づく特定のルールに従って XML を解析しなければならない。

XML がどういうルールで記述される言語（テキスト）であるかについて理解し、そのルールを解析ルールとして取り込む必要があるが、今回の課題では実際の XML の仕様（ここでは XML 言語のルール）を厳密に取り込むことは求められていない。

<WORD> という形式のものを「要素開始タグ」と定義し、</WORD> という形式のものを「要素終了タグ」と定義する。WORD の部分は任意のアルファベットからなり、その文字列自体が要素名となる。要素名の同じ開始タグと終了タグによって囲まれた中身を「要素内容」と呼び、要素内容が要素名に関する情報を表すことにする。その要素名の要素開始タグ、要素内容、要素終了タグを併せて「要素」と呼ぶ。

このように定義すると前述の XML を人が読み取れたようにコンピュータ上で扱うことができる。実際にコンピュータプログラム上でこのデータを表現するための方法として今回は「抽象構文」を用いる。

抽象構文とは意味のある情報のみを保持する仮想的な構文であり、前述の話に関連して述べると、ある要素の要素名は何である、その要素の要素内容は何である、という情報を持つことができる。これに対して元となる XML ファイルは XML 言語（テキスト）が具体的に書かれており、解析しなければある要素の内容を調べるといったことができない。具体的に書かれたデータの表現を具象構文という。

抽象構文を用いずにプログラム上で XML を操作することもできるが、その場合には一つひとつの操作ごとに XML を解析しなければならない。データを構造的な意味も含めて扱いたい場合は、一度解析を行い抽象構文としてデータを保持すると都合がよく、今回の課題でも XML が持つデータ構造に合った抽象構文木としてデータを扱うことが求められている。

1-3. (3)その他のコマンドを受け付けて対応する操作を行う

解析後に作られた抽象構文木に対して実装するコマンド操作を行えばよい。print コマンドを始めとする各種コマンドをどう設計するか、XML パーザプログラムとして何をできるようにするかを考えなければならない。

2. XML の抽象構文

2-1. 抽象構文に必要な情報

XML のデータとして、XML 宣言・DTD 宣言・処理命令・コメント・ルート要素などがあるが、操作する上で必要なデータは「ルート要素」だけである。ルート要素は次の情報を持っている。

name ： 1 つの要素名（文字列）
attr ： 0 個以上の属性（属性名をキーに、属性値を値に持つハッシュのリファレンス）
cont ： 要素内容（要素内容を持つ配列のリファレンス）

ここで、要素内容には「子要素」か「連続する PCDATA」が 0 個以上登場するが、これを 1 つの要素内容として扱うことにする。つまりルート要素は「0 個以上の要素内容」を持つということになる。ある要素内容が要素であればルート要素と同様に上記の情報を持っている。要素内容が PCDATA であった場合は次の情報を持っている。

cont ： PCDATA そのもの（文字列）

ここで、ルート要素・要素である要素内容・PCDATA である要素内容を総称して、(抽象構文木の 1 ノードとしての意味で)「ノード」と呼ぶことにする。

以上の情報さえ持っていれば最低限の操作を行うことができるが、他のコマンド操作を簡単にするため全ノードに次の情報も持たせることにした。

parent ： 親ノードへのリファレンス
type ： 要素か PCDATA かを示すタイプ（文字列 'ELEMENT' または 'PCDATA'）
id ： 抽象構文木におけるノードの ID（非負整数、ルート要素は id=0）
idx ： 親ノードの何番目の子ノードかを示すインデクス（非負整数、最初の子は idx=0）
depth ： 要素タグによるネストの深さ（非負整数、ルート要素は depth=0）

2-2. 抽象構文木におけるノード

```
<person id="1" lang="en">
    <text>My name<br/> is <name>Tanaka</name></text>
</person>
```

これが解析対象の XML であるとすると、次の 7 つのものがノードとして構成される。

person 要素	text 要素	(#PCDATA): "My name"	br 要素
(#PCDATA): "is"	name 要素	(#PCDATA): "Tanaka"	

2-3. Perl で表現するノードのデータ構造

前述の XML を例に、実装したデータ構造による表現を次に示す。

```
$ast = {
  parent => undef,
  type => 'ELEMENT',
  name => 'person',
  attr => {
    id => '1',
    lang => 'en',
  },
  cont => [#A],
  id => 0,
  idx => 0,
  depth => 0,
}
```

```
#A = {
  parent => $ast,
  type => 'ELEMENT',
  name => 'text',
  attr => {},
  cont =>
    [#B,#C,#D,#E],
  id => 1,
  idx => 0,
  depth => 1,
}
```

```
#B = {
  parent => #A,
  type => 'PCDATA',
  cont => 'My name',
  id => 2,
  idx => 0,
  depth => 2,
}
```

```
#C = {
  parent => #A,
  type => 'ELEMENT',
  name => 'br',
  attr => {},
  cont => [],
  id => 3,
  idx => 1,
  depth => 2,
}
```

```
#D = {
  parent => #A,
  type => 'PCDATA',
  cont => 'is',
  id => 4,
  idx => 2,
  depth => 2,
}
```

```
#E = {
  parent => #A,
  type => 'ELEMENT',
  name => 'name',
  attr => {},
  cont => [#F],
  id => 5,
  idx => 3,
  depth => 2,
}
```

```
#F = {
  parent => #E,
  type => 'PCDATA',
  cont => 'Tanaka',
  id => 6,
  idx => 0,
  depth => 3,
}
```

※3 章 4 節参照

```
$ast_info->{id} = [
  $ast,
  #A,
  #B,
  #C,
  #D,
  #E,
  #F,
]
```

#X のような記述は説明のために便宜的に名付けたそのノードの情報を持つハッシュリファレントを示す擬似コードである。実際に A や B などと名付けてリファレンスにはしていない(“無名”ハッシュを用いている)。

1 つのノードをハッシュとして情報を持たせ、自身のリファレントを、親要素が持つ(cont キーの無名)配列の一要素として持たせている。これによって木構造を実現している。

3. パーザの仕組み

3-1. 定義した XML 文書の構成

私が定義した XML を擬似 Perl コードと擬似 BNF 記法で以下に示す。

$re_name	$re_name_1char = qr/[A-Za-z]|[_:]/; $re_name_2chars = qr/${re_name_1char}|[0-9]|[.-]/; $re_name = qr/${re_name_1char}${re_name_2chars}/;
S	(半角スペース ｜ タブ文字 ｜ 改行文字)+
XML 文書	(XML 宣言)? (コメント ｜ 処理命令 ｜ S)* (DTD 宣言 (コメント ｜ 処理命令 ｜ S)*)? (要素) (コメント ｜ 処理命令 ｜ S)*
要素	(空要素タグ) ｜ (開始タグ 要素内容 終了タグ)
要素内容	(要素 ｜ PCDATA)*
PCDATA	(コメント ｜ 処理命令 ｜ CDATA 区間 ｜ (']]>'を含まない文字列))*
CDATA 区間	'<![CDATA[' ('']]>'を含まない文字列) ']]>'
XML 宣言	'<?xml' (VersionInfo) (EncodingDecl) (SDDecl) S? '?>' ※VersionInfo, EncodingDecl, SDDecl の定義は仕様に完全準拠している
コメント	'<!--' ('--'を含まない文字列) '-->'
処理命令	('<?' PITarget '?>') ｜ ('<?' PITarget S ('?>'を含まない文字列) '?>'))
PITarget	(([Xx][Mm][Ll])を除く$re_name)
DTD 宣言	'<!DOCTYPE>' ｜ ('<!DOCTYPE' S (※任意の文字列) '>') ※任意の文字列は、ネストの対応が取れない '>' を除く ※つまり、開始区切り子の '<' とネストの深さが同じ '>' を終了区切り子とする
空要素タグ	'<' (要素名) (S 属性)* S? '/>'
開始タグ	'<' (要素名) (S 属性)* S? '>'
終了タグ	'</' (要素名) S? '>'
要素名	$re_name
Eq	S? '=' S?
属性	(属性名) Eq '"' (属性値) '"' ｜ (属性名) Eq "'" (属性値) "'"
属性名	$re_name
属性値	((属性値を括る引用符文字)と'<'を含まない文字列)

XML の仕様 http://www.w3.org/TR/REC-xml/ に概ね準拠したが、何箇所か独自の定義にしている。

* DTD 宣言の定義を大幅に変えている。ネストの対応した山括弧を終了区切り子としたのは、DTD 宣言中のマークアップ宣言に対応するための安易な発想だが、ENTITY 宣言中で文字列として山括弧閉じが現れていると、正しい XML でもエラーにしてしまう可能性がある。
* PITarget 名・要素名・属性名に使える文字を $re_name で定義した文字列に制限した。
* 文字参照を厳密にチェックできないため、単独の '&' の登場を許可している。

3-2. 解析の流れ

作成したプログラムでは私が定義した XML に基づいて解析を行っている。解析に用いている関数を次に示す。次の 13 種類の関数が parse コマンドにおける主たるルーチンである。

```
buffering_first()                [1]: ファイルから 1 行目を読み込む
buffering()                      [2]: ファイルから次の 1 行を読み込む
parse_main()                     [3]: XML 文書の解析
    xmldecl_reading()            [4]: XML 宣言の解析
      ※attribute_reading()      <A>: (versionInfo, EncodingDecl, SDDecl の解析)
    comment_reading()            [5]: コメントの解析
    pi_reading()                 [6]: 処理命令の解析
    dtd_reading()                [7]: DTD 宣言の解析
    element_reading()            [8]: 要素の解析 (開始タグの解析を含む)
        attribute_reading()      [9]: 属性の解析
        content_reading()       [10]: 要素内容の解析
          ※element_reading()    <B>: (再帰的に要素を解析)
            pcdata_reading()    [11]: PCDATA の解析
              ※comment_reading() <C>: (コメントの解析)
              ※pi_reading()     <D>: (処理命令の解析)
                cdata_reading() [12]: CDATA 区間の解析
        etag_reading()          [13]: 終了タグの解析
```

始めにバッファリングの特徴を示しておく。以下は buffering()の実装である。

```
 1 sub buffering {
 2   my $ref_buf = shift;
 3   my $line;
 4   while ($line = <FH>) {   # (1) readline()を用いて 1 行ずつ読み込みを行う
 5     next if $line !~ /¥S/; # (2) 入力した行が空白のみであれば読み込みしなおす
 6     wstrim($line);         # (3) 連続する空白類文字を 1 個の空白スペースに置換する
 7     ltrim($line);          # (4) 先頭に空白があれば取り除く
 8     $$ref_buf .= $line;    # (5) バッファに読み込んだ行を追加する
 9     return 1;              # (6) バッファリングが成功したこと示す値を返却する
10   }
11   return;                  # (7) ファイルからの読み込みができなくなれば偽を返却する
12 }
```

具象構文中の連続する空白類文字を 1 個の空白に置換している。この理由は上記のバッファリングにおいて 5 行目の(2)で示した条件による再読み込みを採用したため、連続する空白に重要な意味を持たせなかったからであるが、HTML における pre 要素内容のように空白類に意味を持たせたい場合には不適切な操作である。

XML 宣言が正しいかをチェックするため、1 行目の読み込みは上記に示した buffering_first()関数を用いている。buffering_first()では、readline(), wstrim()のみを行っている。

3-3. 詳細な解析の流れ

前述の関数全てについて説明することは割愛する。ここでは、要素の解析を例に、どのような手順で解析が進んでいるかを説明する。

```
    <div id="root"    class="hoge    fuga">
  <p><em>Ex</em>tensible<br />HyperText Markup Language</p>
</div>
```

1. $buf = ' ';

parse_main()では1行目を読み込み、$bufには上記の文字列が入る。末尾の空白は改行文字が空白に置換されたものである。buffering()でltrim()を呼んでいるのは、ファイルから次の行の読み込みを行い$bufに追加する時点で読み込んだ行の先頭に空白があると、前の行の末尾の空白によって空白が2連続してしまうことを防いでいるためである。wstrim()を$bufにかければよいが、$bufが非常に長くなる可能性を考えてwstrim()を読み込んだ行($line)にかけている。

$bufの先頭がXML宣言開始区切り子であるかどうかをチェックする。次に先頭の空白を除去する。そして$buf の先頭がコメント・処理命令・DTD 宣言の開始区切り子でなく '<' であればルート要素の解析に入る。この時点で$bufが空でなければ不正な文字列が登場したことになるため異常終了している。解析の全ての箇所に関して共通することだが、不正な文字列の登場などで構文エラーとならない限りは$bufが解析中の文字列とマッチしなくなった時点でバッファリングするようにしている。

2. $buf = 'div id="root" class="hoge fuga"> ';

ルート要素の解析 element_reading()に入ったところである。チェック時にマッチした文字列は置換演算時に削除しているため、ここでは開始タグ開始区切り子が削除されている。要素の解析では始めに要素名を取得しようとする。' ' か '/>' か '>' が直後にある先頭の$re_name が取得できればそれが要素名である。この正規表現は /^(${re_name})(?= |¥/>|>)/ である。

3. $buf = ' id="root" class="hoge fuga"> ';

次に属性を取得するためのループに入る。ループの中では属性を1組取得するattribute_reading()を呼んでいる。その関数は「属性名」・「属性値を括るクオートの種類」・「属性値」・「属性値に不正な文字が含まれていたか」・「属性の解析中にファイル入力が失敗したか」・「属性を取得できたか」を表す値を持つハッシュのリファレントを返却する。属性が取得できていれば、呼び出した際に解析中の要素の属性としてその情報を追加する。属性が取得できなかった時点でこのループを抜ける。

4. $buf = '> ';

属性をチェックし終えた後は開始タグ終了区切り子を探す。次の文字が、空白が直前にあってもよい '>' または空要素タグ終了区切り子 '/>' でなければ構文エラーとなる。開始タグ終了区切り子が見つかり、空要素タグでなかった場合には要素内容の解析に入る。

5. $buf = ' Extensible
HyperText Markup Language ';

要素内容の解析content_reading()では、終了タグ開始区切り子・開始タグ開始区切り子・PCDATAの順にチェックを行う。開始タグ開始区切り子であれば子要素の開始として、element_reading()を再帰的に呼んでいる。

6. $buf = 'Extensible
HyperText Markup Language ';
div 要素と同様に、その子要素の p 要素の要素内容のチェックまで行った状態である。ここで em 要素が登場したので同じく前述の(2.)から(5.)のチェックを行った状態まで進む。

7. $buf = 'Extensible
HyperText Markup Language ';
PCDATA が現れた。content_reading()では、PCDATA が現れたと判断すると pcdata_reading()を呼ぶ。pcdata_reading()では、要素名や属性と同様に抽象構文木に保存すべき重要な値 PCDATA を取得しようとする。正規表現 /^([^<]*)/ で、コメント・処理命令・CDATA 区間・開始タグ・終了タグの開始区切り子として有り得る '<' よりも前の文字列を全て取得する。取得した文字列の中に PCDATA で許されない ']]>' という文字列が含まれていたら異常終了する。

' Hello ' という文字列を PCDATA として取得すると、先頭と末尾にある空白を削除するようにしている。これは、全ての要素を HTML におけるブロックレベル要素と見做したためであるが、例示している 'Extensible' の部分が 'Ex' と 'tensible' に単語として区切られてしまうことになる。これに対応することも可能だが、今回は要素内容と PCDATA は非連結であるものとした。

8. $buf = 'tensible
HyperText Markup Language ';
PCDATA の解析が終わると、content_reading()で pcdata_reading()を呼んだ位置に戻ってくる。ここで終了タグ開始区切り子が見つかると etag_reading()が呼ばれる。etag_reading()では終了タグの形式を解析して構文エラーがない限り読み飛ばす。ここでは開始タグ解析時に保存していた要素名と照合して、開始タグと終了タグの要素名が一致しているかをチェックしている。

9. $buf = 'tensible
HyperText Markup Language ';
終了タグを解析し終えると、etag_reading()から content_reading()、そして element_reading()まで戻り、element_reading()を呼び出した content_reading()に戻ってくる。em 要素の解析が終わり、p 要素の解析に戻ってきたということである。

10. $buf = 'tensible
HyperText Markup Language ';
以降、再帰・繰り返しが続き、最終的にはルート要素の解析に戻ってくる。

11. $buf = ' ';
ルート要素の終了タグを見つけると parse_main()まで戻ってくる。ここからは parse_main()の中にあるバッファリングにより、ルート要素以降を解析しようとする。ファイルから読み込みができなくなれば parse_main()のループを抜ける。そこでルート要素が登場したかどうかを確認し、登場していたら解析を成功したものとし、parse コマンドのメインルーチン do_parse()で$ast を構築している。

3-4. ノードの関連情報の構築

抽象構文の設計は 2 章で述べた通りであるが、XML の解析中に$ast_info という導出テーブルに相当するデータを構築している。add_info()という関数を element_reading()と content_reading()の PCDATA 解析後に呼んでおり、その要素内容の登場した順に ID を、その要素内容のネストの深さに対して depth を要素内容（ノード）自身に持たせ、$ast_info->{id}[ID]にその要素のリファレントを保存している。

XML の解析が成功すると、その XML の抽象構文木と、ID 順にそのノードへのリファレントが格納された配列$ast_info->{id}が完成する。その具体的なデータ表現は 2 章 3 節の表を参照のこと。

4. Perl ソースコード

4-1. コーディング方針

抽象構文木を作る必要があるように、今回のプログラムでは構造体が重要になってくる。Perl において構造体を実現する方法として、無名配列・無名ハッシュの多次元化があるが、可読性を重視して、配列の方が適していない限りハッシュを積極的に利用した。プログラム冒頭の変数$ac は AcceptCommand (受理するコマンド) を意味するハッシュリファレンスであり、今回実装したコマンドのキー・実行する関数・概要などを格納している。

Perl5 で利用できる機構は積極的に用いているが、5.8.x でも動作するように 5.10.x に依存した構文は用いていない。なお、プログラムファイルの文字コードは BOM なしの UTF-8、改行コードは LF としている。プログラム中で利用しているモジュールは次のものである。

```
use strict;
use warnings;
use Data::Dumper;
use List::Util qw(first max maxstr min minstr reduce shuffle sum);
use Text::ParseWords;
```

Data::Dumper はデバッグ出力に、List::Util は汎用的なライブラリとして、Text::ParseWords はプログラム中で受け取ったコマンドを分割する際に利用している。

4-2. コマンドの分割

このプログラムは、コマンドプロンプトを表示してコマンドを受け取り対応する操作を行う対話型プログラムであるが、そのためにまずメインルーチンとして次のような REPL に相当するループを書いた。

```
1  my $program_quit;
2  for (;;) {
3    my $command = input();
4    my @opes = eval_cmd($command);
5    if (exec_ope(@opes)) {
6      last if $program_quit;
7    } else {
8      message('Invalid command!');
9    }
10 }
```

input()でコマンド入力を受け付け、eval_cmd()では入力されたコマンド文字列を分割している。そして exec_ope()で実際にコマンドを実行し出力している。プログラムの終了を命令するコマンドであれば REPL を抜けるが、今回はグローバル変数を用いて終了するかどうかを操作できるようにした。

コマンド文字列の分割に関して、今回は各種コマンドをシェルのようにオプションをつけて指示できるように設計した。そのため、空白を含む値を取得するためにもコマンドをうまく分割する必要がある。

> get -attr_value "Perl x XML"

このようなコマンドが入力されたら、属性値が "Perl x XML" (引用符を含まない) である要素を取得できるようにするといった -attr_value オプションを実装することができる。

このようなコマンド分割の実現のために、正規表現で実装することも考えたが、Text::ParseWords モジュールを利用することにした。parse_line()でもよいが、shellwords()の方が目的に適しているためこれを利用している。

> get -attr_value ″Perl x XML

このような文字列(クオートが開いたまま)がくると、Text::ParseWords の関数は解析に失敗する(空のリストを返却する)。この不具合を補完するため、開いたままのクオートがあれば末尾で対応するクオートを閉じるような正規表現を書いた。

```
 1  sub quote_validation {
 2    my $str = $_[0];
 3    my $tmp = $str;
 4    # 先頭から順に、開始クオートから終了クオートまでを削除していく
 5    $tmp =~ s/((?<!¥¥)[″’]).*?¥1//g;
 6    # 残った文字列にクオートがあれば、もともとの文字列の末尾にそのクオートを補完する
 7    $str .= $1 if $tmp =~ /([″’]).*$/;
 8    # 無効コンテキストの場合には破壊的な代入を行う
 9    $_[0] = $str if not defined wantarray;
10    return $str;
11  }
```

入力されたコマンド文字列は、chomp(), wstrim(), quote_validation()が適用された後で eval_cmd()の中で呼ばれる shellwords()によって分割される。

4-3. ソースコードの構成

今回作成したソースコードはコメントも含めて 1200 行近いが、その構成を部分ごとに簡単に示す。

行数	概要
1 - 22	冒頭, モジュール読み込み, シグナルの捕獲
23 - 89	受理するコマンドに関するグローバル変数 $ac
90 - 96	各種正規表現のグローバル変数 $re_xxxx
97 - 105	抽象構文木に関するグローバル変数と set_ast 関数
106 - 128	プログラムのメインルーチン：REPL
129 - 224	13 個の関数定義：ヘルパー関数, 汎用関数
225 - 310	7 個の関数定義：入力されたコマンド文字列を操作する関数群
311 - 362	5 個の関数定義：AST に対する情報を操作する関数群
363 - 550	9 個の関数定義：各種コマンドのメインルーチン &do_xxxx
551 - 919	13 個の関数定義：do_parse のサブルーチン群 (XML 解析ルーチン)
920 - 956	2 個の関数定義：do_print のサブルーチン群
957 - 1082	11 個の関数定義：do_get のサブルーチン群
1083 - 1116	2 個の関数定義：do_del のサブルーチン群
1117 - 1131	1 個の関数定義：do_ls のサブルーチン群
1132 - 1156	1 個の関数定義：do_cd のサブルーチン群

4-4. 関数一覧

定義した関数の一覧を示しておく。計 65 個の関数を定義している。

# 抽象構文木$ast にセットする関数	# do_parse のサブルーチン
1. set_ast()	36. buffering_first()
# ヘルパー関数	37. buffering()
2. forced_end()	38. parse_main()
3. sprint_usage()	39. element_reading()
4. print_help()	40. content_reading()
5. quote_validation()	41. pcdata_reading()
6. org_split()	42. cdata_reading()
7. message()	43. etag_reading()
8. parse_error()	44. attribute_reading()
9. ltrim()	45. comment_reading()
10. rtrim()	46. xmldecl_reading()
11. trim()	47. pi_reading()
12. wstrim()	48. dtd_reading()
13. get_num()	# do_print のサブルーチン
14. get_num_opt()	49. print_main()
# コマンド・オプション解析関数	50. sprint_attribute()
15. input()	# do_get のサブルーチン
16. get_ope()	51. get_main()
17. eval_cmd()	52. match_child_depth()
18. exec_ope()	53. match_child_name()
19. eval_opts()	54. match_child_attr()
20. check_opt()	55. match_child_attr_name()
21. help_option()	56. match_child_attr_value()
# AST のノードに対する情報操作関数	57. eval_opt_depth()
22. add_info()	58. check_opt_name()
23. is_root()	59. eval_opt_attr()
24. is_last_child()	60. check_opt_attr_name()
25. exist_root()	61. check_opt_attr_value()
26. get_path()	# do_del のサブルーチン
# 各種コマンドのメインルーチン	62. get_node_del_next()
27. do_parse()	63. del_node()
28. do_print()	# do_ls のサブルーチン
29. do_get()	64. ls_node()
30. do_del()	# do_cd のサブルーチン
31. do_quit()	65. cd_main()
32. do_help()	
33. do_pwd()	
34. do_ls()	
35. do_cd()	

5. プログラムマニュアル

```
============================================================
                                  version: 1.0
    Welcome to XML-Parser           built: 2010/01/05
                                   author: Takai(ne180081)
============================================================
  type "help" if you want to read command list.

> help
---------+--------------------------------------------------------------
 Command | Description
---------+--------------------------------------------------------------
  parse  | Parse XML file and make AST
  print  | Print current node as XML
  get    | Search elements
  del    | Delete an element and the children recursive
  quit   | Quit this program
  q      | Alias of "quit"
  exit   | Alias of "quit"
  help   | Show the command list
  h      | Alias of "help"
  pwd    | Print current path with nodeID
  ls     | List children node
  cd     | Move to an another node from current node
---------+--------------------------------------------------------------
  You can read usage of the command. Type command with [-h] option.
```

上記はプログラムを実行した後に help コマンドを実行した結果です。このプログラムでは次のコマンドが用意されています。[-h]オプションが付けてコマンドを実行すると、そのコマンドの使い方が表示されます。オプションとは、"-XXXX" のように先頭にハイフンが付いた単語です。単語は空白で区切られますが、引用符に括られた文字列は空白を含んでいても 1 つの単語として認識されます。

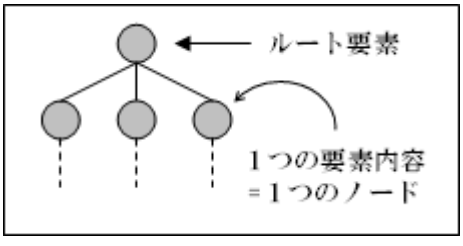
1. parse コマンド
2. print コマンド
3. get コマンド
4. del コマンド
5. quit コマンド (q, exit)
6. help コマンド (h)
7. pwd コマンド
8. ls コマンド
9. cd コマンド

5-1. parse コマンド

```
> parse -h
Desc : Parse XML file and make AST
Usage: parse <FILE>
```

parse コマンドは入力された<FILE>を XML として解析します。<FILE> には解析したいファイルへのパスを入力します。[parse "../The XML document.xml"]のように空白を含むファイル名を指定することもできます。

このプログラムでは「現在地」という概念を実装しています。解析が完了するとルート要素が現在地に設定されます。地とは抽象構文木における「ノード」を意味します。この現在地のことを「現在ノード」または「current node」と呼びます。



5-2. print コマンド

```
> print -h
Desc : Print current node as XML
Usage: print [options]
 [options] are as follows:
       -root : print root node as XML
       -etag : excepts End-Tag
         -id : add id information as prefix
```

print コマンドは現在ノードを基点に XML としての構文で内容を出力します。このコマンドには次のオプションがあります。

-root : 現在ノードではなくルートノードを基点にします

-etag : 終了タグの出力を省略します

-id : ノード ID を接頭辞に付加します

5-3. get コマンド

```
> get -h
Desc : Search elements
Usage: get [search-options]
 * All options adopt "FULL-match" and "AND-search"
 [search-options] are as follows:
     [option] : -opt <arg1> <arg2> ...
        -name : only <arg1> elements
                # -name div
        -attr : has attributes <arg1> & <arg2> & ...
                # -attr foo="hoge" bar="fuga piyo"
   -attr_name : has attribute names <arg1> & <arg2> & ...
                # -attr_name id class title
  -attr_value : has attribute values <arg1> & <arg2> & ...
                # -attr_value section
       -depth : only node which is depth=<arg1> (depth of root element is 0)
                # -depth 3
           -c : only children of the current node
          -id : only children of a node which is id=<arg1>
           -r : search recursive (for [-c] or [-id] option)
        -all : contains "PCDATA" node
```

get コマンドは全てのノードを対象に、そのノードの idx, id, depth 情報と開始タグ（PCDATA であれば"(#PCDATA)"という文字列）を出力します。また、該当したノードをカウント(cnt)します。オプションによって対象とするノードを指定できます。条件は完全一致・AND 検索となります。

-name : 入力された単語と一致する要素名のノード（要素名は 1 つしか指定できません）

-attr : 「属性名=属性値」の形で入力された 1 つの単語で表現される属性を持つノード（foo="attr value" のように記述できます。属性は複数指定できます）

-attr_name : 入力された単語の属性名を属性に持つノード（属性名は複数指定できます）

-attr_value : 入力された単語の属性値を属性に持つノード（属性値は複数指定できます）

-depth : 入力された数値と一致する深さのノード（ルート要素の深さは 0 です）

-c : 現在ノードと、その子ノードのみを対象にします

-id : 入力された ID と一致するノードと、その子ノードのみを対象にします

-r : [-c] と [-id] オプションに対して、子ノードを再帰的に対象にします

-all : 親ノードが対象であるときに、PCDATA のノードも対象に含めます

5-4. del コマンド

```
> del -h
Desc : Delete an element and the children recursive
Usage: del <target>
 <target> is as follows:
     [option] : -opt <arg1> <arg2> ...
          -id : delete a node which is id=<arg1>
         -idx : delete a child node which is idx=<arg1>
```

del コマンドは構築した抽象構文木から指定したノードを削除します。削除されたノードの子孫ノードも一緒に削除されます。ノードを指定するには次のオプションを用います。

-id : 入力された数値と一致する ID のノードを削除します

-idx : 入力された数値と一致するインデクスの子ノードを削除します

両方のオプションが指定された場合、[-idx]オプションは無視されます。ノードが削除されると、他のノードの ID 情報やインデクス情報が更新されます。

5-5. quit コマンド

```
> quit -h
Desc : Quit this program
Usage: quit
```

quit コマンドはプログラムを終了します。q および exit は quit のエイリアスです。

5-6. help コマンド

```
> help -h
Desc : Show the command list
Usage: help
```

help コマンドはコマンドリストを出力します。h は help のエイリアスです。

5-7. pwd コマンド

```
> pwd -h
Desc : Print current path with nodeID
Usage: pwd
```

pwd コマンドは現在ノードのパスを出力します。ルートノードから現在ノードまでの全てのノードの要素名をインデクスと ID 情報を付加して出力します。現在ノードが要素でなく PCDATA であれば要素名の代わりに"(#PCDATA)"という文字列を出力します。

5-8. ls コマンド

```
> ls -h
Desc : List children node
Usage: ls [option]
 [option] is as follows:
           -l : use a long listing format
                format: [idx] (id) <CONTENT>
```

ls コマンドは現在ノードの子ノードの一覧を出力します。[-l]オプションを付けると ID 情報も出力します。このコマンドの出力内容は、[-c]オプションを付けた get コマンドの出力内容と似ています。

5-9. cd コマンド

```
> cd -h
Desc : Move to an another node from current node
Usage: cd <keyword>
 <keyword> is destination, as follows:
        <num> : move to a child node which is idx=<num>
            / : move to root node
          ../ : move to the parent node
    [option] : -opt <arg1> <arg2> ...
         -id : move to a node which is id=<arg1>
```

cd コマンドは現在ノードを他のノードに設定します。つまり現在ノードから他のノードに移動します。

数値 : 入力された数値と一致するインデクスの子ノードに移動します

/ : ルートノードに移動します

../ : 親ノードに移動します (../../ のように入力することもできます)

-id : 入力された数値と一致する ID のノードへ移動します

6. おわりに

6-1. プログラムの作成に関して

最終課題で求められている XML パーザの作成において次の点に苦労した。

1. (XML の構成要素を仕様通りにするか、簡単にするかに拘らず) XML を解析すること
2. パーザとしてどんな機能を持たせるか設計すること
3. 機能を実現するためにデータ構造を考えること
4. 機能を Perl で実現すること (Perl 言語に関する知識、アルゴリズムの考察)

コマンドを自由に設計する点で XML パーザとしてどんな機能があると面白いのかを考えるのに苦労した。結果的に parse, print, get コマンドを実装した後に、「現在地」の概念を取り入れて操作できるようにすることしか発想できなかった。parse, print 以外のコマンドが 1 個なのは寂しいと思ったが、set, add コマンドのうまい設計がなかなかできなかったので、ノードを削除するだけの del コマンドを実装するまでとした。結局 set, add コマンドを実装しなかったのは、parse, get コマンドだけで相当な行のコードを書いてしまったことが一番大きな理由である。

何れにしても、この課題で最も重きをおいた点は XML パーザとしての解析の精度である。正しい XML を正しいと判断し、不正な XML を不正と判断するその基準をできる限り本来の仕様に準拠させたことがこのプログラムにおける特徴の一つとして挙げられる。また、思いつく限りは関数でリファレント渡しにすることで不要なコピーを避けたり、正規表現も含めたある手続きがより効率的になるようにコードを改善したつもりではあるが、Perl や正規表現の書籍を一度も読んでいないので、何らかの点において常道とされる方法に気付いていないことがあるかもしれない。

バッファリングを行いながら解析するようにしているが、バッファリングの恩恵をあまり受けられていないように思う。バッファリングによるメリットの 1 つとして一時的に必要となるメモリ空間を最小限に抑える効果があるが、バッファリングの単位が 1 行なので元々の XML が 1 行に書かれていては意味がない。また、抽象構文木や導出テーブルの設計を十分考察していないため、そのデータの保持のために必要とするメモリ空間が必要以上に大きくなっていると思われる。

6-2. この講義に関して

この講義の授業内容を振り返っておく。

第 1 回	2009/09/29	第 1 話「XML」: テキストによる情報表現・マークアップ・XML の特徴
第 2 回	2009/10/06	第 2 話「DTD」: 文書型定義・DTD の構成・XML の妥当性・正規表現 第 3 話「Perl」: 環境構築・実行方法・基本データ構造と制御構造 小課題 1 : FizzBuzz
第 3 回	2009/10/20	第 4 話「正規表現」: 正規表現の基礎・マッチのルール・後方参照 小課題 2 : It's alphabetical order／メールアドレス／google／ 「()」で囲まれた部分, 括弧がネストされていた場合は？ 第 5 話「I/O」: open・特殊変数$_・左辺や引数を省略したコード
第 4 回	2009/10/27	小課題の解説 : 小課題 2, 小課題 1・省略コード or die・後置 if 文・/usr/share/dict/words を使ってみる 第 6 話「for 文とリスト」: for 文・ファイル入力・バッファリング
第 5 回	2009/11/10	第 7 話「サブルーチンと変数のスコープ」: サブルーチン・my 変数・ バッファリングをサブルーチン化・単語頻度表・置換演算子 レポート 1 : Alice の単語頻度表・チャレンジ（sort・lc・単語数）
第 6 回	2009/11/17	レポート 1 の解説 : バッファリングと置換演算の使い方 第 8 話「括弧構造の解析」: 括弧と木・ネストした 1 種類の括弧・ ネストした 4 種類の括弧・Perl でスタック レポート 2 : ネストした 4 種類の括弧構造の解析
第 7 回	2009/11/24	レポート 2 の解説 : カウンタをスタックに・括弧の対はハッシュで 第 9 話「関数の副作用」: 破壊的代入 第 10 話「関数呼び出しとスタック」: 最内最左・変数表のスタック
第 8 回	2009/12/01	第 11 話「ブロック引数」: ハードコーディングを何とかしたい・ 引数に条件を（文字列・正規表現・関数・ブロック）で 第 12 話「明示的なスタックの排除」: ループと再帰で全探索・ parenthesis・b_paren・e_paren・contents レポート 3 : レポート 2 を改良 : 再帰・条件を引数渡し・内容を出力
第 9 回	2009/12/08	レポート 3 の解説 第 13 話「抽象構文木」: 構文解析と抽象構文木・数式の構文木・ 多次元配列・無名配列・ref 関数・木構造のプリント
第 10 回	2009/12/15	筆記テストについて・最終課題・授業評価 第 14 話「対話型評価器」: Read-Eval-Print Loop・電卓造り
第 11 回	2009/12/22	筆記テスト
第 12 回	2010/01/13	筆記テストの解説・最終課題の質問受け付け

「Perl と正規表現」という表題が付けられそうな授業展開で内容が進み、ネストした複数種の括弧を解析する方法まで説明した後で最終課題「XML パーザの実装」が与えられた。最終課題を完成させるには「Perl の基礎」・「XML の構成要素」・「抽象構文木のデータ構造」・「要素解析のアルゴリズム」・「再帰」について理解・考察する必要があるが、それを自力でできる学生はどのくらいいるだろうか。

この講義において想定している学生のレベルが分からないが、私自身、授業中の話の展開や内容が難しいのではないかと思う場面が何度かあった。Perl 入門・正規表現入門といった講義ではないため理解度の低い学生に合わせる必要は無いが、どこまで知識があることを前提とするかは考え物かもしれない。

6-3. 学習資料・参考文献

私が Perl と正規表現の学習および XML パーザ作成のために参照した文献を記しておく。

Perl 入門

Perl5 編　トップページ

http://www.geocities.jp/ky_webid/perl5/

Perl 講座 [Smart]

http://www.rfs.jp/sb/perl/

Perl - Wikibooks

http://ja.wikibooks.org/wiki/Perl

プログラミング言語 Perl 詳細リファレンス - Perl 入門～サンプルコードによる Perl 入門～

http://d.hatena.ne.jp/perlcodesample/20080229/1204271923

Perl・正規表現

Index of /docs/perl/5.10.0

http://perldoc.jp/docs/perl/5.10.0/

Index of /perldocjp

http://argrath.ub32.org/perldocjp/

Language reference - perldoc.perl.org

http://perldoc.perl.org/index-language.html

Perl メモ

http://www.din.or.jp/~ohzaki/perl.htm

読み物

鳩丸よもやま話 | bakera.jp

http://bakera.jp/yomoyama

水無月ばけらのマニアックな文法論議

http://www.ne.jp/asahi/minazuki/bakera/html/sgml/

bingo_nakanishi の他言語出身者のための Perl 入門

http://d.hatena.ne.jp/bingo_nakanishi_perl/

404 Blog Not Found

http://blog.livedoor.jp/dankogai/

XML 仕様書・XML Validator

Extensible Markup Language (XML) 1.0 (Fifth Edition)

http://www.w3.org/TR/REC-xml/

The W3C Markup Validation Service

http://validator.w3.org/#validate_by_input

この他、XML の仕様を確認するために次のブラウザのエラーチェックを用いた

Firefox 3.5.7

Google Chrome 3.0