2007－NL－178 (8)
2007／3／28

正規表現による書き換えに基づく関数型言語REFLとその言語処理への応用

中西克明
東京大学
k-naka@phiz.c.u-tokyo.ac.jp

概 要

自然言語処理の対象は高度に構造化されている一方でしばしば、また微妙に構造を逸脱する。関数プログラミングに代表される従来のパラダイムは複雑に構造化された対象を整理して扱う強力な方法論である一方、構造を逸脱する対象に対する柔軟性を欠いているとも言える。本稿では言語REFLを提案し、この言語が関数プログラミング的な構造処理の能力を保持しつつ、構造を逸脱するような対象に対してより柔軟な処理を可能とするものであることを、主に自然言語処理のプログラミング例などを通じて示す。

REFL: A regular expression based functional language

Katsuaki Nakanishi
University of Tokyo

Abstract

NLP deals with things that are structured to a great extent but not always completely reducible to structures. Functional programming and other conventional programming paradigms, while they provide powerful tools to process complex structures systematically, can be inefficient for data that deviate from the structures. In this paper, we present a programming language called REFL and show, through examples in NLP, how this language can be more effective than other languages in dealing with deviations from structures while retaining the advantages of functional languages.

1 はじめに

対象を構造において捉え、その構造に即した一群の形式的手続きによって処理、分析することは一般的な科学的態度である。この時、対象の示す複雑な挙動はその対象の持つ構造の複雑さとして理解され得るだろう。即ち、一見複雑で捉えがたく見えた対象は、各々としては単純な関係性が複雑に積み重なった構造物であり、それぞれの局所的な部分構造に対応した操作や認識の積によって扱うことが出来るとされる。これは複雑なものを処理する強力かつ一般的な戦略である。日常的な例で言えば、インターネットブラウザに大量のブックマークを登録する場合、検索性を高めるために、それをディレクトリ構造によって分類している人は多いはずだ。関数型言語等のある種のプログラミングスタイルは、この戦略の端的な実現として見ることが出来る。そこでは、帰納的に定義されたデータ構造の上で関数が(引数がデータ構造に沿ってマッチングされることにより)そのデータ構造に沿った形で再帰的に定義される。例えば自然数上の諸演算は0と後者関数から帰納的に定義される自然数というデータ構造に沿って再帰的に定義され得る。

階層構造による整理は人間の理解の様式に合っており、関数プログラミング的手法は、対象が構造によって厳密に捉えられるようなものである限り、とても強力なツールと言える。しかし同時にこの戦略は構造の複雑化によって対象を十分に捉えられると言う前提に依存しており、構造を逸脱するようなデータに対して融通が利かず、非効率的な扱いしか出来ないということがあり得る。現実には構造による対象の表現は近似でしかなく、構造の複雑化では解消出来ないようなギャップがあり得るだろう。ブックマークの例でいえば、検索性向上のために複雑なディレクトリ構造によって分類しているような場合、新たに追加するブックマークをその構造の何処に分類すべきか迷うということがしばしば起こる。或いは分類の途中で、既存の分類とは互換性を持たない新たな分類の軸を導入したくなることもあるかもしれない。

こうして関数プログラミング的なパラダイムは複雑性のジレンマを抱えることになる。複雑に入り組

んだ構造を明快に整理して処理できる一方で、その明快さは構造の厳密な適用によって可能になるのであって、構造を逸脱する対象に対する柔軟性の犠牲の上に成り立っていると言えるのだ。

1.1 NLP における複雑性のジレンマ

自然言語処理はこの複雑性のジレンマが端的に表れる領域の一つである。自然言語は高度に構造化されている一方でしばしば構造を逸脱する。構造からの逸脱が表面的な物である場合は構造分析に適した形へと正規化する処理を別に設けることで容易に対応できるだろう。例えば文の統語構造は木構造で表されるが、実際の文面とそのような木構造の終端記号列には clitic や単語の活用、単語の区切りなどの問題からもたらされるギャップがあるため、言語リソースからの入力を直接構文解析器にかけることが出来ないというような問題であれば、間に形態素解析などの処理を挟むことによって解決出来る。正規表現を備え、強力なテキスト処理能力を持ったスクリプト言語で雑多な言語リソースをまず正規化し、その後、関数プログラミング的な手法でそれを分析するといったパラダイムが考えられるのだ。しかし、構造からの逸脱が常にこのように外部のモジュールへと切り離せるようなものであるわけではない。例えば、compositionality と idiom の問題 [5]、或いは nonconstituent coordination の問題 [3] などは構造からの逸脱の例でありながら、構造から切り離して論じることも出来ないようなものである。

或いは、近年の自然言語処理の重要な分析対象である HTML 文章等のデータも同様の傾向を示すと言える。HTML 文章は理想的にはタグによって階層構造化されているべきものだが、現実の HTML 文章は或いは人間が、そのレンダリング結果を人間に見せるという目的で書いているため、形式的手順にとって必ずしも可読性の高いものではない。典型的にはタグによる構造が well-formed な木構造を成さない (開始タグと閉じタグの非対応や`<b>x<i>y</b>z</i>`のようなタグ構造のオーバーラップなど) という問題がありうるし、利用したい情報がどの部分にあるのかということを、例えそれが人間にとって視覚的にわかりやすいものだとしても、タグによる階層構造から形式的に判別することも必ずしも容易でない[1]。

本稿では、以上概観した従来型のプログラミングパラダイムにおける複雑性のジレンマの問題に対し、より柔軟に対応できるプログラミング言語 REFL を提案し、それがどのように複雑性のジレンマの問題に有効であり得るのかを、特に自然言語処理への応用を例として解説する。

2 純粋 REFL

2.1 言語仕様

純粋 REFL(以下単に REFL) の言語仕様は以下に示す通り非常に単純である。一般の関数型言語と同様、REFL の任意のプログラムは関数定義の集合であり、プログラムの実行とは、式を定義された関数群を用いて評価する、即ち初期式をそれが正規形になるまで簡約しつづけることである。一般的な関数型言語等において、式が変数、定数、関数、演算子、リテラル等を表すアトムや構文などから構成されるのと異なり、REFL において式とは一つの文字列である。その各文字は 8-bit で表される byte であるか三つの特殊文字、FS, FOB, FCB[2] の何れかである。REFL プログラム内及び本稿においては FS, FOB, FCB を '^', '(', ')' によって表す[3]。式が ^...(...) という形の部分文字列 (但し... は共に byte 列[4] とする) を含むとき、式は簡約可能であるとされる。このような部分は最小関数表現と呼ばれ、'^' と '(' の間の byte 列が関数を表し、'(' と ')' の間の byte 列がその関数への引数を表している。簡約可能な式の簡約とはその式の最小関数表現の内、再左のものを、その関数を引数に適用した結果得られる文字列で置き換えることである。実行の途中で関数適用が失敗するか、正規形になる前に簡約不可能となった場合はプログラムの実行は失敗したとされる。REFL の正規形とは byte 列のことである。

REFL の関数は byte 列を引数として取り文字列を返すか、或いは関数適用に失敗する。REFL のプログラムは関数定義行の列として書かれる。各関数定義行は何れかの関数の定義の一部であり、$name(pattern) = value$ の形をしている。$name$ は '(' 以外の表示可能文字の列であり、$pattern$ は

[1] xml への動きはこうした問題に対する解答の一つであるが、仮に xml が十分に普及したとしても、複雑性のジレンマが解消するわけではない。xml の方向性はテキストを予め定めた標準で構造化することで利便性を高めるというものだが、これ自体が、前述のブックマークの分類と同種の複雑性のジレンマを持ちうる。

[2] それぞれ function start, function opening bracket, function closing bracket を意味する。

[3] 文字 '^', '(', ')', '\' 及び改行文字はエスケープシーケンスを用いて '\^', '\(', '\)', '\\', '\n' と表わす。

[4] 本稿で byte 列とは特殊文字 (FS,FOB,FCB) を含まない、各文字が byte であるような文字列のことである。

```
max((.)(.)(.*))=^max0(^win(\2\3)\0)
max0(T..(.*))=L\1
max0(^F..([^0]*)$)=D\1
max0(.(.)(.)(.*))=^minimax(^next(\1\2/\3))\3
next((.)(.)(.*?)/(.*?)0(.*))=^max(\2\1\3\4\1\5)^next(\1\2\3\40/\5)
next()=
win(^(.)((...)*\1\1\1|.*\1..\1..\1|\1...\1...\1|..\1.\1.\1))=T
win()=F
minimax(L([012]*))=W\1
minimax(D([012]*))=D\1
minimax([012]+)=L\0
```

図 1：三目並べ

正規表現[5]である。*value* は文字列であるが、その各文字には byte と三つの特殊文字 (FS, FOB, FCB) の外に 10 個の更なる特殊文字 (エスケープシーケンスを用いて\0,\1,...,\9 と表される) を使うことが出来る。*name* は定義する関数の名前を表しており、同じ *name* 部分をもつ関数定義行は同じ関数を定義している。関数の引数への適用は、その関数を定義するどの定義行の *pattern* もその引数にマッチしない場合失敗となる。成功した場合、*pattern* が引数にマッチするようなその関数の定義行のうち最初のものの *value* が値として返される。この戻り値において、*value* 内の各\n (n は 0〜9 の何れか) は引数の *pattern* によるマッチングにおいて n 番目の部分正規表現[6]がマッチした部分 byte 列によって置き換えられる。

以上が REFL の言語仕様の全てである。例えば以下は二つの定義行によって byte 列を反転する関数 reverse を定義する REFL プログラムである。

```
reverse((.*)(.))=\2^reverse(\1)
reverse()=
```

この時、'DOG is ^reverse(DOG) spelled backwards.' という式の評価は以下のように進む。

DOG is ^reverse(DOG) spelled backwards.
→ DOG is G^reverse(DO) spelled backwards.
→ DOG is GO^reverse(D) spelled backwards.
→ DOG is GOD^reverse() spelled backwards.
→ DOG is GOD spelled backwards.

(上では簡約可能な式の再左最小関数表現に下線を引いた。) 最初の 3 つの簡約ステップでは、reverse 関数の最初の定義行が使われ、最後の簡約ステップで初めて、最初の定義行の *pattern* が引数にマッチしなくなり、二つ目の定義行が使われる。

[5]正規表現には様々な方言がある [1] が、ここでは PCRE (http://www.pcre.org/) の文法を前提とする。

[6]但しある正規表現の 0 番目の部分正規表現とはその正規表現全体を指すこととする。

2.2 例

REFL の一般的なプログラミングの実例として、三目並べ (マルバツ、tic tac toe) の任意の局面からの最善手を mini-max 法で探査するプログラムを見たい (図 1)。

任意の局面からの最善手を探査するには ^max(<player><opp_player><board>) を評価する。ここで <board> は盤面を、先手 (○) を 1、後手 (×) を 2、空白を 0 とした三進法九桁で表したものとし、<player> は次の手を打つ選手、<opp_player> は <player> の相手の選手を表したものとする (先手:1、後手:2)。戻り値は最初の文字が、局面の <player> にとっての評価 (必勝:W、必敗:L、引き分け:D) であり、続いて両選手が最善手で打ち合った場合の局面の推移の一例を盤面 (三進法九桁) の羅列で表す (順序は推移の逆)。例えば、誰も手を打っていない初期状態からの最善手を探索するには ^max(12000000000) を評価する。この場合戻り値は D112221121...(以下略) となる。

max 関数は <board> が <opp_player> の勝ちの局面であれば <player> 必敗とし、そうでない場合、全てのマス目が埋まっていれば引き分けとする。これらの場合、局面推移としては <board> のみが返される。その他の場合は可能な次の局面に対する max 関数の適用結果から mini-max 法によりこの局面の最善手を決定する。局面推移は最善手を打った次の局面に対する max 関数の適用結果の局面推移に <board> を繋げたものとなる。

^win(<player><board>) は <board> において <player> の手が一列に並んでいれば T に簡約され、そうでなければ F に簡約される。

^next(<player><opp_player>/<board>) を評価すると <board> において <player> が次の手を打ったような各局面に対しての max 関数の返り値を並べたものが返る。この next 関数の返り値に対して minimax 関数を適用すると、その中に一つで

も必敗の盤面推移があれば、その盤面推移を必勝のものとして返し、必敗の盤面推移はなくとも引き分けのものがあれば、その盤面推移を引き分けのものとして返し、さもなければ盤面推移の一つを必敗のものとして返す。

2.3 記述力

REFL は §2.1 で見たとおり非常に単純な仕様の言語であるが、高い記述力を持っており、様々な処理を簡潔に実装できる。実際、本稿で見るサンプルプログラムは（§3.2 の辞書検索のサンプルプログラムにおける httpget 関数を除いて）§2.1 の単純な言語定義だけで、ライブラリや組み込み関数などは一切使わずに実装されている。[4] では整数上の諸演算の純粋 REFL での簡潔な実装が紹介されている。ここでは、そのような記述力の高さを説明するような、§2.1 の言語定義から導かれる REFL のいくつかの特性を見る。特に、§1 で提起した複雑性のジレンマの問題への適性に関係するような側面についても注目する。

各関数定義行は正規表現による文字列の書き換えを指示しており、これによって基本的なデータ操作が成される。このとき、書き換えの右辺 (関数定義行の *value* 部) に FS, FOB, FCB を含められることから、再帰的なものを含めた関数呼び出しが可能となる。また、引数が *pattern* にマッチするような最初の関数定義行を用いるという計算機構[7]により、制御構造を表現することができる。例えば、三目並べのサンプルでは max 関数から呼び出される max0 関数は win 関数の返す真偽値によって処理を分岐させている。

この case 文に相当する制御構造では特殊と一般（の間のグラデーション）の関係を簡潔に記述することが出来る。正規表現による書き換えをこのような形で再帰的に組み合わせることが出来る REFL は例えば多様なバリエーションを持つ書式に対する柔軟な処理を簡潔に書けるなど、強力なテキスト処理の能力を持つことになる。或いはこの再帰の構造の中に単なるテキスト処理とは別の操作、例えばインターネットからのデータの取得などを取り入れれば、動的に HTML 文章を取得、解析しつつインターネットを渡り歩くような crawler 的処理等も容易に書ける。後述する辞書検索のサンプルは、そのような処理の簡単な例でもある。

一般的な関数型言語等において、式が識別子、リテラルや構文などから構成されるのと異なり REFL では式は単に文字列である。関数名も関数を識別するアトミックなトークンではなく、式の再左最小関数表現においてたまたま FS と FOB の間に来た byte 列が関数名として認識されるだけだ。このことの帰結の一つは、REFL は高階関数のための特別な言語機構を持たないが、`twice((.*?),(.*))=^\1(^\1(\2))` のようにして、ある種の高階関数的なプログラミングが可能だということである[8]。

引数も同様に常に一つの byte 列である。データ構造や引数の複数性は、それを何らかの仕方で byte 列にエンコードすることで行なう。例えば引数の複数性やリストは、上記の twice 関数におけるように、区切り文字を決めてそれで区切ることで表現できるし、木構造のような階層構造は`<S <NP John> <VP smiled>>`のように括弧を使って表現できる[9]。このとき例えば`<[^<]*?>`のような正規表現によって高さ 1 の local tree にマッチングさせること等が出来る。実際、正規表現で切り出せる書式の多様さに応じて、様々なデータ構造のエンコードがあり得るだろう[10]。このような、データの byte 列へのエンコードと正規表現によるマッチングの組み合わせの記述力は高い。例えば、三目並べサンプルでは盤面は 3 進数 9 桁にエンコードされているが、win 関数はその盤面で、指定された選手の手が一列に並んでいるかどうかの判定を一つの正規表現で行なう。

ここで、とりわけ §1 で見た複雑性のジレンマの問題に関連して重要なのは、普通の関数型言語等においてはデータ自体が構造を持っており、プログラムはそのデータ構造に沿ってデータにアクセスするのに対して、REFL ではデータ自体はどれも単なる byte 列であって、それを関数側が (各々の恣意的な

[7] これは現代的な関数型言語で一般的に許容されている関数定義の書きかたでもある [2, p.388-391]

[8] 但しラムダ式のような匿名関数を生成する機構は REFL にはない。

[9] 区切り文字や括弧のリテラルとしての使用が必要ならエスケープシーケンスを使うなどする。実際、木構造 (階層構造) は括弧を使ったエンコードのほかに、リストを区切り文字をエスケープしてネストさせることによっても表現できる。括弧を使った木構造のエンコードでは高さ 1 の local tree にマッチングできるため、言わば bottom-up の再帰的処理が簡単に書けるが、今述べたようなリストのネストによる木構造のエンコードではルートが直接支配する local tree を簡単にマッチングできるため通常の関数型言語で木構造を扱う時と同様の言わば top-down の再帰的処理が書ける。用途に応じて適したエンコードを選ぶことになる。(もちろん括弧を使った木構造のエンコードでも、ルートが直接支配する local tree を取り出す関数を使うなどして top-down の再帰的処理を書く事は出来るし、逆も同じである。)

[10] 例えば red:aka,blue:ao,green:midori のような形で書かれた連想リストを検索する関数 lookup は `lookup((.*?),(|.*?,)\1:(.*?)(,|$))=\3` と書ける。

```
start()=^cyk(o/^initial(o,^sentence() ))
sentence()=i saw a man with a telescope
lexicon()=NP:i,N:man,N:telescope,V:saw,D:a,P:with
rules()=S:NP VP,NP:D N,NP:NP PP,VP:V NP,VP:VP PP,PP:P NP
initial(^(o+,)(.*?) (.*))=\1\1^initial_0(\2,^lexicon(),)/^initial(o\1\3)
initial(o+)=
initial_0(^([^,]+),(|.*?,)([^,]+)\:\1,(.*))=[\3 \1],^initial_0(\1,\4)
initial_0()=
cyk(^(o+)/(.*?/o\1.*))=^cyk(o\1/^compact(^update(\1/\2)))
cyk(^(o+)/(|.*/)o,\1,(.*?)/)=^result(\3)
result(^(|.*?,)(\[S .*?),(.*))=\2\n^result(\3)
result()=
compact(^(|.*/)o+,o+,/(.*))=^compact(\1\2)
compact(^(|.*/)(o+,o+,)([^/]*)/(|.*/)\2([^/]*/)(.*))=^compact(\1\2\3\5\4\6)
compact(.*)=\0
update(^(o+)/((|.*?/)(o+,)(o+,)(.*?),/)((|.*/)o\5\1\4(.*?),/.*))
                                        =\2\4\1\4^new(\6/\9/^rules(),)/^update(\1/\7)
update(^o+/(.*))=\1
new(^(.*?),(.*)/(.*/.*))=^new(\1/\3)^new(\2/\3)
new(^(.*)/(.*?),(.*)(/.*))=^new(\1/\2\4)^new(\1/\3\4)
new(^(\[(.*?) .*?)/(\[(.*?) .*?)/(|.*?,)([^,]*):\2 \4,(.*))=[\6 \1 \3],^new(\1/\3/\7)
new()=
```

図 2：CYK 法

仕方で) 構造を持ったものとして解釈する、別の言い方をすれば構造は各関数がデータに投影するものであるということだ。これは、逆にいえばREFLにおいては意図された構造を無視したマッチングが出来るということでもある。例えば括弧で表された木構造を引数としてとりながら、その階層構造を全く無視して特定の単語を検索するようなことも出来るし、或いはある文字で区切られたリストを引数として取りながら、それを別の文字を区切り文字とするリストとして解釈しても良い。REFLでは各簡約ステップ、各関数適用のそれぞれにおいて、それぞれの関数が任意に設定した構造に沿って、或いは構造を無視して、処理が成されるのであって、関数プログラミング的な、構造を利用した処理の能力を保ちつつ、構造に縛られない処理を処理の任意の段階で行なうことが出来ると言える。

次節では自然言語処理の現実的なプログラミングを例にとり、ここで述べたようなREFLの記述力を具体的に確認する。

3 REFLの言語処理への応用

3.1 統語解析

図2はCYK法による文脈自由文法のパーサーである。入力文、文法規則、語彙項目はプログラムの冒頭で定数関数 sentence, rule, lexicon の返り値として指定されている。`^start()` を評価すると指定された入力文を指定された文法規則 (チョムスキー標準形)、語彙項目で統語解析し、可能な全ての構文木を括弧を使った木構造表記で出力する。この例の場合であれば、出力は [S [NP i] [VP [V saw] [NP [NP [D a] [N man]] [PP [P with] [NP [D a] [N telescope]]]]]] と [S [NP i] [VP [VP [V saw] [NP [D a] [N man]]] [PP [P with] [NP [D a] [N telescope]]]]] の二行となる。

以下の解説で n は文字 'o' の個数で数字を表したbyte列とし、*table* は '/' で区切られたCYKセルのリスト (ただしリストの最後にも '/' が付く) で作成中のCYKテーブルを表したbyte列とする。CYKセルは二つの (文字 'o' の個数で表された) 数字 i, j(CYKセルの指標と呼ぶ。またこのとき $j-i+1$ をCYKセルの長さとする) と任意個の構文木 (括弧を使って木構造を表したもの) を ',' で区切って並べたリスト (ただしリストの最後にも ',' がつく) で表され、入力文の i 番目の単語から j 番目の単語までの部分の構文解析結果がそれらの構文木であることを意味する。またCYKテーブル内のCYKセルは、その指標の一つ目の数字に従ってソートされて並んでいるものとする。

table が長さ n までのCYKセルからなるCYKテーブルであるとき、^cyk(n/*table*) を評価すると、テーブルの残りを作り上げ、文全体の構文木として得られたもの (のうちSをルートとするもの) を返す。よって、^initial(o,^sentence()) が長さ1のCYKセルからなるCYKテーブルに簡約されることから、^start() を簡約すると入力文を構文解析した結果が得られる。cyk 関数は再帰的に定義されており、cyk 関数の最初の定義行は指標の最初の数字が $n+1$ 以上であるようなセルが存在するとき、即ち n が文の長さ未満であるとき実行され、後述する compact, update 関数によってテーブルに長さ $n+1$ のセルを加え、再帰的に cyk 関数を呼び出す。cyk 関数の次

の定義行は、その他の場合、即ち全てのセルが作り上げられている場合に実行され、文全体に対する構文木のリストにresult関数を適用したものを返す。result関数は、構文木のリストのうち、ルートがSであるものを改行文字で区切ったリストとして返すので、cyk関数は文全体に対するSをルートとする構文木を改行文字で区切って出力することになる。

*table*が長さnまでのセルからなるテーブルであるとき、`update$(n/table)$は*table*に長さ$n+1$のセルを加えたテーブルを返す。update関数の最初の定義行は、ある数i,jにたいして指標がそれぞれ(i,j)と$(j+1,i+n)$であるようなセルの組(のうち指標(i,j)のセルが*table*において最も左に位置するようなもの)を検索し、その二つのセルからの構文木同士をルートが支配するような全ての構文木(後述するnew関数で与えられる)を持つセル(指標は$(i,i+n)$)をテーブルの指標(i,j)のセルの右の位置に挿入し(この位置に挿入すればテーブル内のセルのソートが崩れない)、テーブルの指標(i,j)のセル以降のリストに対して再帰的にupdate関数を呼び出す。(4番目、5番目の部分正規表現がそれぞれiとjに、6番目が(i,j)を指標とするセルの構文木のリストに、9番目が$(j+1,j+n)$を指標とするセルの構文木のリストにマッチする。)そのようなセルの組がない場合、updateの第二の定義行がテーブルを変更せずに返す。

new関数は構文木のリスト二つと文法規則を'/'で区切ったものを引数としてとり、文法規則から可能な、その二つのリストから一つづつ選んだ二つの構文木をルートが支配するような構文木のリストを返す。最初の二つの定義行は二つの構文木のリストが共に一つの構文木のみを含むような場合へと処理を還元している。引数の二つのリストが共に一つの構文木のみを含むような場合、その二つの構文木のルートの並びを右辺にもつ文法規則があれば三つ目の定義行が実行され、その文法規則に従ってそれらの構文木を支配する新たな構文木が作られ、返される。

このupdate関数の処理では構文木を一つも持たないセルや、同じ指標を持つ複数のセルを追加してしまう可能性があるが、compact関数がテーブルからの空のセルの削除や、同じ指標のセルのまとめを行なう。

このプログラムはCYKアルゴリズムという整理された形式的なアルゴリズムの実装であって、複雑性のジレンマの問題とは基本的に無縁である。しかし、このような形式的なアルゴリズムの実装においてさえ、データ構造を活用出来つつ、それに縛られないというREFLの特性が実装上の効率に結びついている。例えば*table*はCYKセルのリストであり、CYKセルは二つの数と一つ以上の構造木からなるリストであるから、*table*はリストのリストという構造を成しているが、REFLではここから直接、複雑な条件でセルをマッチできる。cyk関数の最初の定義行は$n+1$以上の数を指標の最初の数字とするセルを、次の定義行は$(1,n)$を指標とするセルをマッチしている。或いはもっと複雑な処理も可能であって、update関数の最初の定義行ではCYKテーブルから、指標がそれぞれ(i,j)と$(j+1,i+n)$となるような数i,jが存在するようなセルの組、というものを直接マッチしている。(他にもcompactの第二定義行では、テーブル内の同じ指標をもつセルを纏めるという処理が、またnew関数の第三定義行では、文法規則のリストと二つの構文木に対して、その二つの構文木のルートを並べたものを右辺にもつような文法規則を文法規則のリストから取り出し、それに従って、その文法規則の左辺の非終端記号がルートとしてその二つの構文木を支配する新しい構文木を構成するという処理が、一つの正規表現による書き換えで実現されている。)REFLではこうした処理は特別な機構によるトリッキーなテクニックではなく、通常のデータ構造に即したマッチングと同一平面上にある自然な実践である。

3.2 辞書検索

後述する、純粋REFLに基く処理系の実装に付属するサンプルプログラムの一つにインターネット上の複数のオンライン辞書で単語を引く辞書検索のプログラムがある。各オンライン辞書に対し、単語を引数に取り、その辞書でその単語を引いた結果をシンプルなHTML文章として出力するような関数(以下辞書関数と呼ぶ)が用意され、また一つの単語を全ての辞書で引き、結果を並べて一つのHTML文章として出力する関数[11]が用意されている。各オンライン辞書サイトは、それぞれ異なる複雑な装飾で単語の定義を表示するが、そのサンプルプログラムの各辞書関数はそれを太字と斜体のみを使う単純な装飾に変換し、またその際、強調一般は太字で、例文は斜体で表示するなどし、外見を統一する。

図3はその中のCambridge Advanced Learner's

[11] この関数の実装は§2.3で述べた高階関数的なプログラミングの例でもある。

```
get(.*)=^httpget(dictionary.cambridge.org/\0)
cald(.*)=^cald0(^get(results.asp?searchword=\0))
cald0((?s).*<p>(<strong>.*?</strong> was not found))=\1
cald0(Location: ([^\r\n]*))=^cald1(^get(\1))
cald0((?s)<li><a href='(.*?)'>(.*))=^cald1(^get(\1))<p>^cald0(\2)
cald0()=
cald1((?s)(<span.*)<br><br>)=^cald2(\1)
cald2((?s)(.*)<span class='(.*?)'>(.*?)</span>(.*))=^cald2(\1^cald3(\2,\3)\4)
cald2((?s)(.*?)<a[^>]*>Show phonetics</a>(.*))=^cald2(\1\2)
cald2((?s)(.*?)<(/?)a[^>]*>(.*))=\1<\2b>^cald2(\3)
cald2((?s).*)=\0
cald3((?s)^(cald-example),(.*))=&#149; <i>\2</i>
cald3((?s)^(cald-h?word|def-sensenum),(.*))=<b>\2</b>
cald3((?s)^.*?,(.*))=\1
```

図 3：辞書関数 cald

Dictionary というオンライン辞書 (以下 CALD)[12] を引く辞書関数 cald を定義した部分である。オンライン辞書 CALD では入力した単語に該当する辞書項目が複数ある場合 (多義語などの場合) は、検索結果はそれらの項目に対するリンクの一覧であり、ユーザは列挙された候補から見たい定義を一つ一つ辿って見ていかなくてはいけないが、この辞書関数 cald はそうした場合、列挙された項目を全て読み込み、並べて一つの文章にして出力する。

上記プログラム中で未定義の関数 httpget が使われているが、これは処理系の実装に付属するライブラリの関数で、引数で指定されたアドレス (URL から http://を抜いたもの) に HTTP プロトコルの GET メソッドでアクセスして取得されたデータを返す関数である。

cald 関数は単語の検索結果のページを取得し、それに cald0 関数を適用する。単語の検索結果のページは次の三つの場合で大きく書式が異なる。即ち、該当項目が見つからなかった場合、該当項目が一つ見つかった場合、そして該当項目が複数見つかった場合である。このそれぞれの書式に応じた処理が、cald0 関数の第一、第二、第三 (及び第四) 定義行で記述されている。該当項目が見つからない場合は、その旨を伝えるメッセージをそのまま出力する。該当項目が一つの場合は、その項目の定義のページにリダイレクトされるので、Location ヘッダの値で示されたアドレスを新たに読み込み、cald1 関数に渡す。該当項目が複数の場合は、列挙された定義へのリンクを一つづつ取り出し、リンク先を読み込んでは cald1 関数に渡す。このようにして処理は単一の項目を処理する cald1 関数に帰着する。cald1 関数は HTML 文章全体のうち単語の定義の部分を取り出して cald2 関数へわたす。この定義の部分というのは span タグによる階層構造を成していて、各 span 要素はその class 属性に応じた視覚効果 (外部のスタイルシートで指定された) で表示される。cald2 関数の最初の定義行は、最小の (中に他の span 要素を含まない)span 要素を、その class 属性に応じて cald3 関数で変換する。cald3 関数は、例文を表す class 属性の場合は斜体に (第一定義行)、見出し語や語義番号など強調されるべき要素を表す class 属性の場合は太字に (第二定義行)、その他の場合は装飾なしに (第三定義行) 書式変換する。cald2 関数の最初の定義行が span 要素がある限り再帰的に適用されるので、結局 span 要素の階層構造が言わば bottom-up 的に一つづつ、書式変換されていくことになる。オンライン辞書 CALD の返す定義には span 要素による修飾のほかに、リンクが含まれている場合がある。リンクには二種類あって、一つは発音記号の表示へのリンク、もう一つは定義の中で他の単語に言及する際、そこに張られるその語彙項目へのリンクである。cald2 関数の第一定義行の再帰によって span 要素の階層構造の処理が終わると、同様にして第二定義行で、発音記号の表示へのリンクが削られ、第三定義行でその他のリンクが太字による修飾に変換される。

cald0 関数は必要な情報に効率的に着目して書式の差を認識し、必要に応じて自動的にページ内のリンクを辿り、処理を cald1 関数へと帰着させている。これは §2.3 で述べた REFL の得意とする crawler 的処理の簡単な例でもある。

cald2 関数の最初の定義行は span 要素の階層構造を再帰的に処理している。ここで重要なのは、cald2 関数がここでの処理に必要な span 要素の階層構造だけを処理しているということだ。どういうことか。このような階層構造の再帰的処理に関数プログラミング的な手法が適していることは明らかであるが、普通、そのような手法で HTML 文章の階層構造を扱おうとすれば、まず一般的なライブラリなどを使

[12] http://dictionary.cambridge.org/

い、HTML 文章をタグ構造に従って再帰的なデータ構造にパースする必要があるだろう。しかし、§1.1 で見たように、一般に HTML 文章は整合的な階層構造を成していないので、この方法は使えない。或いは、例えば XML 文章を対象にしているときのように、こうした問題が無かったとしても、やろうとしている処理に全く関係の無い構造まで全てパースするのは単純に非効率的であると言えるだろう。REFL ではどうか。REFL はデータ構造を扱えるが、そもそもデータ構造は byte 列にエンコードして扱うので、HTML 文章のような、構造をエンコードした byte 列はそのままで構造を扱うことが出来、パースの必要がない。そして、必要な構造だけを任意に取り出して (或いは投影して) 処理するので、それ以外の部分の構造の問題に振り回されることがない。例えば cald2 関数の最初の定義行は span 要素の階層構造だけをあつかっているので、その他のタグは何の構造も示さない単なる文字列として扱われる。もしxyzというような不整合なタグ構造がどこかにあったとしても、そうした部分は単なる文字列として処理されるだけなので、何の問題も起きないのだ。

今述べた、REFL では構造をエンコードした byte 列をそもそも扱うので、パースの必要がないというのは、一般的にも大きな利点である。パースの過程を経ずに直接構造化テキストを構造処理出来るのは第一に手軽であるし、或いは XML 文章のように普及した書式でないようなテキストの処理においては、パーサーの調達や自作の大きな手間を省くことが出来る。実は §3.1 の CYK のサンプルでも、入力文や文法規則のリストや語彙のリスト (sentence, rule, lexicon 関数の戻り値) は人間が書きやすい書式で指定されたものであり、通常のプログラミングであればそれを先ずパースする必要があるだろうが、REFL はそのフォーマットをそのまま構造認識に使っている。こうしたプログラムのデータへの距離の短さが、言語を手軽な、直感的なものとしている。

4 まとめ

関数プログラミング的手法は複雑な構造をもった対象を帰納的に定義されたデータ構造と再帰的に定義された手続きによって整理して扱うことが出来る。これは人間の思考形態にも合った、強力な手段であるが、しかしそれは、構造を逸脱するような対象への柔軟性の犠牲の上に成り立つものであった。構造への強い傾向性を示しながら、完全には構造化されきらないような自然を相手にするとき、我々はこの複雑性のジレンマに直面することになる。

自然言語処理の扱う対象もまた、高度に構造化され、構造に沿った処理を要請する一方で、しばしば構造を逸脱する。

本稿では言語 REFL が関数プログラミング的な構造処理の能力を保持しながら、同時に構造に縛られない柔軟な処理の能力を持つことによって、上記の複雑性のジレンマの問題に対してユニークな解答を提示することを、自然言語処理の例で具体的に確認した。

言語 REFL は現代的な要請に応じた実用的なツールとして提案されている。§2.3 で述べ、§3.2 でその簡単な例を見たような、インターネット上のリソースの活用などは REFL がその特性を発揮できる現代的な応用分野の一つだろう。また、言語の定義が簡潔なので簡単に覚えられることやデータを直接的、直感的に扱えるという手軽さから他のプログラムから呼び出して使う組み込み言語としての用途なども期待できる。

本稿で紹介した純粋 REFL に基き、モジュール化や外部 C 関数の呼び出しなどの実用的な機構を備え、多少のライブラリやサンプルプログラムを付属させた処理系の実装 (Windows 版) が http://taurus.c.u-tokyo.ac.jp/nakan/refl.html で公開されている。今後の研究の方向性としては、モジュール化、notation の改善、GUI プログラミング、並列処理などの実際の応用を促進する機構の研究及びそれらの処理系への実装、またライブラリの拡充などを目指すことになる。

参考文献

[1] Friedl, J. E. F. (2002). *Mastering Regular Expressions, Second. Edition.* O ' Reilly.

[2] Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, Vol. 21, No. 3, pp. 359-411.

[3] Milward, D. (1994). Non-Constituent Coordination: Theory and Practice. In *Proceedings of COLING 1994*, pp. 935-941.

[4] Nakanishi, K. (2004). *Introducing REFL.* http://taurus.c.u-tokyo.ac.jp/nakan/refl.html

[5] Westerståhl, D. (1999). Idioms and compositionality. In J. Gerbrandy, M. Marx, M. de Rijke, and Y. Venema (eds.), *JFAK. Essays Dedicated to Johan van. Benthem on the Occasion of his 50th Birthday*, Amsterdam University Press.