PROGRAMMABLE CONTEXTUAL ANALYSIS

David J. Ittner

Henry S. Baird

AT&T Bell Laboratories
600 Mountain Avenue, Room 2C-306A
Murray Hill, NJ 07974-0636 USA
d.ittner@att.com

ABSTRACT

We describe a method for rapid prototyping of contextual analysis algorithms within an experimental page reader. Due to the great variety of such algorithms and their dependency on details of the page-reader's internal data structures, the state of the art today is that each new application requires custom low-level programming. This is undesirable since it impedes experimentation and restricts cost-effective applications to high-volume problems. To make contextual analysis more easily retargetable, we have designed a high-level language with primitives for traversing the document hierarchy and generating, scoring, sorting, and pruning interpretations. It is based on Ousterhout's interpreted language `tcl` which provides constructs such as variables, decisions, looping, etc, and is easily extended by adding functions. Some functions are table-driven or built-in: for example, character typing, typographical morphology analysis, and regular expression matching. Other functions are normally implemented as separately executing UNIX processes communicating with the page reader via pipes. These may be pre-existing software tools imported from other research fields such as computational linguistics, information retrieval, and string matching. We illustrate the expressive power of the language in applications to English text using a spell-checker, Japanese text using character n-grams, and mixed Russian-English text using two lexicons with automatic context-switching.

1. Introduction

Contextual analysis is a subproblem of document image analysis, whose goal is to enforce application-specific constraints on the symbolic content of a document to resolve ambiguities remaining after segmentation and classification of symbols using shape and geometric context. Many OCR researchers believe that significant improvements in accuracy can be achieved through better contextual analysis; some feel that dramatic improvements are possible *only* in this way.

A principal objective of our research program is a versatile page reader which can be easily retargeted to new applications. We have reported some success in retargeting to a variety of symbol sets, typefaces, image degradations, page layouts, and symbolic output encodings. These have been achieved through various strategies: general-purpose algorithms, automatically trainable decision procedures, and table-driven code.

The contextual analysis stage of our page reader, however, has resisted

attempts to generalize, automate, or simplify it along these lines. In part, this is due to the large and growing number of relevant methods emerging from research into computational linguistics, information retrieval, string matching, numerical optimization, natural language processing, and cryptography, to mention only a few. We have come to believe that our page reader must be able to make use of existing software tools without substantial modifications. In the ideal case, it should be possible to run such tools unchanged, as separately executing processes.

Although contextual analysis is separable from image processing in principle, the two are often entangled in practice. For example, lexicon checkers and other data-driven contextual filters often rely on rank orderings provided by image classifiers. Also, the algorithms often must access the page-reader's internal data structures in specialized and complex ways. For example, a character *n*-gram checker needs to sequence through characters *n* at a time, perhaps coding word-breaks specially; the results must be attached to characters so that, later, alternative interpretations can be sorted and pruned. This is a particularly elementary example: its complexity convinces us that the expressive power of some form of programming language will be required in most cases.

Although low-level languages such as C (the language in which the page reader is written) are sufficiently expressive, they seem always to require a good deal of special code written with expert knowledge of the data structures. That is, they demand a custom software engineering project. The effort is so great that we can't run as many experiments as we want to. The practical consequences of this state of affairs is visible in the commercial OCR market, where it is not considered cost-effective to try to exploit special context except on problems occurring in very large batches.

Our attack on this problem involves the design of a high-level language specially tailored for contextual analysis which results in more concise, but no less expressive, programs than C. In particular, we want it to provide:

a. high-level abstractions of the data structure;
b. powerful sequencing primitives; and
c. a generic communications interface to separately executing processes, which is easily tailored by the user.

We have designed and implemented a language of this sort. In this paper, we will describe its present state of development, illustrate its capabilities on three contextual analysis problems, and speculate about future extensions.

2. System Overview

Automatic page readers must cope with many sources of variation, including symbol sets, typefaces, sizes of text, page layouts, linguistic contexts, imaging defects, and output encodings. Our strategy has been to organize the system so that each of these can be handled separately and independently, and the results combined arbitrarily. An outline of this architecture is given in [1]; a detailed description of its central data structure is given in [2]. Here, we summarize the major stages in order to establish the frame of reference for the stage that performs linguistic contextual analysis.

1. *geometric layout analysis*: connected components analysis, skew- and shear-correction, segmentation into text blocks, determination of

textline orientation, line-finding within blocks, and character-finding within lines. The underlying algorithms are described in [3]. Note that this analysis produces a *single* segmentation of the page image into blocks of textlines of individual symbols.

2. *polyfont symbol recognition*: classification of symbols by shape, inference of text size and baseline, segmentation of lines into words by spacing, and shape-directed resegmentation of characters within words to handle touching and broken characters. The result is a segmentation of the page image into blocks of textlines of words (if appropriate for the language), with potentially multiple segmentations of each word into symbols, with each symbol labeled with a short list of alternative interpretations.
3. *contextual analysis*: application of task or language models to resolve ambiguity implied by alternative segmentations and interpretations. This stage is the focus of the remainder of the paper.
4. *output encoding*: mapping the internal representation of the analysis into character codes (*e.g.* ASCII, Unicode, JIS, etc), optionally annotated by formatting directives (*e.g.* SGML or troff).

For the contextual analysis stage we have experimented principally with veto and sorting filters, which merely reorder or reject word interpretations within a list which is implied by the lattice of alternatives provided by symbol recognition. The list is run through each filter in turn: if a filter accepts no interpretation, the list is not modified (in case the filter is not relevant); but if any interpretation is accepted, it is promoted.

These methods are all *data-directed*: they merely select among alternatives generated by shape recognition. In some applications, *model-directed* analysis may be required for best results: these are able to supply missing alternatives by appeal to statistical, linguistic, or semantic models[4].

We would like to be in a position to exploit tools developed in other fields such as computational linguistics or information retrieval. Rather than trying to rip apart working software and interwining it with the internals of the OCR system, we instead leave the programs intact and attach them loosely to the page reader via UNIX pipes. The interaction between the page reader and these external filters is simple: the filter is expected to read from its standard input and to write a response to its standard output.

The syntax of the text passed to a filter is specified through ASCII tables stored in files constructed off-line. Each line of the file contains two fields, the first identifying the element of the document hierarchy (*e.g.* text line, character, or interpretation), the second giving an arbitrarily long byte sequence to be output. The encoding is specified using the C programming language syntax for constants, plus the U+XXXX style specified in The Unicode Standard, Version 1.0, Volume 1. For example, a translation to Unicode for the first 3 Japanese symbols in the national standard JIS X0208 would be specified as (everything beyond the second field is a comment):

```
BEGIN_PAGE   U+FEFF   # byte order
J1601        U+4E9C   # 亜
J1602        U+555E   # 唖
J1603        U+5A03   # 娃
```

We conventionally prefix Japanese OCR classes with 'J', and use the row/column identifier from the JIS

standard. Keywords such as `BEGIN_PAGE` and `END_WORD` allow delimiters to be marked, output as nodes in the hierarchy are traversed depth first.

The value returned by a filter is restricted to 7-bit ASCII strings whose semantics is coordinated in a controlling script. This will be described in the next section.

3. A Script Language for Contextual Analysis

We felt that what was needed was a simple, concise, high-level language through which we could construct and combine contextual analysis algorithms. Such a scripting language should provide:

1. access to the internal page-reader data structure, while hiding details;
2. functions for traversing the document hierarchy;
3. functions for generation, scoring, sorting, and pruning of alternatives implied by ambiguity;
4. built-in functions for (typographic) morphological analysis;
5. methods for customization for symbol usage or particulars of writing systems;
6. built-in functions for regular expression matching;
7. methods for utilizing pre-existing programs; and,
8. modern programming constructs such as variables, control flow, and procedures.

Rather than designing such a language from scratch, we have decided to build on an interpreted language called *tcl* (Tool Command Language) developed by John Ousterhout[5]. Tcl provides generic programming facilities and has a csh-like syntax with elements derived from the C and Lisp programming languages. The only data type in tcl is strings; type-casting is done automatically by the interpreter. The interpreter is embeddable in C programs and can be extended by writing C code and registering public functions.

We have extended the core features of tcl with about a dozen additional commands particular to our purposes; not all variations have been fully implemented. These will be described in the following subsections.

3.1 Traversal of Document Hierarchy

The contextual analysis subsystem typically receives as input a document made up of possibly multiple pages, each of which has been decomposed hierarchically into blocks of textlines of words of symbols. Contextual analysis algorithms need to traverse this hierarchy at various levels. A family of functions analogous to the tcl `foreach` construct are available in the language for this purpose. For example,

`foreachPage` *`var`* `within Document` *`code`*

causes the code body to be executed for each page of the input document. The variable *`var`* can be used to identify the page.

`foreachBlock` *`var`* `within` *`item code`*
`foreachTxtln` *`var`* `within` *`item code`*
`foreachWord` *`var`* `within` *`item code`*
`foreachChar` *`var`* `within` *`item code`*

Each of these constructs iterates over the corresponding elements of the page hierarchy (a page, block, textline, word, or character, respectively) owned by the element *`item`*; the variable *`var`* identifies the element for each iteration. The element identified by *`item`* must be deeper than the iteration element in the hierarchy (*e.g.* a Page is deeper than a Block, which is deeper than a Word). The elements are

visited in reading order, as established by up-stream stages of processing.

The normal `break` and `continue` statements work within the code blocks just as in the built-in tcl looping constructs.

3.2 Management of Alternatives

Ambiguity in the identity of characters on the page are caused both by alternative segmentations of the artwork into symbols, and by alternative interpretations given a fixed symbol. There are two families of functions which produce and iterate over lists of alternatives implied by this ambiguity:

```
foreachAltPage var from item \
     [thresh [limit]] code
foreachAltBlock var from item \
     [thresh [limit]] code
foreachAltTxtln var from item \
     [thresh [limit]] code
foreachAltWord var from item \
     [thresh [limit]] code
foreachAltChar var from item \
     [thresh [limit]] code
```

Each of these constructs iterates over a list of alternatives at the corresponding level of the hierarchy implied by ambiguities at or below the level of *item*. The variable *var* provides access to the alternative during each iteration. The element identified by *item* must be at the same level of the hierarchy as the construct. The optional parameters *thresh* and *limit* limit the number of alternatives considered: stop if the confidence value† falls below *thresh*× the best, or, after generating *limit* alternatives. The alternatives are produced descending on confidence value.

These functions essentially flatten the page hierarchy, producing a stream of items at the lower level while ignoring intermediate levels. Only the Char and Word version have been fully implemented (the combinatorics of producing alternatives at higher levels may make them intractable, or even if we could efficiently compute them, they may be essentially worthless for contextual processing).

For example, the following tcl code:

```
foreachPage P within Document {
    foreachWord W within $P {
        foreachAltWord AW from $W {
            ...
        }
    }
}
```

traverses through each of the words on a page, and generates a list of alternatives implied by segmentation/interpretation ambiguities. One of the English language documents used in the trials described in section 4.2 contained the image for "sensing", where several of the characters touched. Our shape-directed resegmentation algorithm produced 6 alternative segmentations for this word image; a few of the symbols had multiple classifications. The `foreachAltWord` function generated a total of 27 word alternatives:

```
sensing
sewing
wnsing
serving
wwing
 ...
seI&ng
wIning
```

The top-choice alternative is correct, but notice that two others are legal

† The confidence value is a real number between 0 and 1 and is computed as the n'th root of the product of the individual values (*e.g.* confidence of symbol classification).

English words. For most applications the lower alternatives would seem totally unreasonable since they contain embedded punctuation and mixed cases. Perhaps other alternatives such as `wnsing` and `wwing` are at an intermediate level of "believability".

The foreachAlt variety of functions produce alternatives of a *single* item — there is another family which produce a *sequence* of items:

```
foreachNgram N var from item \
      [thresh [limit]] code
```

This function produces n-grams of length *N* starting with (*i.e.* first in reading order) the element *item*. The given code body is executed for each n-gram produced; the variable *var* identifies the n-gram within the loop. The two optional arguments limit the production of n-grams, as above. Only the character n-gram function has been fully implemented.

Once an alternative has been analyzed, it can be marked and scored:

```
keep [-score s] alt
```

The alternative is marked to be kept and optionally assigned the score *s*. In the case of a word alternative, all interpretations making up the alternative are marked; for n-grams, only the base character is marked. Once a set of alternatives implied by an item have been generated and analyzed, it can be pruned by:

```
prune [-sort] item
```

Any character interpretation which did not take part in an alternative marked to be kept is pruned away. If this results in a segmentation with no character interpretations, the segmentation is pruned. Finally, the resulting list of segmentations and interpretations are optionally sorted by the maximum score in which they took part; otherwise, they remain sorted based on shape confidence score. If no alternatives were marked to be kept, the item is left unchanged.

3.3 Analysis of Words

Many writing systems delimit words with white space. However, when typeset natural language words may be catenated with punctuation, compounded, split across text lines, etc. We provide a built-in function to decompose an alternative, *i.e.* strings of symbol interpretations, into substrings:

```
alias -strength var alt
```

The function alias will be described in the next section. The result of the decomposition of *alt* is a list of tuples, placed in *var*: the first element is an ASCII label indicating type, the second element identifies the substring body of that type. The "strength" argument indicates the desired degree of decomposition and associated verification.

A strength of `weak` causes a left-to-right (in reading order) decomposition of the alternative into punctuation symbols, zero, one, or two alphanumeric bodies separated by infix characters, and closing punctuation (the punctuation bodies may be empty). A strength parameter of `strong` causes additional well-formedness checks, possibly causing labels on bodies to become more specialized: if the opening and closing punctuation are all prefix and suffix characters, respectively, they are labeled accordingly. If a body contains all alphabetic characters, it is labeled "alpha"; likewise, if it is all numeric it is labeled "numeric", otherwise, it is labeled "mixed". One special

check is also made: if the last body is alphabetic and is followed by a single suffix character which is a hyphen character which is at the end of a text-line, the body is labeled "alpha_end_hyphen".

More specialized and demanding checks can be applied to well-formed bodies. A strength of `alpha` causes alpha bodies to be labeled according to whether they contain all lower case, all upper case, or lower with an initial upper case. Numeric checks are also built-in, along with inspection of certain punctuation usage providing labeling of alternatives, such as with “end of sentence”.

The typographic morphology function returns true if the alternative could be completely parsed to the desired level, false otherwise.

Symbol type and usage is clearly writing system specific. We generalize these tests by specifying “character types” (analogous to the UNIX **`ctype`** functions) in ctype tables. Figure 1 shows an excerpt from the English-language ctype file used in the experiments of the next section. Naturally some symbols can be used in several contexts. Differences among symbol usage is subtle across some languages, say among the Western European languages[6]. Methods such as these should make it just as easy to move to radically different languages, such as Korean, where there are no case distinctions, no typographic compounding, etc.

3.4 Other Functions

The `declare` command is used to register an alias name and specify arguments for functions, either built-in or user provided:

```
declare alias function args...
```

The alias name can then be used in the later tcl code. There are two functions built into the system: 1) `TM`, which provides the typographic morphology as described in the previous section, and, 2) `REGEXP`, which provides regular expression matching. The only argument to the `TM` function is the `ctype` file. The regular expression function has two arguments, the name of the translation file containing the encodings to be performed on the string pattern before matching to the regular expressions, and the name of the file containing the regular expressions themselves. Our regular expression code has been generalized to the full UNICODE symbol set.

External programs are attached to

```
0      alnum,numeric,digit,exponent                              # digit '0'
A      alnum,alpha,upper
a      alnum,alpha,lower
col    punct,suffix,endphrase                                    # ':'
com    punct,suffix,endphrase,alnum,numeric,numsep               # ','
hy     punct,hyphen,prefix,infix,suffix,numeric,sign,exponent    # '-'
per    punct,alnum,numeric,point,suffix,endsent                  # '.'
pl     alnum,numeric,sign,exponent                               # '+'
qm     punct,suffix,endsent                                      # '?'
lp     punct,prefix                                              # '('
rp     punct,suffix                                              # ')'
```

Figure 1. Excerpts from English language `ctype` file. The meanings of the types are analogous to the UNIX ctype functions.

the system by specifying their file name in place of *function*. The only required argument is the translation file containing encodings to be performed on data sent to the filter. All other arguments on the declare line are passed to the program unchanged.

There is a group of functions, one per level of the document hierarchy, which declare variables, for example:

```
Word var1, var2,...
```

The given variables could then be used to read/write the internal information maintained about the word, such as its position on the page, the confidence value, ASCII labels, etc. The iteration variables on the `foreach` family of functions are automatically registered in this way. As the hierarchy is traversed, the values change automatically to reflect the node visited and its full context.

Finally, there are functions to look ahead in a `foreach` sequence:

```
next    item  n
unnext  item  n
```

The first command returns the *n* items following *item* in the sequence. The `unnext` command puts items back into the sequence.

4. Examples

We have used this language to implement simple contextual analysis algorithms in three different applications: reading Japanese newspaper articles, English-language technical reports, and a Russian to English dictionary. Each will be described in the following subsections.

4.1 Japanese Language

We purchased a corpus of roughly 40,000 articles from the Japanese newspaper Asahi Shinbun[7]. The total corpus size was just under 29 million symbols.

We selected 100 articles from this set, and printed them at 12 point type using the Mincho font style available in the Japanese TeX release (JTeX), then scanned the hardcopy using a Ricoh IS60 flatbed scanner at 300dpi resolution. Figure 2 shows a portion of one image. After digitization, each character is roughly 42 pixels high.

Separately, we constructed a classifier to recognize the same Mincho style, using the methods described in [8]. The training set included punctuation symbols, the Kana syllabaries, Arabic digits, Romaji (the Latin alphabet (Times Roman font style)), the Level 1 Kanji defined in JIS X0208, plus several dozen Kanji from Level 2. The total symbol set size was 3678 characters.

We then ran the 100 test page images through the geometric layout analysis and symbol recognition subsystems. On average, each symbol was labeled with 5.0 possible interpretations. The top-choice per-character accuracy based on shape recognition alone, measured on each page separately, ranged from 87.3% to 94.5%. The average top-choice accuracy across all 100 pages was 91.4%. (Those characters *not* in our training set are not counted in the accuracy figures.)

The remainder of the corpus was used to derive character uni-, bi-, and tri-gram statistics. There were a total of 5177 unique symbols in the corpus (the most frequent 100 characters accounted for 73% of the material). There were a total of 270K unique character bi-grams, or just over 1% of the possible number (5177×5177).

【’８５．１．１　朝刊　５頁　（全３２３９字）】

「いま一発の核爆発が欧州で起これば、軍の通信指揮系統がまっさきに破壊され、結局は全面核戦争に進まざるをえなくなる」「核戦争のあと黒い雨が降り、気温が低下し、地球は凍結する」「第三次大戦は、間違いなく人類最後の戦争となるはずだ」――。

五十九年度芸術祭大賞をとったＮＨＫ特集「核戦争後の地球」は、「核の冬」の恐ろしさをまざまざと映し出した。放送後七万件を超す電話がかけられ、回線はパンク状態になったという。

この番組は世界の主要国でも紹介、放映されて大きな反響をよび、年末にはフランス放送協会の「第一回ダルシー賞」を受けた。

Figure 2. Portion of Japanese test page printed at 12 point and digitized at 300 dpi; most kanji are 42 pixels on a side.

There were a total of just under 1.5M unique tri-grams, or about 1/1000th of a percent of the possible number.

We wanted to apply the bi- and tri-gram statistics in a purely binary way: that is, interpretations making up an n-gram which appear *at least once* in the corpus will overrule interpretations which did not appear. The bi-gram statistics were applied to the 100 pages using the code given in Figure 3.

The first command establishes the alias `jbigrams` for the external UNIX program `lexchk`. This program initializes by reading a lexicon, then loops forever reading a string of null-terminated bytes from its standard input, and writes a ’+’ character to its standard output if the string is in the lexicon or a ’–’ otherwise. The translation file `jisKuten.t` is a table mapping our internal class names to the JIS X0208 “kuten” encoding which provides a convenient 2-byte ASCII representation. Here is an excerpt:

```
declare jbigrams  bin/lexchk ./jisKuten.t ./binary.bigrams

foreachPage P within Document {
    foreachChar C within $P {
        foreachNgram 2 bi from $C 0.7 100 {
            if { [jbigrams $bi] == "+" } {
                keep $bi
                break
            }
        }
        prune $C
    }
}
```

Figure 3. Script to use Japanese binary bi-grams to prune character alternatives. `lexchk` **is an external program which performs lookups.**

```
J0401   "$!"
J0402   "$\""
J0403   "$#"
J0404   "$$"
J1601   "0!"
J1602   "0\""
J1603   "0#"
```

The file `binary.bigrams` contains the bi-grams extracted from the training corpus in the kuten encoding.

All of the foreach nesting simply allows us to iterate over all characters on the page. The `foreachNgram` loop iterates over the implied bi-grams. The square brackets invoke command substitution, causing a bi-gram to be sent, after translation, to the lexchk program. If the external program returns a '+', the bi-gram is kept. Since bi-grams are generated descending on shape confidence, we can stop looking at alternatives after the first hit. We prune the character, hopefully leaving only the interpretation taking part in a legal n-gram.

After filtering the 100 page images through this script, the top-choice accuracy ranged from 89.9% to 96.7%; the average accuracy was 94.0% for an average improvement of 29%.

Applying the tri-grams in the same way required trivial changes to the script: pass the file name containing the tri-grams to `lexchk`, and change the first argument to `foreachNgram` from 2 to 3. After filtering the 100 page images through this script, the top-choice accuracy ranged from 90.7% to 97.2%; the average accuracy was 94.8%.

In an attempt to measure the best improvement we could expect when applying binary n-grams in this way, we again constructed n-gram tables, but this time only over the 100 test pages. Over this material there were approximately 23K bi-grams and 51K tri-grams. Applying these "perfect" bi-grams to the 100 page test set, the top-choice accuracy ranged from 90.3% to 97.8%, with an average of 95.6%. Perfect tri-grams produced an accuracy range from 90.3% to 98.0%, with an average of 95.8%. Table 1 summarizes the experiments over the Japanese material.

Table 1. Top-choice accuracy over 100 Japanese-language test pages. Perfect n-grams refers to statistics taken over the test set.

Method	error rate	change
shape only	0.084	-
bi-grams	0.060	29%
tri-grams	0.053	37%
perfect bi-grams	0.050	41%
perfect tri-grams	0.043	48%

Further accuracy improvements through the use of character frequency information is possible. For example, the most frequent confusion overall, both before and after applying contextual analysis, was mistaking the Kana character た for the Kanji 左. Yet in our training corpus the Kana symbol was 290 times more likely to occur. Since our symbol recognition algorithms are based on Bayesian statistics, applying such uni-gram prior information is straight-forward.

4.2 English Language

The second set of experiments were conducted on the English-language database of technical reports distributed on CD-ROM by the University of Washington[9]. We randomly selected 100 pages from this database (the full list will be provided to any interested researcher). The body type size ranged from 9 to 12 point; all images were digitized at 300 dpi.

The construction of the symbol classifier used for these experiments is described in [8]; it was trained on over 100 font styles of the printable ASCII character set.

The 100 test images were put through the layout analysis and symbol recognition algorithms. On average, each symbol was labeled with 1.4 interpretations. The top-choice per-character accuracy based on shape recognition alone, measured on each page separately, ranged from 57.6% to 98.6%. The average top-choice accuracy was 92.0%.

In order to test the power of typographic morphology alone, the 100 pages were run through the contextual analysis algorithm show in Figure 4.

The `declare` command establishes the alias `tmorph` for the typographic morphology function, applying character types in the file `Ectype` (shown in Figure 1). The nested loops iterate over all words on the page, generating alternatives for each. Not shown is the routine `MIN` which records the minimum value seen in the variable `min`. We first attempt "strong" decomposition; if that fails, we fall back to checking "weak". If strong checks pass, we further analyze each body: if it is an alpha body, we verify it is all upper or lower case, or has an initial upper case.

```
declare     tmorph    TM      data/Ectype
foreachPage P within Document {
    foreachWord W within $P {
        foreachAltWord AW from $W 0.5 250 {
            set min 9
            if {[tmorph -strong tl $AW]} {
                foreach tup ${tl} {
                    set type [lindex ${tup} 0]
                    set body [lindex ${tup} 1]
                    switch $type {
                        alpha_end_hyphen -
                        alpha {
                            if {[tmorph -alpha tl $body]} { MIN 3 }\
                            else { MIN 2 } }
                        numeric {
                            if {[tmorph -numeric nl $body]} { MIN 3 }\
                            else { MIN 2 } }
                        mixed { MIN 1 }
                    }
                }
            }\
            else {
                if {[tmorph -weak tl $AW]} { MIN 1 }
            }
            if {$min < 9} {
                keep -score $min $AW
            }
        }
        prune -sort $W
    }
}
```

Figure 4. Script to perform "typographic morphology" on English words.

If the body is numeric, its syntax is checked for legality specially. If it is mixed, we do no additional checks.

Any reasonably well-formed alternative is kept and given the minimum score received over its bodies. After all alternatives have been examined, we prune the word alternatives and sort descending on their scores.

After filtering the 100 page images through this script, the top-choice accuracy ranged from 50.9% to 99.2%, with an average of 93.3%.

To apply a spell-checker the following code was added to the top of the script:

```
declare english bin/sprog ascii.t \
        spell/amspell
declare special REGEXP ascii.t \
        numforms
```

The external program `sprog` is essentially the UNIX spell program; `amspell` is a pre-compiled American English word list. The file `ascii.t` maps to printable ASCII (*e.g.* "per" to '.'). The file `numforms` contains regular expressions for common counting forms:

```
1st
[0-9]*[12-9]1st
[0-9]*2nd
[0-9]*3rd
[0-9]*[4-90]th
```

Additional code was added in the switch statement so that an alpha body is passed to the spell checker for verification. If the program indicated the body spelled, it was scored a 4. If not, "alpha" morphology checks were made; if that passed it was scored a 3, otherwise it scores a 2 (this quantifies the notion of `sensing` being more believable than `wnsing` which is more believable than `seI&ng`). In the case of a mixed-type body, it is checked against the regular expressions by the function `special`.

Figure 5 shows code added to the switch statement of Figure 4 to handle end-of-line hyphenations: since the alternative `body` ends with a "hyphen", get the next word in reading order and examine its alternatives. If it passes

```
alpha_end_hyphen {
    set NW [next $W 1]
    foreachAltWord AW2 from $NW 0.9 30 {
        if { [tm -alpha tl2 $AW2] } {
            set tup2 [lindex $tl2 0]
            set type2 [lindex $tup2 0]
            set body2 [lindex $tup2 1]
            if { $type2 == "lowcase" } {
                if { [english $body $body2] == "+" } {
                    MIN 4
                    keep -score 4 $AW2
                    break
                }
            }
        }
    }
}
```

Figure 5. Code added to switch statement of Figure 4 to handle end-of-line hyphenations. If the next word passes morphological checks, see if the catenated bodies spell. If so, keep both alternatives.

morphology checks, pass the catenated bodies to the spell checker. If the combined alternative spells, assign both bodies a score of 4.

Passing the 100 test images through the final script, the top-choice accuracy ranged from 56.7% to 100%, with an average accuracy of 94.7%.

Lastly, we constructed a "perfect" lexicon by extracting only those words which appeared in the ground-truth for the 100 test images. The tcl script was only modified to use `lexchk`, with this lexicon, in place of the general UNIX spell checker. The accuracy ranged from 51.8% to 100%, with an average accuracy of 95.3%. Table 2 summarizes the experiments over the English-language material.

Table 2. Top-choice accuracy over 100 English-language test pages. Perfect lexicon refers to word list extracted from ground-truth.

Method	error rate	change
shape only	0.080	-
shape+morphology	0.070	18%
shape+morphology+ spell checking	0.053	34%
shape+morphology+ perfect lexicon	0.047	41%

4.3 Russian-English Dictionary

For the final set of experiments, we scanned 30 pages of a Russian-English dictionary[10]. Figure 6 shows a portion of one of the columns from one page. We constructed a single classifier to recognize *both* alphabets, along with punctuation, digits, etc, for a total of 304 classes.

We had in hand an English language spell-checker, and a (rather small) Russian language word list.

леворазрешимый

леворазрешимый, *adj.*, left-solvable
левосторонний, *adj.*, left-side, left; **левосторонний смежный класс,** left co-set
левоуничтожающий, *adj.*, left-annihilating
левоупорядоченный, *adj.*, left-ordered
левый, *adj.*, left, left-hand; **левый идеал,** left ideal; **левая сторона,** left-hand side
легализировать, *v.*, legalize
легально, *adv.*, legally
легальность, *f.*, legality
легальный, *adj.*, legal
легкий, *adj.*, easy, light; **легко,** it is easy
легко, *adv.*, easily
легкость, *f.*, ease, readiness; **с большой легкостью,** very easily

Figure 6. Column of Russian to English dictionary page digitized at 300 dpi.

These could be combined, or alternatives could be looked up in both, but we decided instead to use script consistency to determine the appropriate context.

Figure 7 gives the top of the tcl script and main loop. The translation table `check.t` maps symbols in the Latin alphabet to an 'L', symbols in the Cyrillic alphabet to a 'C', and punctuation and digits to a '?'. `checkscript` is a 10-line C program which performs the script consistency check. It reads a string and writes an 'L' if the string does not contain a 'C', it writes a 'C' if the string contains a 'C' but not an 'L', or a '?' otherwise. The routines `latin_script` and `cyrillic_script` implement the same algorithm as the script of the previous section: typographic morphology appropriate for the language, along with lexicon checks. They return a 1 if the alternative is in the lexicon.

The lexicon `peculiar.lex` contains a list of words peculiar to this

```
declare checkscript bin/checkscript data/check.t
declare english     bin/sprog       data/ascii.t data/amspell
declare peculiar    bin/lexchk      data/ascii.t data/peculiar.lex
declare russian     bin/lexchk      data/cyr.t   data/russian.words
declare emorph      TM                           data/Ectype
declare rmorph      TM                           data/Rctype

foreachPage P within Document {
    foreachWord W within $P {
        foreachAltWord AW from $W {
            switch [checkscript $AW] {
                L { set res [latin_script $AW $W] }
                C { set res [cyrillic_script $AW $W] }
                ? { set res 0 }
            }
            if {$res == 1} { break }
        }
        prune -sort $W
    }
}
```

Figure 7. Initialization and main loop of script which provides context switching between Russian and English. `checkscript` **performs a simple script consistency check.**

book, like adj, adv, n, and v. The routine `latin_script` first consults the general spell-checker; failing that, it looks in this special lexicon.

Figure 8 shows the top-choice result for the image in Figure 6. The left side is after shape-recognition alone, the right after applying the algorithm outlined above. Considerable improvements can be seen, although quantifiable measurements were not possible since ground-truth was not available.

леворазрешимь%

леворазрешимь%, adj., 1eft-solvable
левосторонний, adj., left-side, left; левосторонний сметнь% хласс, left co-set
левоуничтомающиЙ, adj., left-annihilat-Ug
левоупорЯдоченнь%, adj., left-ordered
левь%, ad)., left, left-hand, левыЙ и)1еал, left ideal; леваЯ сторона, 1eft-hand side
легализировать, v,, legalize
легально, adv., legal1y
легальность, f, legality
легальнь%, adj., legal
легхиii, adj., easy, light; лervto, it is easy
лег'Но, adr., easily
легКость, f, ease, readiness; с %m>шоЙ легхостью, very easily

леворазрешимый

леворазрешимый, adj., left-solvable
левосторонний, adj., left-side, left; левосторонний смежный класс, left so-set
левоуничтожающий, adj., left-annihilat-Ug
левоупорЯдоченный, adj., left-ordered
левый, ad)., left, left-hand, левый идеал, left ideal; леваЯ сторона, left-hand side
легализировать, v,, legalize
легально, adv., legally
легальность, f, legality
легальный, adj., legal
легкий, adj., easy, light; легко, it is easy
легКо, adv., easily
легкость, f, ease, readiness; с большоЙ легностью, very easily

Figure 8. OCR output given image in Figure 6. Left side is result using shape recognition alone, right side is after contextual analysis of Figure 7. (output edited to maintain indentation).

One difficult case is illustrated on the 6th line: the simple script described above cannot handle the case of a compound word where the second body is split across lines. If it could, we would have corrected the trailing "ing", which was the second alternative.

5. Discussion

Although the language is designed to hide details of our data structure, it is constrained by the fact that the data structure describes the physical (geometric), rather than the logical (functional), document hierarchy. In the future we hope to extend it to cope with logical parts of the document such as phrases, sentences, paragraphs, sections, and articles.

For an interpreted programming language, `tcl` is more efficient than we expected. The rapid prototyping supported by `tcl` allows us quickly to test the usefulness of a function; then, if greater speed is needed, it can be rewritten in C locally without affecting the rest of the code. In this way, we have the option of selectively "raising the level" of the language to trade off flexibility for speed. Admittedly, this strategy is likely to appeal only to programmers already expert in the page reader internals.

The computation of some contextual analysis algorithms can be enormous. In the Russian-English example, the long Russian words sometimes produced thousands of alternatives. Of course, no programming language can protect against combinatorial explosion, but it can be expected to provide data structures rich enough for natural implementations of efficient algorithms where they exist. For example, the `foreachAlt` family of functions produce alternatives on the fly, as needed, rather than all up front. We also don't want to force the programmer to explicitly free lists, yet many are produced as a result of the nested loops. We are not sure that our language offers a sufficient set of data structures and iteration primitives in this sense.

There are facilities we know are missing that may prove important. We don't suppress duplicates: *e.g.* alternative segmentations can result in multiple identical interpretations. We don't consider alternative segmentations when generating n-grams: making use of positional information within n-grams is easy for a single segmentation but hard for alternative segmentations. There is the general issue of implicit *versus* explicit representations, which crops up in many forms, which we have not fully mastered. For example, it is easy to store explicit word alternatives, but recording explicit sequences of character n-grams within a text line is much more difficult. We hope to experiment soon with word-level models of English in order to resolve hard cases such as "In the distributed (sensing|serving) application...."

Our contextual analysis language is a work in progress. The experience of applying it to hard cases has forced several rewrites, and we expect more. We are generally happy with its conciseness, but we may not yet have the most natural syntax. The success of the trials so far make us eager to experiment with more complex constraints on these natural languages, and the same constraints on a wider variety of languages.

6. References

[1] H. S. Baird, *Anatomy of a Versatile Page Reader,* IEEE Proceedings, Special Issue on OCR, July, 1992.

[2] H. S. Baird and D. J. Ittner, *Data Structures for Page Readers,* IAPR Workshop on Document Analysis Systems, Kaiserslautern, Germany, 1994.

[3] D. J. Ittner and H. S. Baird, *Language-Free Layout Analysis,* Proceedings, 2nd Int'l Conf. on Document Analysis and Recognition, Tsukuba Science City, Japan, 1993.

[4] H. S. Baird and K. Thompson, *Reading Chess,* IEEE Trans. PAMI, Vol. PAMI-12, No. 6, June 1990, pp. 552-559.

[5] J. K. Ousterhout, *Tcl and the Tk Toolkit,* Addison-Wesley Publishing Company.

[6] H. S. Baird, D. Gilbert, and D. J. Ittner, *A Family of European Page Readers,* Proc., 12th Int'l Conf. on Pattern Recognition, Jerusalem, Israel, 1994.

[7] Asahi Shinbun newspaper, corpus of articles: "Heavenly Voice and Human Word," 1985 to 1991, published by Nichi Gai Associates.

[8] H. S. Baird and R. Fossey, *A 100-Font Classifier,* Proc., 1st Int'l Conf. on Document Analysis and Recognition, St.-Malo, France, 1991.

[9] I. T. Phillips, S. Chen, and R. M. Haralick, *CD-ROM Document Database Standard,* Proc., 2nd Int'l Conf. on Document Analysis and Recognition, Tsukuba Science City, Japan, 1993.

[10] A. J. Lohwater, *Russian-English Dictionary of the Mathematical Sciences*, American Mathematical Society Publishing, Providence, Rhode Island.