A New Masoretic "Spell Checker" or A Fast, Practical Method For Checking the Accentual Structure and Integrity Of Tiberian-Pointed Biblical Texts

Richard L. Goerwitz III

June 22, 2004

Abstract

The orthography of biblical Hebrew manuscripts and scholarly printed editions encompasses not only traditional alphanumeric symbols, but also cantillation marks or "accents" that indicate how the text should properly be chanted. These marks are difficult to typeset, and even the best scholarly Bible editions contain various cantillation errors. Cantillation marks follow a grammar of their own that largely mirrors the syntactic structure of the sentences in which they occur. Although this grammar is not strictly context free, it is possible to construct a grammar very close to it that is not only context-free, but also LR-parsable, and that can therefore serve as the basis for real-world computer-based automata that can efficiently locate accentual errors in scholarly Hebrew Bible editions.

1 Introduction

Any Hebraist who has worked with a critical (or really any printed) edition of the Bible knows that such editions introduce more errors into the cantillation system (i.e., into the "accents") than into any other facet of the text. Even the latest edition of *Biblia Hebraica Stuttgartensia* (Kittel, Elliger, Rudolph, et al. 1997 [BHS]), a generally accurate work, contains a significant number of accentual anomalies—on the order of one every fifteen pages. Some of these anomalies reflect features inherent in the base manuscript (Leningrad B19a). More often they reflect, e.g., deviations from the original manuscript in the printed edition. Because most of these anomalies go unnoted in BHS, they are generally to be construed as errors either in the editorial process or, more typically, in the printed text itself.

In the past, eliminating such errors has proven impractical, both because of the quantity of material, and because of the limited number of proofreaders

available who can efficiently locate and correct them. If human error could be eliminated from this aspect of the production process, the potential benefits to publishers and their readers would be enormous. Not only would publishers be able to attain levels of accuracy and consistency not known since the time of the scribes who wrote the texts, but they would also be able to reduce substantially the resources consumed by the usual editing and proofreading cycles. This, in turn, would ultimately reduce the overall cost of providing such works to the public.

The purpose of this study is to describe such a system, i.e., a system for automatically parsing, locating, and analyzing errors in the cantillation marks of modern scholarly Hebrew Bible texts. In the first half of the study, I will discuss this system in general terms, explaining how it is possible to construct it, and why I have taken the approach that I have. In the second half, I will describe a fast, practical, working implementation of it—a proof of concept, if you will.

Note that although this study focuses on the Tiberian cantillation system (specifically as described in Israel Yeivin's introduction to the Tiberian Masorah, Yeivin 1980), and to BHS, the methods outlined here may be generalized and adapted to other cantillation systems and texts.[1]

2 Cantillation and Context-Free Grammars

In his (1990) study, James Price claims that the Tiberian cantillation system (i.e., the cantillation system used in what most consider to be the best Medieval biblical manuscripts) can be formalized as a self-contained, computer-implementable, context-free accentual grammar. Context-free grammars are sets of rewrite rules, like the classic sentence structure rule, S → NP VP ("a sentence consists of a noun phrase followed by a verb phrase"), which are often used to describe the syntax of a language. Although Price's study does much to systematize and elucidate the structure of the Tiberian cantillation system, he never actually offers a full set of such rules; that is, he never offers a full context-free accentual grammar (Goerwitz 1994). Why? Because doing so, at least in the way he envisions the task, is simply not possible.

To illustrate why it is not possible to create such a grammar, let us examine the case of one particularly difficult accent, *revia*. *Revia* is a disjunctive accent, somewhat like a comma in most modern Latin-based orthographies. *Revia* divides clauses marked by *ṭifḥa*, *zaqef*, and *segolta*—which denote even stronger breaks (somewhat like a semicolon). *Revia* can be repeated as many times in a verse as necessary. *Revia*, however, cannot follow itself too closely. If fewer than three words intervene between one *revia* and the next, the last *revia* turns into *pašṭa* (or, depending on syllable structure, into the variant form *yetiv*), unless this *pašṭa* would land within two words of the next *tevir* or *zarqa*, in which case

[1] Yeivin's work is more concise and less theoretical than Wickes (1970); hence far more useful here.

a *tevir* or *zarqa* (depending what sort of disjunctive clause we are in) is used in its place.[2]

Note how important proximity is to the workings of these replacement rules. Although it is possible to conceive of a context-free grammar that handles all of the various possible combinations, the problem is one of elegance. Context free grammars do not represent concepts like "near" and "far" or "within x words" well. And an accentual grammar proper cannot represent word and syllable structures at all. Although it is sometimes possible to construct brute force grammars that simply list attested combinations, such grammars quickly become unwieldy and unnatural, and impossible to write.

This, then, is why, although Price's formalisms go a long way towards characterizing Tiberian accentual structures, Price wisely refrains from trying to offer a full accentual grammar: The Tiberian accents are simply too complex and multi-leveled a phenomenon to be captured elegantly or completely as a self-contained set of context-free rules.

3 Context-Free Grammars and Tractability

As it turns out, the problem of capturing the Tiberian accents as a self-contained, context-free grammar is not only one of elegance and theoretical completeness, but also one of practical analysis and implementation. Even if we managed to construct a grammar that accounted for *revia* replacement and other such esoterica, and even if we could incorporate extra-accentual features such as word and syllable structures, the fact remains that there would still be no efficient, reliable, easily implementable method for programming a computer to process this grammar and to convert it into a parser.

The most powerful class of grammars that computers can deal with easily are those that can be converted into a type of *deterministic pushdown automaton* known as an *LR parser*.[3] Grammars that fall into this category convert to small, very fast parsers that provide timely, efficient error recovery. Although it is often possible to handle grammars that fall outside this range by preprocessing the input with a so-called *lexical analyzer* (discussed in section 6), such an approach quickly becomes impractical for the sorts of phenomena we typically see in natural languages, ambiguity being one particularly salient case in point (Tofte 1990).[4]

Because the Tiberian cantillation system contains many ambiguities (e.g., one cannot always tell a true *pašṭa* clause from a converted *revia* clause), the Tiberian Hebrew accentual system cannot be reduced, even with help from a lexical analyzer, to an LR-parsable grammar. It thus defies straightforward, computer-based analysis.

[2] See Yeivin (1980) §226, 230, 234. There are in fact additional (!) proximity and combinatory restrictions listed there.

[3] For more extensive discussions of these terms, see an introductory compiler textbook such as Aho, Sethi, and Ullman (1986).

[4] GLR parsers process ambiguous input, but can only do so efficiently if that input has relatively few ambiguities (Tomita 1985).

Despite apparent difficulties, though, the basic goal of machine-based analysis of the Tiberian accents is not altogether outside the realm of possibility. To realize it, however, we must define our goal as simply being able to recognize errors in the accentuation of modern biblical editions. To achieve this goal we do not need to construct a full, theoretically elegant parsing system. We can, instead, settle for a simpler, slightly less accurate parser that brings us back into the realm of computational tractability, and allows us immediate access to a wide assortment of well-developed, reliable software tools and methods that will actually get the job done.

In a theoretical sense, such a move is "cheating." In practical terms, however, we are making precisely those concessions that enable us to develop a system that does what scholars and publishers really need it to do, namely to offer fast, practical help analyzing and correcting the cantillation marks of modern Hebrew Bible texts that are based on Tiberian-pointed biblical manuscripts.

4 "Cheating"

Instances where "cheating" is most critical, i.e., where we must misrepresent the grammar in order to obtain a working system, consist mainly in proximity/position restrictions, such as the *revia* → *pašṭa*, *zarqa*, or *tevir* rule discussed above (section 2). Note also the case of *segolta*, which cannot be used if a *zaqef* or an *atnaḥ* has already appeared in a given verse. Often such phenomena can be re-cast as LR-parsable rules. But, as noted in connection with context-free grammars above, accomplishing this (if it is possible at all) comes only at the expense of verbosity and unnaturalness. And it makes the resulting grammar extremely difficult to write. *Segolta*'s distribution, for example, can be handled by creating one special *atnaḥ* and two special *silluq*, clauses, i.e., an *atnaḥ* clause whose first major divider is *segolta*, and a *silluq* clause whose first major divider is either a *segolta*, or an *atnaḥ* clause with a *segolta*. Such rules, though, are ugly; and they will never capture generalizations about processes that involve syllable structure, or that boil down to questions of how musical patterns interact with the text's syntactic or semantic components (e.g., when should we use *segolta* in place of, say, *atnaḥ*, *zaqef*, or *zaqef gadol*?).

So instead of trying to capture generalizations like this, we should simply give up and "cheat"—our goal being to reduce the complexity of our grammar to the point where it can be processed using simple, widely-available parser-generation software that can produce a real, working system.

Such cheating does not prevent us from dealing with verses containing accents like *segolta*, or *pašṭa* and *revia*. All it does is force us to accept a slightly lower standard of accuracy when validating them.

For example, to account for the distribution of *pašṭa* and *revia* in a simple, tractable way, all we need to do is ignore the *revia* → *pašṭa* conversion rule, accepting as correct any sequence of *revia*s followed by *pašṭa*(s).

Accepting as correct any sequence of *revia*s followed by *pašṭa*(s) means, of course, that our grammar now accepts as valid a few constructs that it really

should mark as invalid. Acceptance of invalid constructs like this, however, does not cause difficulties within an actual, working parsing/error-detection system. The reason for this is that the vast majority of errors introduced by the editing and typesetting processes consist of omissions, mindless mis-keyings. Only rarely are editors or typesetters creative enough in their mistakes to introduce ones that happen to correspond exactly to a concession or "cheat" that we have allowed into our parser/error-detector's grammar.[5]

5 The Base Text

Having outlined the general theory on which an accentual parsing and error-detection system must operate, it is now possible to talk about the practical details of the particular implementation outlined here—details such as what program modules make it up, how these modules interoperate, and what software development tools were used to create them. Before discussing the system's implementation, however, let us consider briefly the data that this system will be working on. Let us consider, in other words, the nature and structure of the texts that the system is supposed to be parsing and checking.

In order for automated parsing and checking to work, we need texts set up in such a way that the computer can recognize the various accents. A good example of such a setup is the machine-readable *Biblia Hebraica Stuttgartensia* (BHS) edition distributed by the Center for Computer Analysis of Texts (CCAT) at the University of Pennsylvania. This edition—developed originally by the University of Michigan under grants from the Packard Humanities Institute and the University of Michigan Computing Center—utilizes a series of two-digit codes to represent the Tiberian accents.[6] For example, the two-digit code 73 stands for *ṭifḥa*; 80 stands for *zaqef*; 92 stands for *atnaḥ*. There are several ambiguous codes, such as 75 (which is either *silluq* or *meṯeg*), but these are all fairly easy to resolve (e.g., 75 is *silluq* if it comes just before *sof pasuq*). The beauty of a simple, clean system like the CCAT's is that it is extremely easy for the computer to process and manipulate. Such a system therefore provides an ideal basis for a computer-based accentual parser/error-detector.

Consider, by way of contrast, the antithesis to the CCAT BHS (hereafter eBHS) texts: A proprietary coding system designed to work with a specific brand of typesetting or word-processing software. Such a system would provide no motivation for distinguishing between, say, *mahpaḵ* and *yetiv*, or between *azla* and *pašṭa*. Why? Because the same graphic symbols are used in both cases. Remember that typesetting and word-processing codes exist mainly just to tell software where to print what symbols. And because *mahpaḵ* and *yetiv* are the same symbol, as also are *azla* and *pašṭa*, there is no need in this context to use different codes to represent them. Unfortunately, in contexts such as automated

[5] See section 9 for a discussion of where such mistakes are most likely to fool the working proof-of-concept system.

[6] An online version of the CCAT BHS codebook is available, as of December 2003, at the following URL: `http://www.wts.edu/hebrew/whmcodemanual.html`.

error checking for format conversions, distinctions such as this are extremely important, because without them vital structural information is obscured or entirely lost.

It might be added that proprietary typesetting or word-processing systems have a limited lifetime—usually the same as that of the software they are used with. This includes *Extensible Markup Language* (XML)[7] based coding systems, which (despite XML's current fashionableness) are not necessarily tractable or usable outside the context of the proprietary software used to create them.

Unlike proprietary coding schemes, schemes like the one used in the eBHS text serve as efficient information repositories. They do not contain superfluous information. They convert readily into other formats. And they can be readily accessed, maintained, and corrected. Such schemes, therefore, are what we should be using as the basis for error checking. They may be stored in a platform-neutral state and then converted, as needed, into this or that (or perhaps several) typesetters' native formats. In essence, the arguments for using a format like that of eBHS are the same as those for using structural or *content-based* (rather than presentation-based) markup schemes in general—what XML and its predecessor, SGML, were, in theory, intended to be.[8]

6 The Implementation

Having discussed both the theory on which a practical accentual parser/checker must operate and the data formats it prefers, it is now possible to discuss details of the proof-of-concept implementation offered here.

Stated briefly, the parser/checker offered here (which I call simply **Accents**) consists of two basic modules: 1) a lexical analyzer, and 2) a parser. The first module, the lexical analyzer, translates accentual codes into a form that the next module, the parser, can utilize. The parser then restructures these accents into a simple human-readable parse tree (see fig. 1), and flags any errors it detects. The parser itself is fairly abstract, and deals only with generic representations of the accents it is processing. Although the parser is, in theory, agnostic about what lexical analyzer is used, and what coding system the lexical analyzer operates on, in actual fact I have written only one lexical analyzer for it. This one is tailored specifically for the eBHS texts mentioned in the preceding section (page 5).

[7] XML is actually a metalanguage for defining the syntax of markup (e.g., "tags" like <body></body>) in documents. Markup/tags give structure to what would otherwise be plain character data. XML markup syntax is formally defined using *document type definitions* (DTD) or *schemas*. The XML specification itself is controlled by the World Wide Web Consortium (http://www.w3.org/XML/). When people talk about XML they are typically talking about document instances that conform to one or another XML DTD or schema. XML itself is not a markup language or document format, and it defines the actual semantics of markup in only limited ways. Contrary to popular belief, there is nothing intrinsic to XML that prevents one from using it in obtuse or proprietary fashions.

[8] See Coombs, Renear, and DeRose (1987) for further, if early, discussion of the distinction between markup intended for presentation, like "color" or "boldface" codes, and markup intended to delineate real underlying textual structures like paragraphs and sections.

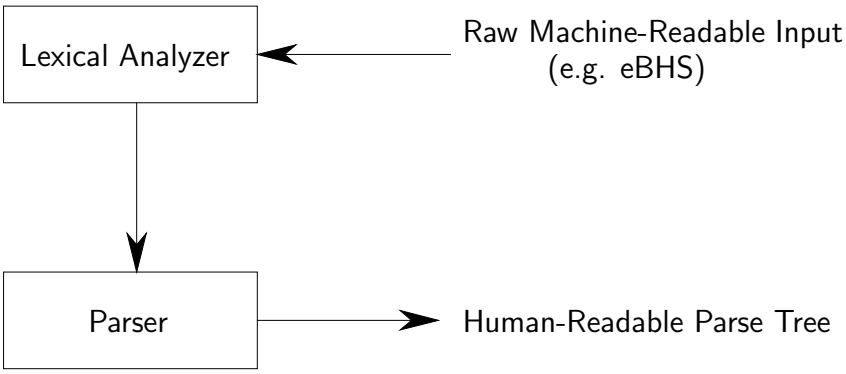

Figure 1: Overview of parser/checker

One key point about the Accents program is that it is small and conceptually very simple—simple enough to be implemented using off-the-shelf tools like C (ANSI), YACC, and Lex.[9] C has been around since the 70s, and is ubiquitous. YACC, a simple language for generating LR parsers, is a standard component of most stock Unix and Unix-like systems. Lex, a tool for creating modules that break data streams down into so-called "tokens," is also a standard Unix utility. Versions of YACC and Lex exist for many different operating systems besides Unix. They are well understood, reliable, readily available tools.[10] Because of their reliability and ready availability ANSI C, YACC, and Lex were natural choices as the implementation languages for Accents.

Without wanting to turn this paper into a tutorial on YACC and Lex (which, like all software, will seem outdated and quaint within at most a few decades of their creation), let me nevertheless pause for just a moment to review what modules written for these tools look like and how they work. YACC grammars, it turns out, are written in a notation linguists will easily grasp; and Lex files make extensive use of a system of commonly used pattern-matching language called *regular expressions* (see, e.g., Friedl (2002)). YACC and Lex therefore serve as useful tools for explaining how Accents is put together and how it functions.

The YACC portion of Accents consists of a series of rules of the form:

LHS : RHS

where LHS ("left-hand side") represents a node in a given accentual parse tree having RHS ("right-hand side") as its child, or children. When several rules

[9] ANSI C is the version of C defined by the American National Standards Institute Committee X3J11. Draft standards began in the late 80s. The latest official standard is ANSI/ISO 9899-1990.

[10] Lex and YACC were developed at Bell Laboratories in the 1970s—YACC by Stephen C. Johnson, and Lex by M. E. Lesk and E. Schmidt. Both were shipped as standard Unix utilities as far back as version 7 (1978). See Institute of Electrical and Electronics Engineers (IEEE) Std1003.2 (POSIX Shell and Utilities) for the most recent applicable standard.

have a common LHS component, they may be grouped together, in which case a simple slash is written instead of the repeated LHS. By convention, "terminal" symbols, i.e., symbols that form the leaves of the parse tree, are written in capital letters:

```
silluq-clause   : silluq-phrase
                | tifcha-clause silluq-clause
                | tevir-clause silluq-clause
                | zaqef-clause silluq-clause
                | atnach-clause silluq-clause
silluq-phrase   : SILLUQ
                | MEREKA SILLUQ
                etc.
```

The above YACC input rules may be read, in English, as follows:

1. A `silluq-clause` consists of either
   (a) a `silluq-phrase`
   (b) a `tifcha-clause` then a `silluq-clause`
   (c) a `tevir-clause` then a `silluq-clause`
   (d) a `zaqef-clause` then a `silluq-clause`, or
   (e) an `atnach-clause` followed by a `silluq-clause`.
2. A `silluq-phrase` consists either of
   (a) `SILLUQ`, or
   (b) `MEREKA` then `SILLUQ`

   ...

The actual rules found in the Accents source code[11] are, in reality, considerably more complex than the ones given above. The Accents source code, for example, characterizes the cantillation mark *ṭifḥa* in such a way that it cannot occur after *zaqef* within a *silluq* clause. What it does, in other words, is to prevent rule 1c above from applying to the output of rule 1d. Doing this is not terribly difficult. But to explain fully how it is done would require a rather lengthy digression. It is enough here merely to note that same basic principles illustrated above apply to the full grammar.

What YACC does is to take the entire set of syntax rules that make up the Accents grammar, and turn these into a working parser. This parser may then be used to process the Bible verse-by-verse, building linear sequences of accents up into two-dimensional trees (e.g. fig. 2)—or else reporting any failures to do so, presumably due to errors in the text.

In order for the parser to do its work, something has to convert the accent "codes" present in the actual Hebrew text into symbols the parser can recognize

[11] The term *source code* refers to human-readable computer instructions, which must be converted or *compiled* into something a computer can execute, i.e., into a computer program. See page 15 below for information on obtaining the Accents source code.

like `MEREKA` and `SILLUQ`. As noted above, this conversion is handled by the lexical analyzer.

The lexical analyzer is generated via Lex from a set of directives or "rules" that map specific patterns in a machine-readable input stream to tokens (terminal symbols) that a parser can understand. The following Lex rule, for example, tells Lex to send an `ATNACH` token to the parser whenever it encounters the characters '9' then '2' on its input stream:

```
92          { return ATNACH; }
```

All Lex rules have this same general form. Basically, the material at the left-hand margin lists the character sequence to look for. The remainder of the line contains computer code written in the C programming language, which tells the lexical analyzer what to do when it finds that character sequence. Here are several more examples:

```
01          { return SEGOLTA;   }
65{TEXT}05 { return SHALSHELET;}
80          { return ZAQEF;     }
85          { return ZAQEFGADOL;}
81          { return REVIA;     }
```

The TEXT string above is a macro that expands to an expression that matches non-numeric characters. That macro is defined elsewhere in the Lex input file.

In a few cases the Lex rules become considerably more elaborate than what we see above, as, for example, when the rules must distinguish *munaḥ+paseq* combinations that are simply that from ones that are actually the elusive (and graphically identical) accent *legarmeh*. The Lex rules also handle verse and/or so-called Betacode delimiters (used by older eBHS texts to mark books, chapters, and verses; they are now obsolete).

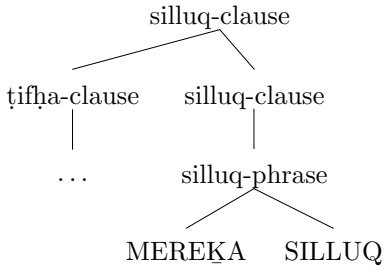

Figure 2: Sample parse tree

```
0 silluq_clause
  1 atnach_clause
    2 tifcha_phrase
      tifcha
    2 atnach_phrase
      munach atnach
  1 silluq_clause
    2 tifcha_phrase
      mereka tifcha
    2 silluq_phrase
      mereka silluq
```

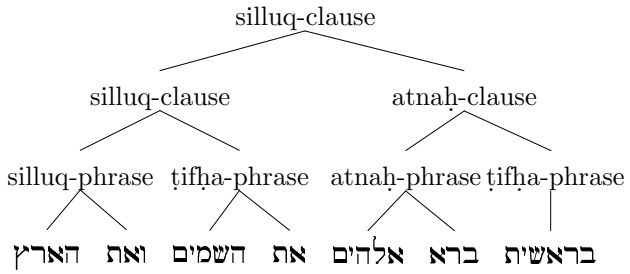

Figure 3: Accents' structural interpretation of Gen 1:1

7 Running the Accents Program

Accents is provided as a source-code distribution. This means that people who want to use it must compile it into executable form. I.e., they must themselves convert the C, YACC, and Lex files into a program their computer can actually run. Doing this is not difficult for those who have worked in C, or its daughter language, C++. On Unix and Unix-like systems compiling is likely to require nothing more than a run of the included `configure` script and an invocation of the standard program-building utility, `make`. (Those who are not accustomed to compiling executables from source should contact their information technology support staff and avail themselves of a local Unix programmer or systems administrator.)

Once set up, Accents can be set to work, from a command-line interface (in Unix terms, a "shell"; in Microsoft Windows terms a "command window"), directly on eBHS-format texts—which must, incidentally, be in Unix LF (as opposed to DOS CR-LF) format. Accents simply reads the text from the "standard input" and sends a list of verses it has processed to the screen or "standard output," flagging errors as it finds them. Alternatively, it can take its input from one or more files specified on the command line. Systems with no concept of standard input or a command line will not support Accents, at least as it is currently configured. Those not familiar with command-line interfaces should again consult their local information technology support staff.

Under Unix (or any Unix-like operating system), for example, Accents would normally be invoked as follows:

`accents -p` < *name-of-your-eBHS-file*

where *name-of-your-eBHS-file* is the full pathname of the file where your eBHS text resides, and where `-p` is a command-line switch that tells Accents to print trees for the verses it parses.

The trees Accents outputs to the computer screen are not nearly so elaborate as the one depicted above in fig. 2. Rather, Accents outputs its parse trees

using a simple, indented, text-only notation. The digits at the left-hand side of each line of output indicate the degree of nesting. Literal accent names such as *ṭifḥa*, *munaḥ*, and *atnaḥ* appear at the innermost clausal levels, with no preceding digit. For an example of this notation, see fig. 3, which shows a piece of Accents' actual output (left), and depicts graphically (on the right) how this output relates to an actual Hebrew verse (in this case, Gen 1:1). Read the right side of this figure in reverse order, following the natural order of the Hebrew words at the bottom.

When invoked with the `-p` command-line option, Accents reports errors as part of the accentual parse trees it produces (see section 8 for a full discussion of this feature and its potential usefulness to editors, proofreaders, and typesetters). If no `-p` command-line switch is provided, Accents merely lists parsing errors as it finds them, and emits *book chapter:verse* references for each verse it has successfully processed, e.g.:

```
Gen 1:1
Gen 1:2
Gen 1:3
Gen 1:4
Gen 1:5
...
Exod 4:8
Exod 4:9
accents warning 7 (yyparse): error encountered in Exodus 4:10
Exod 4:10
Exod 4:11
...
```

Although on some computer systems error output doesn't end up quite where one might expect it to, notice of an error will normally precede reference to the verse that caused the error—at least when running in this mode. If `-e` is supplied on the command line, error messages are suppressed, and only those verses that contain errors are listed on the screen. In other words, if you type

`accents -e` < *name-of-your-eBHS-file*

you will see, instead of the above output, only references to verses containing errors, namely—

```
...
Exod 4:10
...
```

Note that the `-e` option will also work with the `-p` option. Using these two options together tells Accents to display trees for only those verses that contain accentual errors.

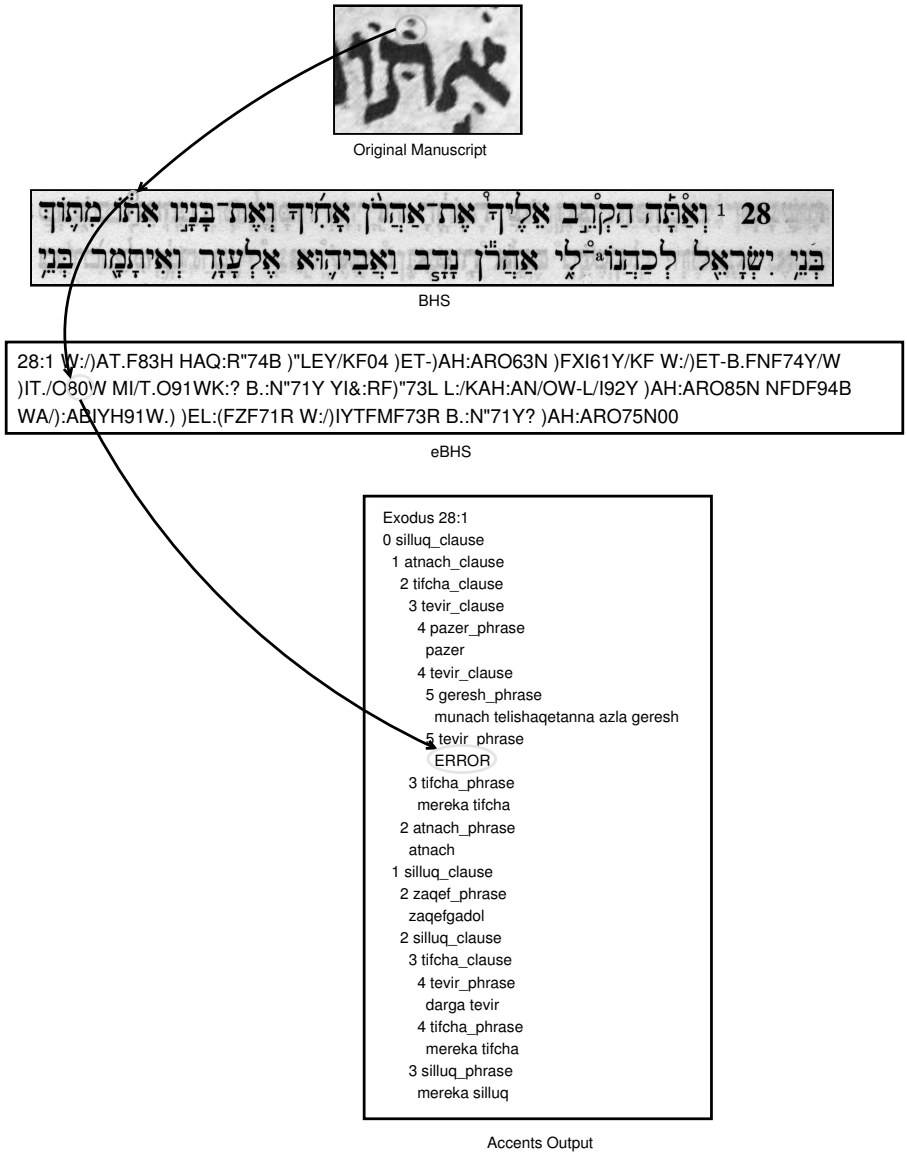

Figure 4: Sample error from *Biblia Hebraica Stuttgartensia* (Karl Elliger and Wilhelm Rudolph eds., Fifth Revised Edition, edited by Adrian Schenker, ©1977 and 1997 Deutsche Bibelgesellschaft, Stuttgart. Used by permission.)

8 Full Example

To illustrate the utility of Accents for editors, proofreaders, and typesetters, let me offer a full, real-life example of how it manages to ferret out, with relative

ease, a subtle error that made its way into both BHS and eBHS.

Above the second letter of the ninth word of Exodus 28:1 in the Leningrad Hebrew Bible manuscript (B19a) there appears to be a stray mark—a dot that sits over, and very slightly to the right of, the somewhat more darkly drawn accent *revia*. The trouble with this stray dot is that it doesn't look like a stray dot unless one's attention is called directly to it as a possible error. As a result, the dot makes the *revia* it sits atop (which looks something like a period, .) look very much as though it were, instead, the lower member of the colon-like accent, *zaqef*.

The seeming reasonableness of this reading, visually speaking, is evident in BHS, where the editor of Exodus, G. Qwell, without any comment at all, transcribes the accent as *zaqef*. As one might expect, Qwell's oversight also appears in eBHS, where it has gone probably undetected until now. This type of error, which would be virtually impossible to eliminate from traditional, hand-proofread biblical editions, is caught easily by Accents, which, when applied to Exod 28:1 (invoked with the `-p` option), produces the error report shown at the bottom of figure 4.

Notice that in that report, Accents manifests some confusion about where exactly the error occurs (it omits a preceding *munaḥ* and marks the segment containing the error as a *tevir* clause). As noted above, this sort of confusion is not uncommon. Accents must often throw out a few tokens or terminal symbols before it can reset itself well enough to continue parsing. Usually a quick look at the manuscript, printed edition, or electronic text in question (any of which may prove to be the error's source) is all that is required to pin down where error that Accents reports actually lies.

9 Known Shortcomings (Bugs)

As noted above (section 4), there are a few accentual errors that Accents will not always catch. These are listed in the comments to the YACC parser code, as found in the Accents source code distribution. Salient examples include 1) *pašṭa* for what should be *gereš* and 2) some cases of *yetiv* for *mahpaḵ*, and the reverse. These are very easy errors to make, from an underlying data-encoding standpoint (particularly when using naive optical character recognition to input character codes). But fortunately they are not common in eBHS, and they are not visually difficult to spot in print-outs (the slant of the mark and/or its position will be completely wrong).

Occasionally the accentuation of a verse is also just too bizarre for Accents to handle. In Exodus 20, for example, the doubly accented Ten Commandments cause Accents to spew an amusing series of error messages. Those who attempt to run Accents themselves should simply ignore these messages. So also for the second version in Deuteronomy 5. Although Accents handles some doubly accented verses quite elegantly (e.g., Gen 35:22), such verses are more the exception than the rule.

Sophisticated users wanting a deeper understanding of these problems, or

of any others that surface, are invited to try out the `-d` option—which causes Accents to emit profuse diagnostic messages that can often be of help in determining what it is "thinking." Under Unix or Linux, for example, one might type

```
accents -d < text 2>&1 | less
```

It is possible to separate out only those diagnostic messages that seem particularly relevant for a particular problem by piping Accents' output through a standard Unix filter program, such as `egrep`. For example, to obtain a list of superfluous or unrecognized accents encountered during parsing, one might type

```
accents -d < text 2>&1 | egrep 'Unrecog' | more
```

Note, however, that with the `-d` option, Accents runs considerably more slowly than it does without. (It runs fastest when invoked with just the `-e` option.) Running Accents with the `-e` option (with or without the `-d` option) may cast light onto subtle parsing issues, i.e., onto why Accents is flagging an error as such.

Another issue worth mention is the whole concept of errors - a term that mirrors closely the simplistic way in which Accents views Hebrew accents. In Accents' view a given accentual sequence either passes or it gets flagged as bad. There are no gray areas. In real life, however, there are gray areas. Accentual sequences may violate purported accentual "rules" without rendering the text erroneous in a strict sense. Another way to view the system described here might therefore be as an *anomaly* detector—i.e., a system for locating sequences of accents that merit attention, either because they reflect errors, or else because they constitute interesting deviations from the norm. I have generally used the term *error* here simply because it is a more natural one for information scientists, and because, in the great majority of cases, the Accents program seems in fact to be finding errors (rather than debatable "anomalies") in the text.

One final issue worth mentioning here is that Accents can currently process only the twenty-one prose books of the Hebrew Bible. The three poetic books, Psalms, Proverbs, and Job, in other words, have yet to be integrated into the distribution. There were three reasons for this omission. First of all, I had limited resources at my disposal to devote to the project. Secondly, my knowledge of the poetic cantillation system was limited, and I simply did not feel as comfortable designing for these accents as I did for the prose ones. Thirdly, I felt it made sense to start with the prose accents, because the prose books constituted about ninety percent of the Hebrew Bible.

If time and resources permit, or if some new source of funding appears on the horizon, I will extend the system to cover the poetic books. In the meantime, if some other scholar would like to make the required modifications, he or she would have my blessing. See page 15 below for directions on how to obtain the source code.

10 Concluding Remarks

The central intellectual question that this paper addresses is whether the fairly complex Tiberian Hebrew accentual system can be re-cast as a simple, computationally tractable "grammar." Theoretically, the answer to this question is "no." Practically, however, the answer is "yes." *If* we are willing to extend our grammar so that it accepts not only valid accentual constructs but also a few (unlikely) erroneous constructs as well, we can, in fact, reduce that grammar to a simple, computationally very tractable form.

The fact that our grammar can be reduced to a simple, computationally tractable form has enormous practical consequences. As noted at the outset of this paper, it means, for one thing, that it is possible now for scholars and publishers to produce accentually correct biblical editions. It also means that they can easily construct tools for asking questions of the data like

1. What range of accentual clause types does accent X occur in, in manuscript or edition Y?
2. How does manuscript/edition Y's use of accent X differ from that of, say, manuscript Z?
3. Where in Z did the scribe make the most accentual errors, and what sorts of errors did he tend to make?

As proof of this concept, i.e., as proof that we can construct a practical computer-based tool that can analyze biblical Hebrew cantillation marks, this paper has introduced *Accents*. Accents is a simple tool for analyzing and validating the accentuation of machine-readable Tiberian texts. It is obviously not the last word in accentual validation. But it is practical, stable and very fast. Most importantly, however, it offers *prima facie* evidence that, in fact, it truly is possible to construct practical, working automata, based on an almost-correct accentual grammar that can efficiently and reliably locate errors in machine-readable Hebrew Bible texts.

Anyone wanting the source code for Accents should contact the author at *richard@goerwitz.com*. At least through the end of 2005, the source code will also be available at `http://www.goerwitz.com/software/accents/accents-1.1.4.tar.gz`.

References

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers, Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley PubCo.

Coombs, James H., Renear, Allen H., and DeRose, Steven J. (1987). "Markup Systems and the Future of Scholary Text Processing". *Communications of the ACM* 30 (11): 933-947 (1987).

Friedl, Jeffrey E. F. Friedl (2002). *Mastering Regular Expressions* (2th ed.). Sebastopol, CA: O'Reilly and Associates.

Goerwitz, R. L. (1994). Review of Price (1990). *Journal of the American Oriental Society 114*(1), 276–277.

Kittel, R., K. Elliger, W. Rudolph, et al. (Eds.) (1997). *Biblia Hebraica Stuttgartensia* (5th ed.). Stuttgart: Deutsche Bibelgesellschaft.

Price, J. D. (1990). *The Syntax of Masoretic Accents in the Hebrew Bible*. Number 27 in Studies in the Bible and early Christianity. Lewiston: Edwin Mellen Press.

Tofte, M. (1990). *Compiler Generators—What They Can Do, What They Might Do, And What They Will Probably Never Do*. Berlin and New York: Springer-Verlag.

Tomita, M. (1985). *Efficient Parsing for Natural Language: A Fast Algorithm For Practical Systems*. *Kluwer International Series in Engineering and Computer Science* volume SECS 8. Boston: Kluwer Academic Publishers.

Yeivin, Israel. (1980). *Introduction to the Tiberian Masorah*. Edited and translated by E. J. Revell. SBL Masoretic Studies 5. Scholars Press.

Wickes, William. (1970). *Two Treatises on the Accentuation of the Old Testament*. Reprint of 1881 edition, with prolegomenon by Aron Dotan. New York: Ktav.