HUHU: THE HEBREW UNIVERSITY HEBREW UNDERSTANDER

SERGEI NIRENBURG* and YOSEF BEN ASHER
The Hebrew University of Jerusalem, Givat-Ram, Jerusalem, Israel

(Received 18 November 1983)

Abstract—This paper describes a system of morphological and syntactic parsing of the Hebrew language. It contains an extensive morphological analyzer and an augmented transition network-based syntactic parser. The system has been written in the YLISP dialect of Lisp. A parallel effort for English (different grammars that use the same parsing software) has also been developed.

Natural language processing Morphological and syntactic analysis Parsing Augmented transition networks The Hebrew language

1. INTRODUCTION

1.1 The Objective

The Hebrew University Hebrew Understander is a system of syntactic analysis of Hebrew based on an augmented transition network (ATN) grammar for this language. It obtains as input Hebrew sentences and produces syntactic structure trees quite similar to the standard deep structure representations. An example of a parse tree and its English translation can be found in Appendix A.

1.2 System Architecture

HUHU consists of the following components:

(2) The morphological analyser MORPHPARSE.
(3) A morphological grammar (affix rules) for Hebrew.
(4) An ATN grammar.
(5) The syntactic parser.

The process of parsing starts with the search for the morphological and syntactic information on every word in the input sentence. This process is twofold: first of all it is necessary to determine the dictionary form (citation form, entry head) for the above words and, second, to enter the dictionary and find the corresponding entry. Next, the information peculiar to the entire lexeme is added to the (word form) information gathered during the process of stripping the stem of affixes, and the findings for the given sentence are recorded in the appropriately organized storage structure (the “small” dictionary).

As a matter of fact, the syntactic parser does not deal with an unprocessed input sentence at all. Its input is in the form of the small dictionary for the given input sentence. The parser operates by matching input against a Hebrew grammar presented in the augmented transition network formalism. In this grammar our knowledge of the language is encoded as permissions or prohibitions for the parser to match a certain component of the input sentence with one of the known correct “prefabricated” patterns.

The result of the parser’s work may be either the message that the sentence is incorrect or the output of all the “pares”, syntactic analysis trees corresponding to the given sentence within the given grammar.

Let us have a closer look at the components of the system in turn. In Section 2 we describe the design of each of the system’s components as enumerated above. Section 2 will be devoted to the

*Present address: Department of Computer Science, Colgate University, Hamilton, NY 13346, U.S.A.
2. THE COMPONENTS

2.1 The Hebrew Dictionary

HUHU distinguishes between the following syntactic classes of words:

- noun
- verb
- adjective
- adverb
- pronoun
- preposition
- demonstrative
- numeral_1
- numeral_10
- definite article
- conjunction_1
- conjunction_2
- preadverb
- question_word
- negation_word.

Not all of the above categories are familiar "parts of speech". The two types of numerals were introduced to facilitate processing of multi-word numerals in the ATN. The numeral_1 category contains the numerals from zero to nineteen, while the numeral_10 category contains the numerals \{twenty, thirty, ..., ninety\}. The demonstratives were elevated to the status of an independent class from their usual place as a subset of pronouns because in Hebrew they appear in an altogether different type of sentence positions (demonstratives behave like adjectives, while personal pronouns behave like nouns).

In Hebrew adverbs usually follow the verbs they depend on. The rule, however, has exceptions. These exceptions could be dismissed as colloquial usage. But we decided to treat them by introducing a new category of preadverbs for this purpose. We have also separated question words from pronouns. Finally, we decided to distinguish between the conjunctions that join the parts of a compound sentence (e.g. "or") and those connecting subordinate clauses (e.g. "which", when used as a conjunction).

The dictionary entry consists of the entry head and the information part. The information part contains a number of attribute values, varying from one (the "part of speech" marker itself) to four. These are values that are valid for the whole lexeme, that is for the word in any of its forms. The information about the form of the word is retrieved by the morphological analyser, appended to the dictionary information and output into the small dictionary.

In addition to the category marker (NOUN), the dictionary information on the category of nouns includes: TYPE (values: common, proper), GENDER (masculine, feminine) and TYPE_OF_PLURAL_SUFFIX (-im, -ot, -iot, -aim). The entries for verbs feature TRANSITIVITY (transitive, intransitive) and SENTENCE_OBJECT [the mark about whether the given verb can take sentence objects, including infinitives] (+scomp, −scomp). Pronouns feature PERSON (first, second, third), GENDER (masculine, feminine) and NUMBER (singular, plural). The information on demonstratives consists of GENDER (masculine, feminine) and NUMBER (singular, plural). Information on NUMERAL_1 includes GENDER (masculine, feminine).

2.2 Morphological Analysis

In order to supply the syntactic parsing with the information on the category and form of the words in the input sentence, we must perform morphological analysis. This stage is recognized as necessary by many workers in the parsing field, but its importance within a comprehensive natural language parsing system is considered minor. This consensus can be explained by the simple fact that a majority of natural language parsing systems in existence are built for English, and form formation is a department in which the English language cannot boast a great variety of means. (Thus, the explication of the English affixation facilities in Peterson [1] covers but three pages.) For
predominantly non-analytical languages the problem of morphological analysis is not altogether trivial. Hebrew features a number of additional complications, since its morphology regularly encroaches upon what is traditionally considered the domain of syntax: for instance, such phrases as “and when I met you” can be compressed into a single word.

Thus, morphological analysis of Hebrew warrants a closer attention than that of English. What results would we like to obtain at this stage? For each word we would like to list all the lexemes it can be a form of, together with all the relevant information on the form it is in. If an input word is an amalgam of lexemes then the information on all the participating lexemes is to be found.

Let us first dwell on the case of “non-compound” words, those referring to a single lexeme. In addition to the data from the main dictionary, the following information must appear in the small dictionary:

—For NOUNS:

(1) NUMBER (singular, plural) [the dual forms of nouns are treated as separate lexemes when relating to a single object, cf. MICHNASAIM, “trousers”, but not otherwise (DAKATAIM, “two minutes” will not appear in the main dictionary)].
(2) SMICHUT (+, −) [this is the peculiar Hebrew way of expressing “of”-chains (cf. “the wife of the brother of the manager of the plant…”): in such chains all the nouns except the last one are in the SMICHUT form]. We would use the English notation _of_ for this feature.
(3) POSSESSIVE (possessive_(_person_); possessive_(_gender_); possessive_(_number_))

In Hebrew the semantic relation of possession is regularly expressed through affixation, not with the help of pronouns. A noun’s suffix may contain information on the number, person and gender of the person or object to which the notion referred to by this noun belongs.

—For VERBS:

(1) BINYAN (pa’al, pi’el, hi fil, hif’al, ni fi’al, hitpa’el) [this is a network of verb forms (in modern language many such forms are really different words) built similarly, according to well-specified rules, on the same root; a number of inferences can be made knowing that a verb is in the form of a certain BINYAN, for example, the hi fil verbs are always transitive; the hitpa’el ones, reflexive].
(2) TENSE (past, present, future)
(3) GENDER (masculine, feminine)
(4) NUMBER (singular, plural)
(5) PERSON (first, second, third).

The principal means of form formation in Hebrew is affixation. Prefixes and suffixes prevail, though infixes can also be found. Our task is to take a word from the text and, by means of coordinated scissoring of prefixes and suffixes, to find a lexeme in the dictionary of which the given word is a form. The technique we suggest for performing the morphological analysis of Hebrew is NOT structured particularly for this language. It can be employed for morphological analysis of other languages (the HUHU project, as a matter of fact, has an English parser built on the basis of the same software)—given the relevant dictionary and affix rules.

The technique for Hebrew consists of two consecutive stages: at first we deal with affix processing, that is, determining the grammatical form of the input word; next we describe the processing of prefixed prepositions, conjunctions and articles.

In addition to the input word, the morphological analyser requires two more data types:

(1) the MAIN DICTIONARY (see above, Section 2.1) which consists of a line for each entry and contain grammatical information pertinent to all the forms of entry head (e.g. past, present and future are [tense] forms of a verb, whereas transitivity vs. intransitivity is a feature not of a verb FORM but of a verb as an entry head and is preserved in all the latter’s forms).
(2) a set of AFFIX RULES which contains information on correct affixation of words in a certain language, in our case, Hebrew. This is actually the morphological grammar for this language used to “recognize” the affixation pattern of input words.
2.2.1 Affix rules

In this section we will, for the sake of simplicity, deal with input words that do not have "appended" prepositions or conjunctions. An input word can then be represented as a string of the form:

\[ w[i] = \{p[1]...p[n]\} r[1]...r[u] \{s[1]...s[k]\}, \]

where \( p[i] \) is a letter from the set of letters comprising the prefix of a word, \( r[i] \) is a letter from the stem (the "root") of the word, and \( s[i] \), a letter from its suffix; \{ \} enclose optional elements.

The program returns the following sequence:

\[ W[k] = \{Wo[i], i = 1,...,m\} \quad (1) \]

where \( Wo \) is the entry for the word in the "small dictionary" (which consists of the entry header and information, partly taken from the MAIN DICTIONARY and partly supplied by the program MORPHPARSE itself). The "small dictionary" is implemented as an array (see 3.1.4).

It follows from (1) that one input word in Hebrew can lead to a number of output words. This corresponds to the possibility that an input word can be a form of more than one lexeme (for example, the string "DBR" in Hebrew can be a noun or an imperative form of a verb).

The recognition of the possibilities is carried out with the help of the morphological grammar presented in the form of affix rules. An affix rule is actually a conglomerate of rules united under a single header which describes the morphological pattern of the word. The header's format is:

\[ \langle \text{prefix} \rangle \ast \langle \text{suffix} \rangle, \]

where \( \ast \) standards for the "residue", or candidate stem. In the simplest case the stem becomes the key for dictionary search, while the affixes determine the form in which the word appears. This is recorded in the corresponding line of the affix rule in question. The two parts (stem and affixes) of the information are appended and output to a special structure called "small dictionary", the dictionary for the current sentence.

The main problem is to detect what are the correct boundaries of the stem and the affixes in a given word taken in the context of the input sentence. This can be known only if the input is vocalized or if the results of syntactic analysis of the sentence are available. We desired to be able to work with unvocalized texts, because this is the regular way for a Hebrew text to appear. Secondly, syntactic analysis was our main objective which was supposed to be assisted by the morphological analysis, not to provide inputs into the latter. The communication between the morphological and syntactic stages can be achieved in a multiprocessor approach to parsing (cf. e.g. the Hearsay II system—Erman et al. [2]; cf. also the multiprocessor approach in Lozinskii and Nirenburg [3]). As a result, we cannot make a final decision on the boundaries of the stem.

The obvious solution then is to output all the candidate lexemes with all information about their forms into the "small dictionary". And this is precisely what is done by the MORPHPARSE program.

The program produces, first of all, a list of all the affix rules which are to be checked for the input word and then goes on to stripping the affixes, looking for the residue in the main dictionary and comparing the information on the part of speech contained both in the dictionary and the affix rules (affixes and their combinations almost always determine whether a word is a form of, say a noun or a verb). If the corresponding entry is found and the initial check is passed, then the rule "has worked". But if the search fails, the affix rule may suggest the possible changes in the stem itself that may have been done to put the lexeme in the given form (actually, these are infixes). Using this information the program changes the stem and goes on to another round of searching. The infix information is recorded in the corresponding affix rules in the form of test—action pairs, where actions are special string manipulation functions (see below).

2.2.2 Dealing with composite words

The following words can be prefixed to other words in Hebrew:

- \( -B \) "in"
- \( -K \) "as"
To make things even more complicated, combinations of the above are possible. Yet another complication is that all the above letters can also be elements of the stem. So, the detection of these "admixtures" should be done rather carefully. The input word now acquires a more general format of

\[ (\text{admixture}_{\text{letters}}) (\text{prefix}) \ast (\text{suffix}) \]

An attempt is made to gradually separate the admixtures from the word, process them and the rest of the word separately (though, several compatibility checks are run even at this stage) and then gather the results in the "small dictionary" entry for the corresponding word.

Knowledge about the legal combinations of the one-letter words in Hebrew is summarized in the system in the following transition network:

![Transition network diagram](image)

The "admixture" letters are enclosed in square brackets. Angular brackets enclose compatibility tests, for example, if a word starts with the letter "hei" then, if the "residue" after "hei" was removed is a verb stem, the "hei" belongs to the syntactic category of Question words; if the residue is a noun, then the "hei" is a Determiner; if neither, the letter "hei" is part of the stem and our attempts at understanding it as a one-letter word failed.

Every state in the network can be exited on a "jump" arc, that is, without reading input. This means that the string of admixture letters may start at any node. Every state in the network can also be final; The network displays all the legitimate combinations of the prefixed words. So, we can see, for example, that only the letters "hei" and "vav" can appear twice in the string, and at that, not one after the other. The expected category of the word under processing was marked in angular brackets in the diagram above.

The introduction of such a network is not mandatory. It was possible to do without it, by extending the regular affix rules.

The "small dictionary" array produced during this stage of analysis is the interface between the morphological and the syntactic component of the system. The syntactic analyser does not deal
with the initial sentence at all. It uses the above array and a syntactic grammar for Hebrew as the input data structures. Let us at this stage describe the format of the grammar.

2.3 The ATN Grammar for Hebrew

The augmented transition networks (ATNs) are probably the most widely used means for writing grammars for use in automatic syntactic recognition of natural language sentences. Alternative proposals have been developed; and the ATN approach has been criticized for a number of deficiencies, including its non-deterministic character, clumsiness in dealing with languages featuring free word order, etc. But still, the ATNs are the simplest and the most general tool in the field.

The theory and applications of augmented transition network grammars are amply described in literature (see especially Woods [4], Bates [5] and Winograd [6]). So, we shall restrict ourselves to a very concise description of the approach and a discussion of our own implementation.

An ATN is a transition network consisting of states connected by arcs. There are two types of “marked” states in every network: a single starting state and a number of final states, the traversal of a network starts in its starting state, and the sentence is found to be correct if when the input string ends, the parser finds itself in one of the final states.

A specification of the ATN language we used follows:

\[
\begin{align*}
\text{ATN} & \rightarrow ((\text{registers}) (\text{state}) (\text{state}*)) \\
\text{registers} & \rightarrow (\text{register}) (\text{register}*) \\
\text{register} & \rightarrow (\text{any identifier}) \\
\text{state} & \rightarrow (\text{state_name}) (\text{arc}*) \\
\text{state_name} & \rightarrow (\text{any identifier}) \\
\text{arc} & \rightarrow (\text{cat} (\text{category_name}) (\text{test}) (\text{action})* \\
& \quad (\text{to} (\text{state_name}))) \\
& \quad (\text{push} (\text{state_name}) (\text{test}) (\text{action})* \\
& \quad (\text{to} (\text{state_name}))) \\
& \quad (\text{mem} (\text{word_list}) (\text{test}) (\text{action})* \\
& \quad (\text{to} (\text{state_name}))) \\
& \quad (\text{wrd} (\text{word}) (\text{test}) (\text{action})* \\
& \quad (\text{to} (\text{state_name}))) \\
& \quad (\text{jump} (\text{state_name}) (\text{test}) (\text{action})* \\
& \quad (\text{pop} (\text{form}) (\text{test}))) \\
\text{category_name} & \rightarrow \text{N} | \text{V} | \text{ADJ} | \text{ADV} | \text{DET} | \text{QUANT} | \text{PREP} | \\
& \quad \text{CONJ1} | \text{CONJ2} | \text{NUM1} | \text{NUM2} | \text{PRON} | \text{DEM} ... \\
\text{test} & \rightarrow (\text{any LISP function}) \\
\text{action} & \rightarrow (\text{setr} (\text{register}) (\text{form})) \\
& \quad (\text{sendr} (\text{register}) (\text{form})) \\
& \quad (\text{liftr} (\text{register}) (\text{form})) \\
& \quad (\text{addr} (\text{register}) (\text{form})) \\
& \quad (\text{addl} (\text{register}) (\text{form})) \\
\text{form} & \rightarrow $ \\
& \quad (\text{getr} (\text{register})) \\
& \quad (\text{buildq} (\text{pattern}) (\text{registers})) \\
& \quad (\text{append} (\text{form}) (\text{form})) \\
& \quad (\text{quote} (\text{any structure})) \\
& \quad (\text{getf} (\text{feature}) (\text{register})) \\
& \quad (\text{look_ahead} (\text{integer}) (\text{test}))) \\
& \quad (\text{what_token} (\text{register}) (\text{feature})) \\
\text{feature} & \rightarrow (\text{any value of syntactic category, e.g. PLURAL})
\end{align*}
\]

Most of the components of the above ATN are standard. We chose to eliminate the non-terminal category \text{term_act} and elevate “jump” to the arc status (this is also quite a standard procedure). Among the non-standard elements: the “look_ahead” predicate, which returns TRUE if the information about the word standing \text{integer} places to the right of the current word \text{integer} are also allowed and mean movement to the left) contains a feature we are looking for; the “what_token” function helps retrieve facts from the “small dictionary” about a word, provided the order in which this information appears in the list is strictly determined. We tried not to
introduce such an order—because of the natural desire to make the data structure as flexible as possible—but still this function can be found useful.

The current version of the Hebrew ATN can be found in Appendix B.

2.4 The Syntactic Analyser

The technical discussion of the program that carries out the parsing will be deferred till Section 3. The parser takes as input the "small dictionary" for an input sentence (supplied by the MORPHPARSE program) and an augmented transition network in which the knowledge about the syntactic structure of the language (in our case, Hebrew) is encoded. It produces syntactic structure trees similar to those of generative grammar deep structure trees. If a sentence has multiple syntactic readings, the program will list all of them.

The tree (2) of Appendix A is obtained by scanning the ATN grammar using the small dictionary for the sentence (1) of Appendix A. The content of the small dictionary for this sentence is as follows:

\[
\begin{array}{ll}
\text{w11} & : \text{(BAT: noun common -im plural smichut (_of_) )} \\
\text{w12} & : \text{(BAT: noun common -ot feminine singular poss_first poss_singular)} \\
\text{w13} & : \text{(BAT: noun common -ot feminine plural poss_first poss_singular)} \\
\text{w21} & : \text{(SEFER: noun common -im masculine singular definite)} \\
\text{w22} & : \text{(SAPAR: noun common -im masculine singular definite)} \\
\text{w31} & : \text{(IR: noun common -im feminine singular preposition in)} \\
\text{w41} & : \text{(SHAVAT: verb intrans present plural third masculine active kal)} \\
\text{w42} & : \text{(SHOVET: n common -im masculine plural)} \\
\text{w51} & : \text{(BEMESHEH: preposition)} \\
\text{w61} & : \text{(KIM'AT: preadverb)} \\
\text{w71} & : \text{(HAMISHA: numeral_1 masculine)} \\
\text{w81} & : \text{(SHAVUOT: noun common -ot masculine plural)}
\end{array}
\]

What the parser is supposed to do is to try to parse all the combinations of "small dictionary" entries and in the process decide what are the appropriate readings of the input words. Thus, in the above sentence, the parser is supposed to "try" 3*2*1*2*1*1*1*1*1 = 12 possible combinations, which in this case would produce 2 different parses (one would include the word SEFER, "book" and the other, SAPAR, "barber").

The necessity to scan all the combinations of the word senses requires a traversal procedure. We used backtracking. Our approach to this problem is described below, in Section 3. For the time being, let us suppose that the ATN is supplied with the correct syntactic meaning of the input word.

The parser enters the Hebrew ATN in the start state of its "highest level" subnetwork with the first input word (w11) in a special register 'S'. We leave the actual scan as an exercise to an interested reader.

3. THE PROGRAMS

3.1 The Program MORPHPARSE

This program performs morphological analysis of Hebrew words and records results in a "small dictionary". As mentioned above, this program makes use of two data structures: the main dictionary and the affix rules. We shall discuss the program routine by routine. First, the routines taking care of the dictionary will be presented, followed by the routines for dealing with affix rules.

3.1.1 Ar_load

This routine loads the main dictionary into the memory. The main dictionary entry is a line whose name is the citation form of a word after which the category values for this word appear. In the program environment the dictionary entries are understood to be atoms whose name is the citation form and whose value is the list of categories. The YLISP system we are using has an eight-bit field for an atom. To mark an atom as a dictionary entry we set the first bit to 1.

There are two modes of access to the entries. First, every entry may be accessed as a LISP atom (by name) and, second, all atoms are stored in an array called "ar_dic" and therefore can be accessed by array index. The storage in the array is inexpensive, since for every atom we store no
more than a single pointer. The above structures allow both direct and sequential access to the data. Since the YLISP system holds all the declared atoms in the system hash table, the access time is minimum. Of course, the price is large storage volume, since the main dictionary is stored in the core memory in toto. In research environment this did not pose major problems. Using secondary storage for AI databases is a separate research issue that we did not address in this study.

The function ar_load obtains two parameters: the name of the array and the file in which the main dictionary is stored. The function loads the main dictionary into the core memory. It also checks whether the current entry name was already loaded. If this is the case (this accounts for syntactic homographs), the routine "synonym_entry" is called which gives the current word a unique name by concatenating a serial number to the original word.

Should the memory space allocation for the array ar_dic become insufficient, ar_load extends it with the step of 500.

AR_LOAD makes use of the following functions:

1. "synonym_entry" produces the serial number (see above) and in its turn calls the function "new_entry" which performs the construction of the new entry name. (The function "new_entry", in its turn, makes use of two auxiliary functions itod and dtoi which transform an integer into a character and a character to an integer, respectively.)

2. "is_in_dic" checks whether a given entry is already present in the dictionary.

3. "put_rule" declares that the entry is in dictionary by setting its first bit to 1. This function is used not only by "ar_dic", but also in writing affix rules. A sampling of affix rules for Hebrew can be found in Appendix C.

4. "read_sent" is a utility function for reading a line of input. Used extensively throughout the system. Reads until end of file or end of line or ".

3.1.2 Get_dic_forms

This package takes care of morphological analysis of words in an input sentence exclusively on the basis of the affix rules.

The set of AFFIX RULES contains information on correct affixation of words in Hebrew. An affix rule is a named list. The name is an atom of the type

\[ A_R[i] = \text{prefix*suffix}. \]

where A_R stands for "affix rule". The value of this atom is a list whose task is to modify the input word in accordance with the given affix. The structure of this list is as follows (we show that this list is the result of the LISP evaluation of the atom A_R):

\[ (\text{gval A_R[i]} \rightarrow ((A[1])(A[2])...(A[k]))) \]

A[j]'s are specific rules operating on the word (i.e. on what remained after the prefix and/or suffix have been removed by A_R[j]). We deal here with various kinds of exceptions to the rule A_R[j]. The LISP S-expression A[j] (which is an expression corresponding to a single interpretation of the affix A_R[j]) has the following structure

\[ \text{Sexpr A[j]} = ((\text{ARI}).(\text{SEF})) \]

where ARI ("affix rule information") contains pertinent syntactic information and SEF holds a set of string editing functions used for processing exceptions.

SEFs are used for

1. infix processing (inclusion and removal);
2. processing grammatical "exceptions" [e.g. in the so-called ‘hitpael’ form of the Hebrew verb we can realize that the prefix letters ‘dathel’, ‘tet’ and ‘tav’ appearing, due to phonological context, in the third position, really belong in the second, and are not parts of the stem).
3. processing affix ambiguity (e.g. ‘jod’ as the suffix of the Imperative vs. the same ‘jod’ in the role of the suffix of first person singular possessive).
The format of a SEF is as follows:

\[(\langle f \rangle \langle w \rangle \langle c_1 \rangle \langle c_2 \rangle [f[B]])\]

where

\(w\) stands for the word string
\(f = _f\) (meaning that function \(f\) is executed, if permitted)
\(?f\) (meaning that the function is executed if a Boolean function \(f[B]\) is true)
\(c_1 = \langle \text{char} \rangle\) (meaning that \(f\) is applied to the first \(\langle \text{char} \rangle\) in the string)
\(\langle \text{integer} \rangle\) (meaning that \(f\) is applied to the char with number \(\langle \text{integer} \rangle\) from the left)
\('s\) (meaning that \(f\) is applied to the last char in the string)
\(''\) (\(f\) is applied to an empty string)

\(c_2 = \text{char OR empty string.}\)

The algorithm "get_dic_forms" follows.

(1) Obtain the input word \(W\).
(2) Build list \(L\) of all possible affix rules \((A_R)\) for \(W\) \((L\) includes only NAMES of \(A_R)\):
\(L \leftarrow (\text{all} \ast \text{l} \text{if} \text{W}).\) For the input word ‘abcd’ the function \(\text{all} \ast\) produces the following list of affix rules to be checked:
\(\ast, a \ast, a b \ast, a d, a d, a d, a d\) (we postulate the rule that the residue \(\ast\) cannot be shorter than 2 symbols).
(3) With every member of \(L\) DO:
   (a) take \(W\) and strip affixes off it in accordance with every applicable \(A_R[i]\):
   \[ ((\text{GETW} W A_R[i]) \rightarrow W[1]) \]
   (It becomes clear at this point that the mechanism of 2.1.3 is an ad hoc measure: we could strip the word of any of the seven prefixes or their combinations HERE, at the same time as other affixes; this would simply involve writing down several thousand additional affix rules.)

(c) with every \(A[1]\) DO:
   (1) \(W[2] \leftarrow (\text{eval} (\text{cdr} A[1]))\); at this point all the functions listed in the given affix rule are applied; \(W[2]\) is a final candidate for the dictionary entry for a given word. \((\text{cdr} A[1])\) holds functions that modify \(W[1]\);
   (2) Check whether \(W[2]\) is an entry head. If YES then check also whether there is a syntactic match between the information in the main dictionary and \(A_R[i]\). This is carried out by the function "part_of_speech". \((A\) match occurs if, for example, the affix rule applies to nouns and the dictionary entry for the entry candidate contains the category "noun."
   If a match occurs, we unite \((\text{cdr} A[1])\) and all the information from the dictionary entry and \(W[2]\) itself. The result is added to the list of variant outputs;
   if NO MATCH then \(1 \leftarrow 1 + 1\);
   if NO then \(1 \leftarrow 1 + 1\);
   (3) with the list of candidate results we return to the outer loop.

The package "get_dic_forms includes the following groups of functions:

(1) string manipulation functions (more on these see above)
   (a) \(_s\) ‘abc ‘b’ d) will replace b by d in the string abc and return: ‘adc
   (b) \(_a\) ‘abcd 3 ‘e) will append ‘e after the third character in the string and return: ‘abced
   (c) \(_b\) ‘abbed ‘b’ e) will append ‘e before the first b in the string and return: aebbed

(2) functions of the all \(\ast\) routine (see above). This heavily recursive routine returns a list of all combinatorially legitimate combinations of prefixes, stems and suffixes for a given input word;

(3) the function "is_ok" This function obtains an \(A_R[i]\) and the input word after the affixes have been stripped off; it executes all the functions which are listed under \(A[j]\) (the "subrules" of \(A_R[i]\)) and if, first, the result of the application of each of these functions is in the dictionary and, second, the result of the part-of-speech compatibility check is positive, then the function unites the information on the word in the dictionary and in the corresponding affix rule and returns this information in a list, followed by the stem itself.
3.1.3 The word prefix tree

The raison d'être of this algorithm is to decrease the number of affix rules. In fact, it rather hampers the performance of the program, to say nothing about elegance. As mentioned above, the algorithm takes care of the one-letter prepositions 'beth', 'kaf', 'lamed' and 'mem'; conjunctions 'shin' and 'vav' and the (definite) article 'heci'—as well as of all the legitimate combinations thereof. This algorithm traverses a small transition network built specifically to represent the prefix structure in a Hebrew word.

The arcs are marked by the GROUPS of prefixes whose appearance in the word makes it possible to make a transition on this arc. The syntax of the only arc type in this transition network is as follows:

\[ (\text{next-cat} \langle \text{prefix} \rangle \langle \text{weight} \rangle \langle \text{checkword} \rangle \langle \text{state} \rangle) \]

where \( \langle \text{state} \rangle \) corresponds to "term act" in ATNs, next_cat is the name of the arc, \( \langle \text{prefix} \rangle \) is one of the above 7 prefixes or any legitimate combination thereof; weight and checkword will be explained below.

The algorithm itself can be described as follows:

1. Check whether the input word begins with one of the 7 prefixes or a combination of them.
2. If YES then the given next_cat \( \langle \text{prefix} \rangle \) is carried out:
3. The prefix is removed;
4. GET_DIC_FORMS is run on the remainder of the word.
5. In the result of GET_DIC_FORMS (which, the reader will recall, is a sequence of \{Wolij\}, \( i = 1 \ldots m \)) all entries whose grammatical information does not contain a checkword from the given next_cat arc are deleted. This procedure serves as a preliminary filter: it discards words that include prefixes of some other part of speech, not permitted to appear with a word of the given part of speech.
6. If the final result is not NIL then it is added to the global list of results, and the weight featured in the instance of the arc, to the global weight (if the weight is 20 (an arbitrary number), it means that the sentence is interrogative, and the given word is a question word).
7. If the final result is NIL then we concatenate the prefix back in its place in the input word and proceed to check the rest of the list of prefixes on the given arc.

The group of functions taking care of this part of the program includes:

1. next_cat (see above) makes use of pre_root;
2. pre_root obtains the union of grammatical information (from the dictionary) about the candidate stripped prefixes and the root itself and inserts the information on prefixes into the entry of the root. The latter operation is done with the help of the functions join-pre and pre-check.

3.1.4 Output

The program MORPHPARSE outputs a "small dictionary" which consists of zero or more output words (in Hebrew it is highly probable to obtain more than one morphological analysis for a given string of letters). The actual structure of an output word \( W_o \) is as follows:

\[ W_o = (E[1] \ldots E[q] \ E[r]), \]  

where \( E[i] = (EH \cdot (GI)) \), where

\( EH \) is Entry Head, an atom, and \( GI \) is Grammatical Information, a list of atoms, each of which is a grammatical category name.

This program was devised as a part of a parsing system. This dictated the necessity of organizing the small dictionary at the sentence level.
The data structure containing the SD ("small dictionary") for a sentence of K words, each of which has at most m morphological readings, is a two-dimensional array of the following form:

\[
\begin{align*}
Wo[21] & \quad Wo[22] & \ldots & \quad Wo[2k] \\
\vdots & \quad \vdots & \ddots & \quad \vdots \\
\vdots & \quad \vdots & & \quad \vdots \\
Wo[m1] & \quad Wo[m2] & \ldots & \quad Wo[mk]
\end{align*}
\]

To describe how the small dictionary is filled in it will be necessary to give an account of the top level control mechanism of the parsing program.

### 3.2 Top Level Control of the Parsing System

The main function of the system is the function \( \text{IN} \langle \text{filename} \rangle \). If \( \langle \text{filename} \rangle \) is not specified, the function works interactively. This function reads sentences from input until it reaches \text{end.of.file} or receives "quit". For every sentence morphological analysis is carried out whose results are stored in the small dictionary. Next, syntactic analysis is carried out on this small dictionary. These tasks are performed by the function \text{inputsent}. \text{Inputsent} scans the words in the input sentence, performs morphological analysis on them and produces a column for the small dictionary. This loop is performed until \text{end.of_line} or period appears. Next \text{inputsent} calls the function \text{parser} to perform the syntactic analysis. Note that the small dictionary array is local to the function \text{inputsent}.

#### 3.2.1 Parser

The parser, the syntactic analysis program, performs two main tasks: first, it builds all the combinatorially possible strings of syntactic meanings of words to be parsed and, second, calls for the ATN interpreter to process each such candidate string.

The backtracking mechanism used is centralized and is not dealt with in the ATN interpreter itself. If the average number of meanings in an input word is \( n \), and the length of the sentence is \( m \) then the program scans \( n^m \) candidate strings on average. The ordering algorithm is based on the well-known “meter principle”.

#### 3.2.2 ATN interpreter

Note. The input pointer, next-word, is moved by the function (to \( \langle \text{state} \rangle \)), which is standard ATN. When a certain arc cannot be traversed, but in the process of finding this out we have already moved the input pointer again, we can return to the original state of the input because the input sentence, \( \text{sent} \), is transferred as a parameter to every function. Let us have a quick look at the functions that implement the interpreter.

1. (def-net) obtains the list of register names to be used in the given fragment of the network. The name of the first state in the network becomes the name of the network itself, net-name. The register names are stored in the property list of net-name under the key “regs”. The body of the network is the global value of net-name.

2. (ev-sent) is the main function of the interpreter. It obtains the input sentence, puts the first element of the sentence in the special register “$” and assigns value \(-1\) to the variable “level” and performs push to the network whose net-name is “s”. If the result is nil, the message “incorrect sentence” is printed.

3. (mem), (wrd), (jump), (pop), (buildq), (fcat), (getf), (getr), (nullr), (addl), (addr), (look_ahead) and (what_token) are either standard ATN functions or special predicates in our implementation (see above, Section 2.3). “fcat” is our alias for the standard ATN arc “cat”.

4. (doacts \( \langle \text{actions} \rangle \langle \text{sent} \rangle \)) executes actions “setr” and “to”; if the action name is not one of the above, then the function assumes that the name is that of a standard YLISP function.
(5) (doarc) enters a loop on arcs within a state; when it finds an executable arc, it executes it by means of the function (doarc) and stops.

(6) (doarc) checks whether the predicate of the arc has the value "true" and if yes, it further checks whether the arc-name is "pop" and if yes, calls the function (pop); if the arc-name is not "pop", (doarc) calls the function (trans).

(7) (trans) monitors the execution of the specific arcs.

(8) (to) calls (do_to (cdr sent) (car sent) state). The latter function is defined as (do_to sent $ state). So, the sentence and the register "$" obtain new values, but when we return from the lower level of recursion with "nil", the above variables regain their original values.

(9) (push) obtains the argument "args". It stores the place where to jump. If the test on the push arc succeeds, then the function (do_push where_to_jump $) is called, and the result of the (do_push) is returned as the value of the register $.

(10) (do_push state $): "state" is a net-name we want to push; $ contains the current word (earmarked to be the first input word in the network). The sentence is bound in the YLISP stack as a parameter, so that both while pushing and popping we protect the correct value. (do_push) builds a small YLISP program:

    (return
     (prog (net-name regs)
     (doarcs net-name-body))
    )

and evaluates it with the help of the YLISP interpreter "eval". This is done in order to create new register names in the YLISP stack, so that when we come out of the function (prog), we return to the previous state from the standpoint of the registers and the input.

4. CONCLUSION

The paper has described an application of augmented transition networks to parsing yet another natural language. The system of parsing Hebrew has been used in conjunction with a semantic parsing module that was developed separately and is essentially an alternative parser for the conceptual dependency knowledge representation language (modified by the present authors). Both the morphological and the syntactic parsers of Hebrew can be used in different applications.

The paper is also a description of an extensive Lisp software system. An important feature of the software system is that it can be used for parsing both Hebrew and English (and, in fact other languages for which an ATN grammar is specified).

Acknowledgements—The authors are indebted to the members of the natural language processing project at the Hebrew University, and especially Jaakov Levy, Shmuel Bahr, Shaul Markowitz and Chagit Attiya. Many thanks are also due to Allen Tucker for much encouragement during the preparation of the final draft of this paper.

REFERENCES


About the Author—SERGEI NIRENBURG was born in 1952 in Kharkov, U.S.S.R. He attended the Department of Mathematical and Computational Linguistics at Kharkov State University from 1969 to 1974, graduating with an M.A. (summa cum laude) in Computational Linguistics. In 1975 Dr Nirenburg emigrated to Israel, where in 1980 he received his Ph.D. in Linguistics from the Hebrew University of Jerusalem. In 1980–82 he was a junior lecturer at the Hebrew University Department of Computer Science,
where he organized a research team in natural language processing. Since 1982 he has been an Assistant Professor in the Computer Science Department of Colgate University. Dr Nirenburg's research interests are in the field of artificial intelligence and cognitive science. He is a member of The Association for Computational Linguistics, The American Association for Artificial Intelligence and The Cognitive Science Society.

About the Author—YOSEF BEN ASHER was born in 1957 in Jerusalem, Israel. He attended the Departments of Physics and Computer Science at the Hebrew University of Jerusalem, receiving B.Sc. Computer Science in 1978. In 1980 he received an M.Sc. degree from the same department (dissertation topic: "Extending Pascal for artificial intelligence applications"). At present he is a doctoral student at the Hebrew University of Jerusalem. Mr Ben Asher has been active in the natural language project at the Hebrew University during 1980–82 and is also associated with the machine translation project at S.T.I. in Jerusalem.

APPENDIX A

Example of a Parse Tree

Tree (2) contains the results of the application of the HUHU parser to sentence (1): English translations are added in brackets.

(1) Batei hasefer bair shovtim bemeshchek kin'at hamisha shavuot

(1') The city schools have been on strike for almost five weeks.

(2) (mishpat [sentence]
   (sug HATSGARATI HOVE [type DECLARATIVE PRESENT])
   (tsiruf_etsem [noun phrase]
     (shem_etsem [noun] BAIT)
     (milat_kinyan [nominal adjunct] SEFER)
     (tsiruf_yachas [prepositional phrase]
       (milat_yachas [preposition] B)
       (tsiruf_etsem [noun phrase]
         (shem_etsem [noun] IR)
         (mispar YACHID [number SINGULAR])
         (min NEKIVA [gender FEMININE])
       )
     )
   )
   (mispar RABIM [number PLURAL])
   (min ZACHAR [gender MASULINE])
 )
 (tsiruf_poaal [verb phrase]
   (poal [verb] SHAVAT)
   (tsiruf_yachas [prepositional phrase]
     (milat_yachas [preposition] BEMESHECH)
     (tsiruf_etsem [noun phrase]
       (kdam_toar_poaal [pseudverb] KIM'AT)
       (milat_mispar [numeral] HAMISHA)
       (shem_etsem [noun] SHAVUA)
       (mispar RABIM [number PLURAL])
       (min ZACHAR [gender MASULINE])
     )
   )
 )
)
APPENDIX B

The Hebrew ATN (Fragment: Upper Level Sentence Network)

(def-net '((subj vpr ppr pprl type questr)
  '((s
    (cat nlzy'ild [questionword] ; check for question
t      (setr questr $) ; put quest word in questr.
      (to nounverb)) ; try VP or NP.
    (push vp
      (getf d $) ; question prefix?
      (setr questr 'd) ; light questr register
      (setr vpr $) ; put verb-phrase in vpr.
      (to findtype)) ; check if DEC or QUEST sentence
    (push pp
      t
      (addr ppr $) ; catch a prepositional phrase.
      (to s))
    (push np
      t ; no test (true).
      (setr subj $) ; push in subj.
      (setr type 'geei) [decl] ; Declarative sentence!
      (to nextpp))
  (jump nextpp t))
(nounverb ; try to match NP or VP.
  (push np ; trying NP
t    (setr subj $)
    (to questvp)) ; try VP after the NP.
  (push vp ; trying VP (a separate
t    (setr vpr $) ; subnetwork here!)
    (to findtype)) ; next problem: find type of
    ; sentence
  (jump findtype t)) ; match '.' or '?'
(questvp ; after matching NP try VP.
  (push vp
t    (setr vpr $)
    (to findtype)) ; find sentence type.
  (jump findtype t))
(nextpp
  (push pp t
    (addr ppr1 $)
    (to nextpp))
  (push vp t
    (setr vpr $)
    (to findtype)))

(findtype
  (cat nilzy'ild t
    (setr type 'y'ild)
    (to poparc)
    ; if so, type is QUEST
    ; to wrap-up
  )
  (jump poparc t))

(poparc
  (pop (and (nlist sent) (not (nullr type)))
    ; the final structure is
    ; ready: pop and finish.
    (buildq ((nyth (geb +)) (nlzy'il'd +)
      +
      +
      +)
      type
      questr
      subj
      ppr1
      vpr
      ppr))))
2 examples of affix rules (Hebrew morphological grammar) follow. The entire grammar contains over 90 rules. English translations are in brackets.

; 067/ "z"
; ends with "tay" (Hebrew ASCII code "z")
; verb [of binyan] "kal" past singular masculine second active
; e.g. achalta [you ate/masc]
; verb "kal" past singular feminine second active e.g.
; achtalt [you ate/fem]
; noun in "smichut" form (another noun necessarily follows to
; form a phrase corresponding to the English "table napkin")

(putrule '"z"
(setg '"z"

((rvm [noun] igic [singular] pwad [feminine]
  gnikeyz [ _of_ ]).((a ww 's 'd)))
; the string manipulation function says that the letter "hei", "d" in
; Hebrew ASCII code, must be appended at the end of the stem
((trl wl ["kal"] rax [past] igic ypi [second]
  tril [active]).((EE ww)))
; the string manipulation function returns the string as is
((trl wl deed [present] igic pwad tril).((?s ww 2 '"' (=str ww 2
  'e))))
; the string manipulation function deletes the second letter of the
; stem
; if this letter is "vav", "e" in Hebrew ASCII
]

; 048/ p*
; starts with "nun" (Hebrew ASCII code "p")
; verb [of binyan] "piel" future plural first active:
; nedaber [we will talk]
; verb "kal" future plural first active:
; nitzhak [we will laugh]
; verb "nifal" past singular third passive:
; nividak [(it has been) checked]
; verb "nifal" present singular masculine passive:
; nividak [(he is being) checked]
; verb "hif'il" future plural first active:
; nazmin [(we) will invite]
; verb "pual" future plural first passive:
; nechubbaa [we will be washed]
(putrule 'p*"
(setg 'p* "( (trl tirl rzic [future] xaim [plural]
     x'yeo [first] tril).((EE ww)))
((trl wl rzic xaim x'yeo tril).(EE ww)))
((trl rax igrig yliyi [third] fkkx [masculine]).((EE ww)))
((trl deed igrig fkkx).((EE ww)))
((trl wl rzic xaim x'yeo tril).((?s ww 3 'n' (=str ww 3 'e))))
  ; delete third character if it is "vav"
((trl dtril rzic xaim x'yeo tril).((?s ww 3 'n' (=str ww 3 'i'))))
  ; delete third character if it is "yod"
((trl terl rzic xaim x'yeo qail).((?s ww 2 'n' (=str ww 2 'e')))
  ; delete second character if it is "vav"
((trl rzic xaim x'yeo pitrl).((EE ww)))
((trl rax igrig yliyi fkkx pitrl).((EE ww)))
((trl rax igrig yliyi fkkx pitrl).((?s ww 1 'i (=str ww 1 'e'))) ; if first character is "vav", substitute it with "yod"
; treatment of special stem forms follows:
; pey-yod (root starting with "yod") : nosad
((trl rax igrig yliyi fkkx pitrl).((a ww '$ (lastchar ww) (eq (strlen ww) 2))))
  ; kfusim (repeated letters in root): nasav
((trl deed igrig fkkx pitrl).((EE ww)))
; pey-yod (root starting with "yod") : nosad
((trl deed igrig fkkx pitrl).((?s ww 1 'i (=str ww 1 'e))))
; pey-yod (root starting with "yod") : nivvased
((trl rzic xaim x'yeo pitrl).((?s ww 1 'i (=str ww 1 'e'))) ; kfusim (repeated letters in root): nibboz
((trl rzic xaim x'yeo pitrl).((?s ww 2 (lastchar ww) (=str ww 2 'e'))))
; kfusim (repeated letters in root): nissav
((trl rzic xaim x'yeo pitrl).((?s ww '$ (lastchar ww) (eq (strlen ww) 2))))
)

For comparison, below is one rule of the affix rules for English, created as part of the English parsing system that was built with the help of the same programs:

; deals with the suffix ED
(putrull '*ed
(setg '*ed "( (V past).(EE ww)) ; look -- ed
(V pastpart).(EE ww))
(V past).(a ww '$ 'e)) ; mov(e)-- ed
(V pastpart).(s ww '$ 'y))
(V past).(s ww '$ 'y)) ; classif(y) -- classif(i) ed
(V pastpart).(s ww '$ 'y)))
APPENDIX D

The HUHU ATN Interpreter

The morphological parser was treated in greater detail in the text. Here we present the part of the system that interprets and executes the Hebrew ATN.

; interp - ATN network interpreter.
; ---

(global 'debugflag nil t) ; debugging switch

(def ev-sent
  (lambda (sent)
    (prog ($ r$ res level)
      (set 'level -1)
      (set '$ (setcdr 'sent))
      (cond ((set 'res (dopush 's $)) (return (car res)))
            (t (return "incorrect sentence."]))

(def def-net ; define an ATN network
  (lambda (regs net)
    (prog (net-name state)
      (set 'net-name (caar net))
      (put net-name 'regs regs)
      [put net-name 'eval-after
        (list (list 'def-net
                    (list 'quote regs)
                    (list 'quote net))

        1 (cond ((null net) (return t))
                 (set 'state (setcdr 'net))
                 (setq (car state) (cadr state))
                 (go 1)]

(def mem ; is $ member of (car name)
  (lambda (name)
    (cond ((and (eval (cadr name)) ; predicate non-null
               (memq $ (car name)))
           (doacts (cddr name) sent])

(def wrd ; is $ eq to (car name)
  (lambda (name)
    (cond ((and (eval (cadr name)) ; test predicate
                (eq $ (car name)))
           (doacts (cddr name) sent])

(def putquote ; put quote around an argument
  (lambda (arg)
    (list 'quote arg)
(def doacts ; do each of the "actions"
  (lambda (actions sent)
    (prog (thisone)
      (cond ((nlist actions) (break ""illegal action list""))
        (set 'thisone (setcdr 'actions))
        (cond
          [(eq (car thisone) 'to)
            (return (eval thisone))
          (eq (car thisone) 'setr)
            (set (cadr thisone) (eval (caddr thisone))))
        (t (eval thisone))))
      (go 1))
    (def doars ; do each arc from this state until one
      ; succeeds or we run out of arcs to try
      (lambda (arcs)
        (prog (result)
          (cond ((nlist arcs) (return nil)) ; no more
            (set 'result (doarc (setcdr 'arcs)))
            (return result)) ; success
            (t (go 1))
          (def doarc ; try a single arc
            (lambda (arc)
              (cond ((eval (caddr arc)) ; test the predicate
                (cond
                  [(eq (car arc) 'pop) (pop (cadr arc))]
                (trans arc) ; special actions.
              (trans arc))
              (def trans ; special actions
                (lambda (arc)
                  (cond
                    [(eq (car arc) 'wrd) (eval arc)]
                    [(eq (car arc) 'push) (set 'r$ (eval arc))]
                    [(eq (car arc) 'mem) (eval arc)]
                    [(eq (car arc) 'jump) (eval arc)]
                    [(eq (car arc) 'cat) (eval (cons 'fcat (cadr arc)))]
                    [(eq (car arc) 'vir) (cond
                      (not (null (eval (cadr arc)))))
                      (eval (caddr arc))
                    )
                  )
                )
              )
              (def to ; go to specified state
                (lambda (state)
                  (and (debugflag)
                    (patomlist ""to " state "n")
                    (doto (and sent (cadr sent)) (and sent (car sent)) state)
                  )
                )
              (def doto ; rebind sent and $ for MY environment
                (lambda (sent $ state)
                  (doars (gval state)
                )
              )
            )
          )
        )
      )
    )
  )
(def jump ; go to state, consume no input
  (nlambda (state)
    (and (debugflag)
      (patomlist "jump " (car state) "n"))
    (cond ((nlist state) (tpatom "jump: illegal args.")
              (return nil))
      ((nlist (cdr state)) (tpatom "jump: illegal args."))
              (return nil))
    (eval (cdr state)) t)
  (t (return nil))) ; test predicate failed -
  (doarcs (gval (car state)) ; don't jump; go to new state
    (gval state))

(def dopush ; recursively call an ATN net
  (lambda (state $)
    (and (debugflag)
      (patomlist "push " state "n"))
    (inc level)
    (eval (list 'prog
                (get state 'regs)
                (list 'return
                       (list 'doarcs (putquote (gval state))
                        (gval state))))

(def push ; full version of push with actions etc.
  (nlambda (args)
    (prog (to result)
      (set 'to (setcdr 'args))
      (cond ((nlist args) (tpatom "push: predicate missing.")
              (return nil))
       ((eval (setcdr 'args)) t)
       (t (return nil))) ; predicate failed -
       (set 'result (dopush to $))
      (cond ((null result) (return nil))) ; failure of
       (set '$ (setcdr 'result)) ; entire push
      (cond (args (doacts args result))
             (t (return $))

(def pop ; pop a recursive state
  (lambda (retval)
    (dec level) ; decrement recursion level.
    (return (cons retval (cons $ sent))

(def c-end ; are we at the end of parsing?
  (lambda nil
    (and (eqn level 1) (nlist sent))

(def buildq ; full buildq, except for "@
  (nlambda (fragment)
    (dobuildq (car fragment) (cdr fragment))
  "
(def dobuildq ; build a list of registers and other stuff  ; for returning results
    (lambda (fragment reglis tmp)
      (cond ((nlist fragment) nil)
            ((eq (car fragment) '+)
              (set 'tmp (eval (car reglis)))
              (cond (tmp (cons tmp (ddobuildq (cdr fragment) (cdr reglis))))
                                  (t (dobuildq (cdr fragment) (cdr reglis)))
            )
            ((eq (car fragment) '0)
              (set 'tmp (eval (car reglis)))
              (cond (tmp (append tmp (dдобiuldq (cdr fragment) (cdr reglis))))
                                  (t (dobuildq (cdr fragment) (cdr reglis)))
            )
            ((eq (car fragment) '$)
              (cons $ (dobuildq (cdr fragment) reglis))
            )
            ((eq (car fragment) ')
              (cons (eval (car reglis))
                    (dobuildq (cdr fragment) (cdr reglis))
            )
            ((atom (car fragment))
              (cond ((setq tmp (dobuildq (cdr fragment) reglis))
                                 (cons (car fragment) tmp)
              )
              (setq tmp (dobuildq (car fragment) reglis))
              (cons tmp (dobuildq (cdr fragment) (leftover reglis))))
              (t (dobuildq (cdr fragment) (leftover reglis)))
      )

(def leftover ; which registers have not yet been used?
    (lambda (reglis)
      (*nthcdr reglis (countplusses (car fragment))
    )
    )

(def countplusses ; count how many "+" appear in lis
    (lambda (lis)
      (cond ((eq lis '+) 1)
            ((nlist lis) 0)
            (t (+ (countplusses (car lis))
                 (countplusses (cdr lis))))
      )
    )

(def fcat ; full version of "cat"
    (nlambda (args)
      (prog (cat)
        (setq 'cat (setcdr 'args))
        (cond ((dogetf $ cat)(doacts (cdr args) sent]))
      )
    )

(def dogetf ; internal version of getf
    (lambda (wrd feature)
      (setq (feat-lis result)
              (set 'feat-lis (gval wrd))
              (cond ((null wrd) (return nil))
                    (cond ((null feat-lis)
                                  (patomlist wrd "" not in dictionary")))
                    (cond ((memq feature feat-lis) (return feature))
                                    (t (return feature))))
    )

(def getf ; called from doacts
    (qlambda (feat reg)
      (dogetf (eval reg) feat)
    )
  )
(def getr ; get a register's content
  (lambda (reg) reg)
)

(def nullr ; is register "reg" nil?
  (lambda (reg)
    (null reg)
)

(def addl ; add to left end of register.
  (qlambda (reg val)
    (set reg (cons (eval val) (eval reg))
  )

(def addr ; add to right end of register.
  (lambda (reg val)
    (prog (r v)
      (set 'r (eval reg)) (set 'v val)
      (set reg
        (cond
          ((null r) val)
          ((atom r) (list r v))
          ((atom (car r)) (list r v))
          (t (append r (ncons v)))
        )
      )
    )
  )
)

(def lookahead ; returns true if catg is in the list of the lookahead word
  (lambda (n catg)
    (prog (ww )
      (set 'ww (*nth sent n))
      (cond ((null ww) (patom "ERR TOO FAR") nil)
            (t (member catg (eval ww))
        )
    )
  )
)

(def find (lambda (lis a)
  ; returns the place of a in lis as a number
  (cond ((nlist lis) nil)
        ((eq (car lis) a) (inc c))
        (t (inc c) (find (cdr lis) a))
  )
)

(def whattoken (lambda (greg ctg)
  ; return the token of the category catg in greg
  (prog (c)
    (set 'c 0)
    (*nth (eval greg) (addl (find (eval greg) ctg)))
  )
)

(def nsubstr ; new substring ....
  (lambda (fr len str)
    (substr (- fr l) len str)
  )
)

(def setr
  (qlambda (to from)
    (set to (eval from))
)