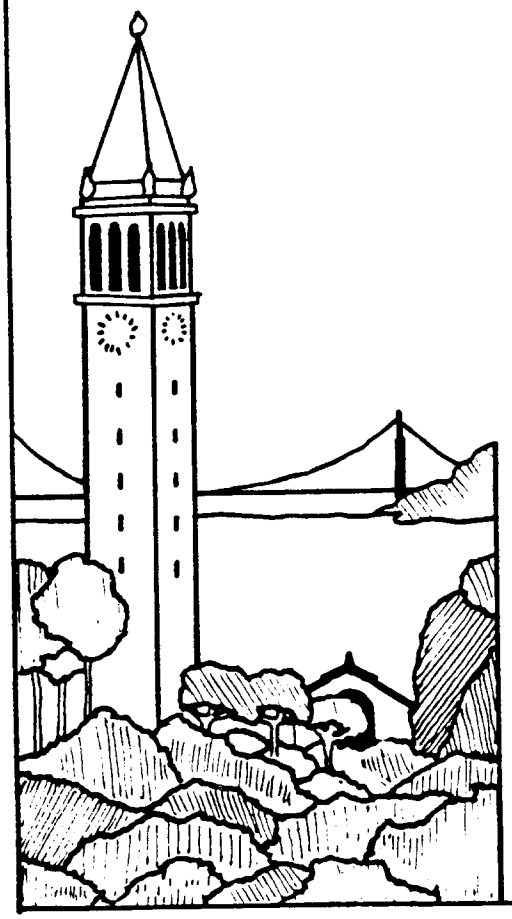
ALANA
Augmentable LANguage Analyzer

Charles A. Cox



Report No. UCB/CSD 86/283

January 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This report introduces a reimplementation of the ideas for phrasal analysis originally implemented in PHRAN as part of the original Unixotnote{UNIX is a trademark of ATT Bell Laboratories} Consultant project (UC). The new implementation, ALANA, presents a general algorithm using chart parsing techniques for phrasal analysis.

Also presented in this report is a discussion of KODIAK, the new form of representation from UC Berkeley into which ALANA analyzes. We include a description of how language is represented in ALANA and how it ties in with the KODIAK representation model.

Finally, we look at how ALANA fits into the new UC, a new implementation of UC using ALANA, KODIAK, and the latest ideas from the UC Berkeley BAIR (Berkeley Artificial Intelligence Research) project on understanding and planning.

Author Charles Arthur Cox

Title ALANA--Augmentable LANguage Analyzer

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee: _____, Research Adviser

11/25/85
Date

_____

12/2/85
Date

ALANA—Augmentable LANguage Analyzer

Charles A. Cox

January 26, 1986

Abstract

This report introduces a reimplementation of the ideas for phrasal analysis originally implemented in PHRAN[WA80] as part of the original Unix[1] Consultant project (UC[WAC84]). The new implementation, ALANA, presents a general algorithm using chart parsing techniques for phrasal analysis.

Also presented in this report is a discussion of KODIAK, the new form of representation from UC Berkeley into which ALANA analyzes. We include a description of how language is represented in ALANA and how it ties in with the KODIAK representation model.

Finally, we look at how ALANA fits into the new UC, a new implementation of UC using ALANA, KODIAK, and the latest ideas from the UC Berkeley BAIR (Berkeley Artificial Intelligence Research) project on understanding and planning.

This report was sponsored by

Defense Advanced Research Project Agency DARPA N00039-82-C0235

Office of Naval Research ONR N0014-80-C-0732

[1]UNIX is a trademark of AT&T Bell Laboratories

Contents

List of Figures

1 Introduction

The main goal of the research presented in this report was to improve and simplify a natural language analyzer developed in the Berkeley Artificial Intelligence Research (BAIR) project. This previous analyzer[WA80], named PHRAN for PHRasal ANalyzer, used the idea that information relating natural language to concepts be stored as a pairing between linguistic patterns and concepts. Concepts in PHRAN are represented in a special language such as Conceptual Dependency[SA77] (CD). The same main idea of language to concept pairs is also used in my new analyzer ALANA (for Augmentable LANguage Analyzer), yet ALANA is a cleaner implementation of the ideas of PHRAN and uses an improved method for representing concepts.

What makes ALANA better than PHRAN is that PHRAN was designed only to analyze into Conceptual Dependency structures, whereas ALANA, being developed independently of representation, depends less on the language it is analyzing into, and more on other components of the underying understanding system. In this report, I shall also argue that PHRAN was trying to do too much as a analyzer. This made its use too restrictive. ALANA, on the other hand, is a smaller and more versatile program, which, using the same mechanism can analyze not only into Conceptual Dependency structures as PHRAN had done, but also to different knowledge representations such as the Berkeley KODIAK[Wil84] representation language also to be described in this report. Finally, what makes ALANA better than PHRAN is that PHRAN had been developed incrementally over a period of several years, and ended up being very difficult to maintain. PHRAN needed an overhaul, and ALANA was the result.

The current ALANA has successfully fit into two separate versions of UC[WAC84], the Unix[2] Consultant natural language dialogue system being developed by the BAIR project. The ALANA code, being much smaller and more modular than PHRAN, will be much easier to maintain than was PHRAN. ALANA was designed to achieve everything PHRAN could do plus be easily modifiable and extendable to other systems. Also, the knowledge base is separated from control so that, as with PHRAN, the analyzer can be theoretically extended to other natural languages.

This report will focus mainly on the ALANA's implementation. However, before we look at the analyzer, we will look, as background, at some parsing techniques that inspired ALANA's development in some way, and we will see

[2]UNIX is a trademark of AT&T Bell Laboratories

the strengths and weaknesses of these techniques. In addition, to elucidate the description of concept building, we will see an overview of the KODIAK representation system and its interpreter (originally written by Peter Norvig). Then I will describe in detail how pattern matching is accomplished followed by how declarative style concepts are translated into KODIAK. My descriptions will be accompanied by my ideas on how I think natural language processing is done in a person's head.

2 Background

Before going into the details of ALANA, let us first look at the context in which it is built. We will first look at some previous parsing techniques and examine their strengths and weaknesses. We will end this section by looking in detail at the strengths and weaknesses of PHRAN, and giving an overview of the KODIAK representation system.

2.1 Other Parsing Techniques

Most parsing techniques fall into one of roughly two main categories, those doing a lot of syntactic analysis followed by semantic analysis, and those integrating the two analyses. As we shall see, ALANA's direct predecessors (PHRAN[WA80], ELI[Rie78], IPP[SLB80]) were more in the integrated category. We shall also see that ALANA attempts to be even more integrated than its predecessors by its continual inserting what it "finds out" from the parse into the semantic representation for other processes to deal with.

2.1.1 ATNs

Let us look first, however, at a syntactic analyzing technique which provided some of the inspiration for ALANA's pattern matching. An Augmented Transition Network[Woo70] (ATN) is an early technique for natural language parsing and has become a popular paradigm on which to build analyzers. It is used in such systems as PLANES[Wal78], LUNAR[WKN72], and even in a Unix Consultant system (UCC)[DH82] being developed outside of Berkeley. These systems all use ATNs to parse natural language input into formal queries for database management systems.

An ATN is an extension of an RTN where an RTN (for Recursive Transition Network) is simply a Finite State Machine[Har78] with the ability to

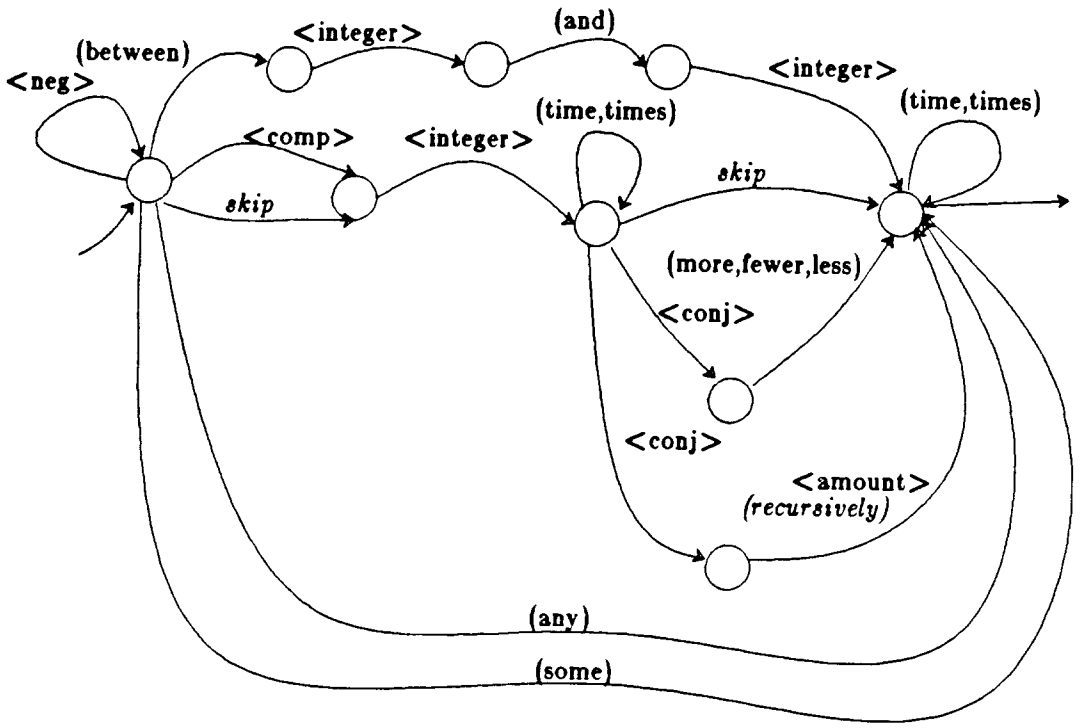
recursively use parts of itself to determine whether an edge traversal is possible. It can be shown that RTNs recognize the class of languages known as *Context Free Languages[Har78]*. We can build machines (i.e. write programs) to parse these languages by having the program read input from a stream and try all of the edges, and backtrack when it hits a dead end. Some researchers, however, believe that the syntax of English is not context free. If so, it cannot be parsed using only the power of RTN's. The extra power can be gained by using augments on the edges. Hence we have Augmented Transition Networks or ATNs.

An ATN is simply an RTN with the added ability that not only do the routines on the edges read from the input stream and return back to the caller, but they also can maintain and access global registers. With this added ability, they are functionally equivalent to a Turing Machine which means they should theoretically be able to compute any computable function. In other words, they should be able to recognize any language that is effectively recognizable by a machine. Hence, ATNs should be powerful enough to be used as a natural language parsing technique. A sample ATN, from Wood's article[Woo70], along with some of the *amount* phrases it can recognize is shown in Figure 1.

There are two main problems with ATNs. The first is that with only syntactic recognizers on the edges, it is strictly a syntax-based parser. The idea of having a strictly syntax-based parser was fashionable in the 1950's and 1960's when linguists were mostly studying the structure and distribution of morphemes in a language. Semantics was given a back seat to syntax in those days, and it was felt that the meaning of a analyzed sentence could be determined by merely computing a function of the dictionary meanings of the words and the syntactic construction of the sentences. Since that time, many linguists as well as AI researchers have argued against the separation of Linguistics into pure syntax and pure semantics[Lak69].

The other main problem with ATNs is their efficiency. They are usually implemented using a top-down, depth-first search mechanism, usually with blind backtracking. Thus, the edges are tried in some arbitrary order, and whenever an attempted traversal down one edge fails, the ATN routine will just try the next edge. The following example of this inefficiency is taken from R. Johnson[Joh83].

If we were to use a standard syntactically-oriented ATN confronted with the English sentence in Figure 2, we can get an alternate bracketing up to the word **program** which is rejected on finding the verb **enhance**. If the analysis routine started with the second parse, then upon seeing **enhance**, it would



<comp>	<conj>	<neg>
as many as	but	not
more than	or	none
at least	and	no
.	.	.
.	.	.
.	.	.

Figure 1: ATN amount subnet

have to back up all the way to the beginning of the sentence and start the parse again, including a second traversal of the Prepositional Phrase (PP) *in the program.* Note that the PP already had been correctly parsed, but we had to undo it, and then redo it over again.

The idea of a *cascaded ATN*[Woo80] is an extension to the purely syntactic ATNs described above. A "cascaded ATN grammar" has a semantic component which tries to build arbitrary structures. During processing, if the semantic component is unable to build the structure, it returns a failure message to the syntactic parser, which will then act as if it couldn't traverse the edge which caused it to call the semantic routine. The ATN parser will then backtrack just as in the regular syntactic parsing case. Such a method is used in Psi-Klone[BW80].

We shall see that the idea of using the syntax to determine which semantic

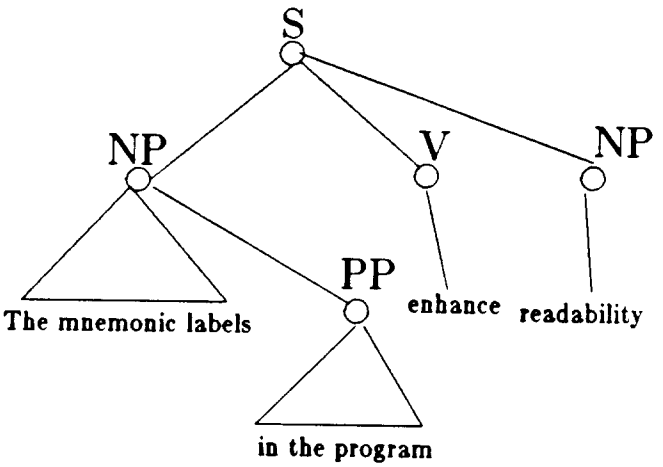


Figure 2: Correct parse of sentence

routines should be considered, as is done with cascaded ATNs, is also done to some degree in ALANA. The fundamental differences, however, between ALANA and cascaded ATNs, is that ALANA has much less emphasis on syntax, and also does neither blind backtracking nor depth-first search. Rather it allows the other semantic components to guide the analysis rather than be a slave to the syntactic parser.

2.1.2 ELI

Reacting to the pure syntactic analysis of parsing techniques, researchers began looking at how to get the meaning of sentences by performing semantic analysis of the input as it is being read, and not waiting until after pure structural analysis. It is this semantic-oriented tradition of analysis into which ALANA falls. In this section, we discuss one of the earlier attempts at semantic style analysis by looking at ELI[Rie78]. But before we look at ELI itself, we shall look at the theory behind it known as *Conceptual Analysis*.

Conceptual Analyzers work by intimately knowing about the representation into which they are analyzing, and thus set up *expectations* as they read the natural language input stream. With ELI (and later, PHRAN), the concepts were represented using Conceptual Dependencies (CD's), a graph-

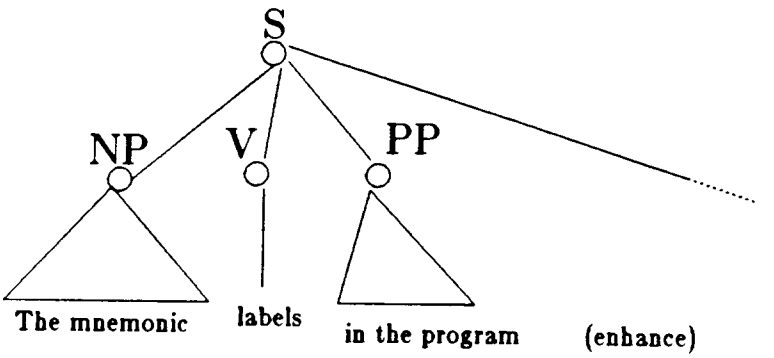


Figure 3: ATN parse needing to backtrack

ical structure in which all actions were reduced to about eleven primitive actions. The idea with Conceptual Dependency was that meanings could be represented unambiguously in a canonical form, and from that form inferences could be made. For example, the verbs 'give', 'take', 'transfer', 'donate', 'send' could all map into the ATRANS primitive. The ATRANS primitive has a source and a recipient. By using this primitive, the system can then infer such things as the source used to have the object, the recipient now has the object, and that the source no longer has the object[3].

Getting back to the actual conceptual analysis, let us look at an example as presented by Birnbaum and Selfridge[LB81]. When analyzing a sentence such as "Fred ate an apple", the system reads that sentence left to right. The words it reads early on (i.e. closest to the left), will influence how it reads the rest of the sentence. So, when the word 'Fred' is encountered, the system understands it to be a reference to a male human being named Fred. This idea is also shared in PHRAN and ALANA. It then stores the token it has named FRED into a short-term memory. When it encounters the word 'ate', the system understands that there is an "eating action" going on which in CD is represented by an INGEST CD frame. By the fact that INGEST takes an ACTOR and an OBJECT, the system knows it can expect references in the sentence to fill these slots. For example, there is an expectation that the ACTOR of the INGEST is animate. Since the system already has an animate

[3] There were actually three notions of transfer in CD: PTRANS, ATRANS, and MTRANS. ATRANS meant that an *abstract* transfer was taking place, thus what may be transferred may really have been a transfer of possession.

object in its short term memory (FRED), it goes ahead and makes FRED be the ACTOR of the INGEST.

Stepping back and looking at what we have seen so far, the system has only seen "Fred ate...", and without completing the analysis, it *already knows* that a person named Fred is doing some kind of ingesting. It also knows that it can expect an object as the direct object of this sentence, and that that object will be the thing ingested. It is this idea, of integrating linguistic processing with other reasoning and memory processing, that gives ELI and its descendants, including ALANA, efficiency and speed (in understanding) over syntax-first parsers.

Continuing along with the example, the system next reads the word 'an'. The system having expectations associated with 'an', knows that a Noun Phrase (NP) is coming up, and that the object associated with the NP should be marked as an *indefinite* object. When the word 'apple' finally gets read, the system knows it to be a food which is an object, which in turn is eligible to fill the INGEST frame suggested above. Thus, the system ends up with the frame

```
(INGEST
    ACTOR (FRED)
    OBJECT (APPLE REF (INDEF))
```

ELI was an early analyzer (written by Christopher Riesbeck[Rie78]) that did conceptual analysis for CDs. The analyzer executed in a top-down fashion and produced only conceptual structures for which it had *expectations*. analyzed into a conceptual structure only if it had previously been *expected*. Expectations were implemented using test-action pairs that were called *requests*. If the test were true, then the actions were executed. In the above example, one request tested for the concept of apple being an edible object. When it found that an apple was indeed an edible object, it knew that it could fit as an object of the INGEST, so the result or action of the request stated that the INGEST's object should get the apple concept.

There are two kinds of requests used in ELI, *lexical* and *conceptual*. Lexical requests are stored under words in a kind of dictionary. Thus, words such as 'Fred' will have a request stored under it where the *test* is T (denoting an always true condition), and the *action* is to add the structure (PP CLASS (HUMAN) NAME (FRED))[4], which is a CD structure indicating a human named Fred, to the C-LIST or short term memory of the system. Many

[4] We will not worry about the detailed syntax of the forms used by ELI. They are usually

words, such as verbs, have lots of requests associated with them. For example, the word 'gave' triggers requests to check which types of objects are in short term memory and which should be expected. In this way, the system is able to understand the different uses of 'gave' in sentences such as **Fred gave Sally a book**, **Mary gave Sam a punch**, and **John gave Sue a headache**. The requests associated with 'gave' check for all of these and build the appropriate conceptual structures.

An extra component was also put onto the system to look for noun-noun collocations such as **stairway handrail**, **cigar smoker**, **dog leash**, and **car seat**. Cases like these were handled by using heuristics[Ger77] for determining noun group boundaries. The heuristics of where the boundaries were depended on conceptual as well as syntactic knowledge.

In my view, it is the arbitrary nature of these heuristics and requests that gives the system problems. The problem is that in order to get ELI to work properly on a sentence, one needs to attach a lot of ad hoc knowledge at the word level (especially for verbs), and this knowledge needs to tell the system to look ahead and to look behind for arbitrary structures. Some of the problem cases can be illustrated by looking at idioms. If the word 'kick' is read by the system, then all forms having to do with 'kick' have to be handled as requests. This presumably includes idiomatic forms with 'kick' including **kick the bucket**. So, there has to be an explicit request that tests for 'kick' followed by 'the' followed by 'bucket'. If it is there, then we want to build the concept for dying[5].

As a result, entering new knowledge to this system is not a trivial task. We have to make sure that all of the requests get entered in the right order, which may mean indexing them under the word. As we saw earlier above, when the system saw the word 'gave', requests for all forms of 'gave' including **gave a book**, **gave a punch**, and **gave a headache** had to be triggered.

ELI also had a problem with forms such as **Chinese restaurant**. The intended meaning of **Chinese restaurant** is a restaurant which serves Chinese food, and may have a Chinese decor, etc. The problem was that it saw the word 'Chinese' before seeing 'restaurant'. After seeing 'restaurant', even with Gershman's analyzer, ELI could not realize that it was supposed to build a concept for a specialized type of restaurant unless an expectation had been set up for it. The only possible way to set up an expectation was to attach

---

pretty readable. The one word of this form which may be confusing is *PP* which stands for Picture Producer, or an object which can form a mental image of "seeing" the object.

[5]In CD, dying is a state-change, where death = -10 health.

a request to the word 'Chinese'. In other words, whenever ELI sees the word 'Chinese', it would have to expect that 'restaurant' will be one of the possibilities to follow.

But ELI has to know to expect other words as well after 'Chinese', for example 'laundry' in 'Chinese laundry'. The way to let it do this is to attach the requests for 'laundry' and 'restaurant' as well as all others to the word 'Chinese'. A more serious theoretical problem with just attaching knowledge to words is that there really is a generalization to `Chinese restaurant`, for we have `Italian restaurant`, `French restaurant`, `Indonesian restaurant`, etc. The generalization is that there is the idea of a NATIONALITY `restaurant` which we should not expect until after we have seen the word 'restaurant'.

The solution to this problem of one-way-only requests was to change how ELI handled requests by allowing them to look *backwards* as well as forwards. So, ELI ended up being a word-expert system with lots of arbitrary knowledge being stored at the word level, and with lots of requests being asserted just because a particular word in a phrase had been used. The logical extension of this word expert analysis idea is captured in Small's[SR82] word expert analyzer in which the knowledge was all attached to the words. Adding knowledge to these word experts becomes a very formidable task since it is difficult to know how new knowledge will affect the old. This problem continually recurs in analyzing natural language.

It was these problems that caused Wilensky and Arens[WA80] to propose a simpler method for dealing with maintaining linguistic and conceptual knowledge in the analyzer. Their solution was the *Pattern-Concept Pair* which we shall discuss next when describing PHRAN (**PHR**asal **AN**alyzer).

2.1.3 PHRAN

One main idea that PHRAN extended was ELI's idea of associating request-like knowledge to language components. However, in PHRAN the attachment is not just to words, but to *phrasal structures* as well. A simple example is the idiom '`(root kick) the bucket`' where '(root kick)' means 'kick' can actually be of any tense or be the root of a complex verb phrase (as in "would have kicked ...."). Instead of attaching requests to the *word* 'kick' that *expect* the words 'the' and 'bucket', assert the request only after we have seen the *phrase* 'kick the bucket', and associate the meaning with the *phrase*, and not with the individual words.

Another useful idea incorporated into PHRAN was that knowledge at-

tached to the phrasal constructs was *declarative*. This became important in that it not only made adding knowledge less of a chore than adding intricately interrelated procedural knowledge (as needed in ELI), but also that one could think of the knowledge as a *pairing* between linguistic information and conceptual information, and that analysis then became the process of taking the natural language input, matching it against the phrasal patterns, and then returning those concepts that had matched patterns. So, we had a 'language → concept' process.

By extending the idea of pairing linguistic information to concepts, there seemed no reason why one could not process in the reverse direction, i.e. by taking conceptual information and producing a linguistic form expressing that concept. This reverse process is just that of natural language *generation*, the counterpart to natural language analysis. In fact, after this realization was made, a companion program to PHRAN, named PHRED[Jac83] (**PHR**asal **E**nglish **D**iction) was written and was successful in using the *same* knowledge base that PHRAN used for its analysis.

Another advantage of pairing linguistic forms to concepts was that if the concepts did not rely on English (and technically they were not supposed to in CD even though labels in CD were always written in English), the mechanism of PHRAN could be used to analyze other languages. Once the analyzing had been done using a set of pattern-to-concept pairings (known in PHRAN as *Pattern-Concept Pairs*) for one language, say English, a set of pattern-concept pairs for another language, say Spanish, could be used to generate the same utterance in Spanish. Hence, depending on the strength of the underlying representation (i.e. that it be canonical and interlingua or language independent), PHRAN and PHRED together formed a primitive language translation system. In fact, work has been done in making PHRAN and PHRED understand languages such as Chinese and Spanish[WM81], [Jac83].

A sample Pattern-Concept Pair taken from PHRAN's database is shown below

```
(index-under-pattern (kick the bucket)
 [(nil
   [(root kick) the (* and bucket)]      ;; pattern
   [p-o-s 'verb                          ;; links to concept
    tense (value 1 tense)
    root 'kick-bucket
    voice 'active
    form (value 1 form)])])
```

This pattern is the one used to analyze 'kick the bucket' using any tense of the verb 'kick'. The concept analyzed into is shown as a "kick-bucket". Actually, "kick-bucket" is the name of a special type of pattern used in PHRAN called a *named group*. A named group is to be thought of as a meta-pattern or a macro pattern representing the concept of a Pattern-Concept pair which expands into a regular pattern when encountered in an instance of a "`(index-under-pattern ...)`" as above. The idea was to save typing for patterns that built the same information.[6]

The named group (and thus, the real concept) for *kick-bucket* is shown below

```
(name @kick-bucket
      ((active)
       [(person) (root kick-bucket)]
       [concept '(state-change (actor ?actor)
                               (state-name health)
                               (to -10))
        actor '?subject]))
```

Basically, PHRAN processes its input by reading in the sentence one word at a time left-to-right and matching the input against its patterns. As it analyzes, it replaces individual words by *terms* which, based on what patterns the terms match, may be combined together to form more abstract *terms* later on in the analyzing process. We note that once a term is created, PHRAN does not backtrack to undo a term's creation. PHRAN's inability to deal with possible multi-parses (i.e. creating different sets of *terms*) is a problem which will be examined in the next section.

2.1.4 Problems with PHRAN

There were a number of problems with PHRAN. Most of these problems concerned the implementation of PHRAN, not the theoretical ideas. While PHRAN had succeeded in being used for several purposes, including being used for different languages and being the front-end of the original UC, it became difficult to maintain the code. Also, in my experiences with PHRAN, problems showed up that *needed* correction.

Consider the problem of adding patterns to PHRAN. For example, a typical question asked of a Unix Consultant may be `How can I send a message`

[6]Named groups, as well as other details regarding Pattern-Concept Pairs for PHRAN are explained in Arens' unpublished "How to Write PHRAN Patterns" [Are81]

to someone?. To analyze this question, we need to write the pattern [(person) (root send) (message) {from (user)} {to (user)}], where the braces ({}) indicate optional parts of the pattern, and attach this pattern under a "named group" for **send**. As mentioned above, a named group in PHRAN was a type of meta-pattern macro that was meant to expand when referenced inside a PC-pair. The idea was to save a lot of rewriting for patterns that built the same information.

A problem comes up, however, in writing a pattern for the sentence **How can I give a message to someone?**. We have to write almost the same pattern as we had in the *send* case so that it could be stored under the word **give**. In other words, we are not able to generalize the "Distribute message to someone" construct because of PHRAN's rewrite rules. We have to duplicate the pattern whenever we wanted to add the ability to analyze a variant of it. In other words, with this implementation, PHRAN still seems too word oriented.

There were other major problems with PHRAN not having anything having to do with re-write rules. Redundancy arose in the pattern database. Whenever a pattern writer wanted to add a pattern for a new sentence, s/he had to specify everything in the pattern. For example, the pattern had to contain what PHRAN called *links* to specify which part of the pattern denoted the actor, which denoted the object, etc. This information would have to be duplicated in every pattern representing sentences between actors and objects[7].

Another problem was that patterns were not as autonomous as they were supposed to be. For example, consider the questions **How do I find out how much disk space I have?** and **How do I find out how much disk space I have used?**. At first glance, it seems all one needs to do is create a Pattern-Concept Pair where the pattern [is 'how much NOUN PERSON HAVE']. However, when PHRAN sees 'ADJECTIVE NOUN', it calls the pattern that replaces the above two terms into the single term 'NOUN-PHRASE'. This means that PHRAN collapses the terms [<how> <much> <N>] to [<how> <NP>] which causes it to miss the [<how> <much> <N> <person>...] patterns. Even though the specific word **much** is included in the pattern, PHRAN uses

[7] As we will see in the description of my analyzer, my solution to the problem of duplicating information could have been integrated into the old PHRAN by having patterns correspond to different *levels* of analysis. The question is, would PHRAN's pattern matcher have handled things the proper way. Some of the problems discussed later indicate PHRAN might have had trouble with patterns being matched in parallel.

the more general pattern, thereby violating the important analysis principle of using the most specific patterns first[WA80]. Furthermore, even if this PC-pair is indexed under (how much), the problem occurs. The reason for these problems is that PHRAN does not do arbitrary backups because of the difficulties involved. PHRAN not backing up had not caused troubles in the past since pattern writers had always been able to write patterns that got around backing up.

The general problem was that one could not just add a new pattern as a discrete piece of knowledge. One had to know how the new pattern would be matched, how backup might or might not be performed, and most importantly, how it interacted with other patterns. One of the main goals of my analyzer is that any pattern that gets added to its knowledge base will get instantiated if it matches the input, and finding such patterns is not done with merely ATN-style backup, but in a parallel style to be described later.

PHRAN's syntax for pattern-concept pairs is painful, especially when one wanted "optional" parts for a pattern. An example of a rewrite-rule such as what a pattern writer might have to write (along with optional subparts) is included below[8]:

```
; thing be eaten
(index-under-pattern (noun-phrase be perfective)
   (get (from-end 1 root))
   (3 (root be) (* and 2 (form perfective) (negative nil))
      ([(by 1) (actor (opt-val 2))])
      @rest ([(by 1) (actor (opt-val 2))]))
   (cd-form '?concept
      p-o-s 'sentence
      subject nil) passive)
```

The first line of the example (index-under-pattern . . .) indicates where this pattern is to be indexed. The idea was that instead of suggesting all patterns all the time, a pattern would be suggested only when enough of it had been seen. N.B. the index is not the same as the pattern, but in almost all cases it was a prefix of the pattern. The patterns were indexed under a *Discrimination Net*[CRM80].

The next three lines of the example indicate the pattern[9]. These lines have a very complicated meaning which depends on the fact that at pattern-match

[8] This pattern is taken from the PHRAN database with the '#' signs changed to '@'.

[9] See Yigal Arens "How to write PHRAN patterns", §4.3 for details.

time, this pattern will be accessing various "named groups". The **(get (from-end 1 root))** indicates that a named group is to be taken and modified at pattern-match time, where 'get' is a special function which finds desired PCPs. Because of the state of the system when this pattern is being processed, the '1' refers to the verb that it is looking at (the pattern writer has to know it is a verb because we have reached the 'perfective' in the pattern index). Then a new named group gets created. The third, fourth, and fifth lines of this example, then, are built in to be the parts of the pattern that go into a new meta-pattern (or named group). In other words, while the concept itself is a declarative (empty frame) structure, this actual piece of knowledge is procedural in that it builds, during the pattern matching, a new named group that will be used by other pattern-concept pairs.

Looking at the second and third lines (which are the skeleton for the new pattern), we see numbers, a '*', and a '@rest'. The numbers refer to components in the pattern being built up. The **(* and ...)** indicates to the pattern matcher just where the indexer will have left off on this particular pattern. The *and* in this starred expression is to be thought of as a boolean 'and'—not the word **and** (even though it is put in the list just as if it were a pattern component). The last expression beginning with 'cd-form' is the concept part, which in this case just amounts to setting a few links since the concept will be stored in the variable '?concept' when the subpattern matches.

All in all, this is much too complicated (I do not expect my very cursory description to have been able to make anyone understand just what PHRAN is doing). The main problems are that these meta-patterns are not really necessary if one has what I call 'high-level' patterns that match at the same time as the 'low-level' patterns. The low-level patterns are meant to match the least abstract phrases, "passing" information upon their instantiation to the high-level patterns. The main difference between "high" and "low" level patterns is the level of abstraction each exhibits. So, for example, a fairly 'high-level' pattern is 'NP → Art NP', which would match anytime one sees an NP preceded by an Art. This will not always result in the correct analysis right away, but all patterns are matching at the same time. Those patterns that do match, but turn out not to be correct may be dead-ended in their usefulness when other understanding components rule them out. These other processes can indicate to the analyzer that certain patterns should no longer be considered. We shall see later how the analyzer interacts with the other understanding processes when the actual pattern matching process is described.

Therefore, all linguistic patterns can be written as rules, instead of per-

forming special tricks to get meta-patterns to come out correctly (as shown in the above example). Thus, the complexity of meta-patterns (or rewrite-rules or named-groups) can be avoided if one uses a single representation for patterns. Then simpler pattern matching than PHRAN's can be used to analyze the input.

Additional unnecessary complexity can be seen in the optional parts of patterns. The optional sub-patterns are indicated by the extra brackets and require that their links be set in the pattern itself (as opposed to being in the concept part, which is where non-optional links get set). For their added complexity (both in pattern writing and especially pattern matching), they are not worth having. Their exact same effect can be had by having separate patterns.

Another problem with PHRAN had to do with the way it fit into the rest of an understanding system. PHRAN was inadequate for KODIAK since it had been designed to create and instantiate only Conceptual Dependency frames. The idea was that when PHRAN finished its processing, a completed CD representation would be returned by PHRAN. Only after PHRAN had finished would other understanding components be able to take over to further process what PHRAN had given them.

My feeling is that the "pipe-lined" approach to natural language understanding implied by PHRAN-style analysis is wrong since the grammaticality of the sentence is heavily influenced not only by the semantics of the sentence, but also by the *understanding* process of that sentence. To have a linguistic analyzer be the only component used to build a representation means that the analyzer must know all about metaphors, metonymies, and categories so that the grammaticality of basic sentences may be determined.

Given the above problems, it was clear that even though the ideas of PHRAN were essentially correct, the implementation had problems serious enough that they could not be simply patched.

2.2 KODIAK

One of the major reasons PHRAN needed to be rewritten was that the representation language, Conceptual Dependency[SA77] (CD) was inadequate. PHRAN was designed to create such CD frames and instantiate them by filling in their slots. The idea being that when PHRAN finished its processing, a completed CD representation would be returned. Arguments for why CD's are not strong enough for representation are detailed in Wilensky's paper[Wil84],

but a basic summary of the arguments is that the CD frames are too ad hoc, and the frames' slots were unmotivated. A new Berkeley representation named KODIAK[10] has been developed to address their deficiencies. KODIAK is similar to KL-ONE in that it is a semantic network with multiple inheritance. I will not explain the motivations behind the details of KODIAK. They can be found in Wilensky's paper[Wil84]. I will, however, give a brief description.

Objects in KODIAK are either *absolutes* or *relations* (which are used to relate two or more objects together). Objects can represent classes (such as the class of all books) or can be subclasses of other classes (such as the class of females being a subclass of humans). This type of subclass relation is denoted as a *dominates*. It is represented in KODIAK by a *dominates* (or D) link.

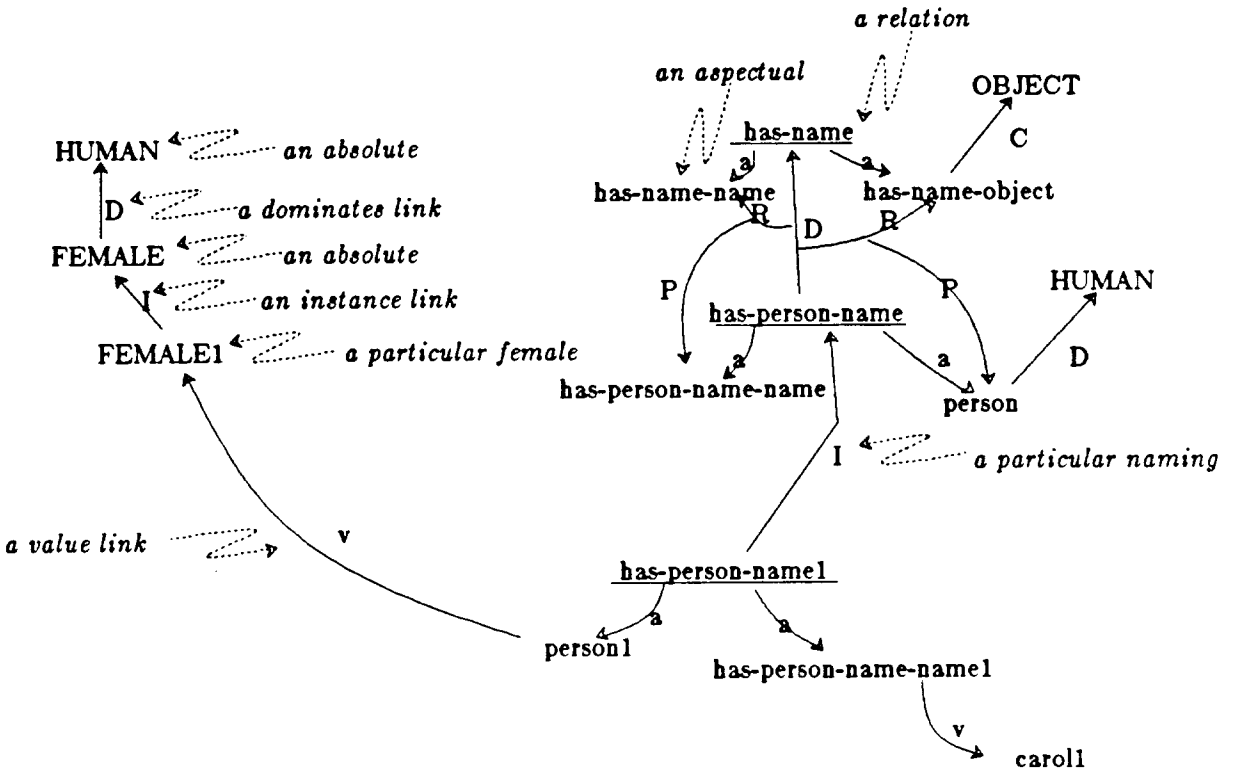
Relations can be denoted as subrelations of other relations also by using *dominates*. Relations have two or more *aspectuals* where an aspectual is a concept equivalent to the argument position of a relation. Inheritance of aspectuals is then indicated by a special two-pronged link known as a *Role-Play* link. A role-play link is associated with a particular *dominates* link and has the role part pointing to the "parent" aspectual, and the play part pointing to the associated "child" aspectual. All other aspectuals of the dominator's relation get inherited by the dominatee exactly as they are in the dominator. Finally, one can denote particular *instances* of a class by using an *instance* (or I) link. Likewise individual instances of relations can be formed by using an instance link. Examples of these KODIAK concepts are shown in Figure 4, which shows how we can indicate that a particular human female exists and is named Carol.

2.3 ALANA's Knowledge

At this point, we look at what ALANA's knowledge looks like. Later we will examine in detail how the actual analysis is done and how representations get built. The heart of the ALANA's matcher, the chart parsing process, is described in § 3.1. The main idea behind ALANA is that pattern knowledge is used to aid in the syntactic analysis, which takes place at the same time as semantic analysis. ALANA's main function is to use the patterns to suggest parses. All pattern matching knowledge, from the individual words up to syntactic rules are handled as Pattern-Concept-TEST triples (sometimes just simply called *defpats*).

As we go through the description of the data with which ALANA works,

[10] Keystone to Overall Design, Integration and Application Knowledge.



A Section of KODIAK indicating a female's name is Carol

Figure 4: KODIAK examples

we must keep in mind ALANA's overall understanding strategy. Any pattern that can be matched *will* be matched. It is not necessarily just up to the pattern matcher to determine which matches are to be ruled out. ALANA expects that other routines will look at what has been matched and that other processes running as coroutines will suggest information that can aid the matcher. This information is in the form of adding or removing patterns being considered, thus affecting the way pattern matching will be performed.

ALANA's matcher provides tools to add or remove patterns being considered. These tools can be used by other inference components to provide *hints* to ALANA's matcher. In this way, analysis expectations can be added, or known dead-ends can be avoided. While current systems which use ALANA

do not take advantage of such tools, their intention is to have language analysis be a subservient part to the whole of the understanding process. In this way, ALANA can run as a coroutine with other components in an understanding system and it may be desirable for these other components to directly affect how sentences are being read by ALANA.

The basic form of ALANA's data is

(**defpat** [*pattern-label*] (*pattern-name* => $component_1$ ... $component_n$)
(TEST (LISP-FCN $component_1$ ... $component_n$))
(**builds** (LISP-FCN $component_1$ ... $component_n$)))

The *pattern-label* is an optional atom which distinguishes this particular PCT-triple and is used only as a mnemonic for the user. The *pattern-name* is an atom which is the label put on the pattern instantiation created when the components ($component_1$ ... $component_n$) have all been matched. In the simplest form, each component is an atom[11], and refers to a *pattern-name* of a defpat. A word is indicated as a list (w *word*) so that ALANA can know which pattern components should be passed off to the spelling corrector when an unknown word is encountered. During analysis, after the components are found, the *TEST*, if present, is evaluated and if it returns non-nil, the result of evaluating the *builds* function is attached to a new pattern instance for pattern-name as the (**concept-of** *pattern-name*).

*TEST*s are meant to be used for linguistic tests such as agreement, whether prepositions can modify a noun-phrase, and other forms of grammaticality known about at the language analysis level. *TEST*s are given entry points to the parsing chart so that the *TEST* function can look around on the chart for any information it may need. Most morphological analysis, however, is done on the word of the chart edge which can be accessed via the *word-of* function.

The *TEST* and *builds* parts of the defpat are described in LISP functions of components. In processing these parts, the LISP-FCNs for the *TEST* and *builds* are applied to the values of the concepts of the pattern components. Because of the way these values are stored (attached to chart edges on a parsing chart—to be explained later), functions can not only access the components directly, but through them, can access any previously built up concept value. The way to access the different concepts is through chart accessor functions. By using the chart accessor functions, the *TEST* and *builds* functions can look in other places in the chart to gather any needed information.

One danger of using arbitrary functions for *TEST* and *builds* is that analysis could become too procedural and as such, it could become impossible to

[11] Actually a component can also be a list. We will discuss later what this means.

trace or keep straight just how the analysis takes place. It is for this reason that certain strict conventions have been adopted. One of these is that *TEST*s should not modify the chart or assert any new conceptual information. They only check conditions and return nil or non-nil. A nil result means the *TEST* failed and not only should the concept (*builds* function) not be asserted, but neither should any new chart edges be added. The other convention is that, as much as possible, concepts should be written in a declarative style using macros that expand to the low level KODIAK primitives. Let us now look in detail at how these macros are defined and how concepts are processed.

2.3.1 Builds (Concept) Processing

Here are some actual sample defpats:

```
(defpat (V => Print))

(defpat (Print => (w print))
  (builds (i PRINT-ACTION
             witha pr-effect (i PRINT-EFFECT))))
```

These two defpats contain patterns, with the first having neither a *TEST* nor a *builds*, and the second having no *TEST*[12], but with a simple *builds*. The first pattern says, whenever a 'Print' category is found, always instantiate it as a verb (V). We note that though the pattern matcher will always put in a V wherever it sees a Print, this may not always be desired, and that there are cases where 'Print' ought to refer to a Noun (as in a 'Print-out'). To rule out such bad possibilities, we could rely on either having a *TEST*, which would perform a linguistic test to see if it were possible to have a N here, or we could just go ahead and assume that 'Print' is always a Verb. If it is not a Verb, a later process will use one of the functions provided by the analyzer to remove it from further consideration as a Verb. In neither case do we rely on just the syntax alone. This is a place where we would call on semantics to aid in the pattern matching.

The second defpat actually associates a word with a category that gets used in the first defpat. Since it also has no *TEST*, it will always be instantiated whenever the word `print` is seen. In addition, it uses the *builds* function to build the parenthesized expression `(i PRINT-ACTION witha pr-effect (i PRINT-EFFECT))`. This expression is read as

[12] A sample defpat with a TEST appears in an example on page 27.

Create an instance of a PRINT-ACTION object (where we assume PRINT-ACTION is already defined) and assert an instance of a pr-effect for that PRINT-ACTION to be an instance of a PRINT-EFFECT.

The i expression is actually a LISP macro and is defined so that the above expression will expand to the following LISP code calling KODIAK primitives

```
(progn

   (push (relation (new-unbound-sym 'PRINT-ACTION))
      *REL-namestack*)
   (instance (relation 'PRINT-ACTION) (top *REL-namestack*))

   (push (aspectual (new-unbound-sym 'pr-effect))
      *ASP-namestack*)
   (argument (top *REL-namestack*) (top *ASP-namestack*))
   (role-play (aspectual 'pr-effect) (top *ASP-namestack*))
   (value (top *ASP-namestack*)

      (progn
         (push (absolute (new-unbound-sym 'PRINT-EFFECT))
            *REL-namestack*)
         (instance (absolute 'PRINT-EFFECT)
            (top *REL-namestack*))
         (pop *REL-namestack*)))

   (pop *ASP-namestack*)

   (pop *REL-namestack*)))
```

which is equivalent to the following series of KODIAK calls (the '#' means that a new atom is created by using newsym).
Make a new instance of PRINT-ACTION called PRINT-ACTION#

```
(relation 'PRINT-ACTION#)
(instance PRINT-ACTION PRINT-ACTION#)
```

Make a new aspectual called pr-effect, and make it be an aspectual of the newly created PRINT-ACTION

```
(aspectual 'pr-effect#)
(argument PRINT-ACTION# pr-effect#)
(role-play pr-effect pr-effect#)
```

Give the newly created aspectual, the value of the inner 'i' macro which will be the new PRINT-EFFECT#

```
(value pr-effect#
```

(Inner block) Create a new instance of a PRINT-EFFECT which will be the value of the pr-effect# above

```
(absolute 'PRINT-EFFECT#)
(instance PRINT-EFFECT PRINT-EFFECT#))
```

Graphically, the new network looks like Figure 5.

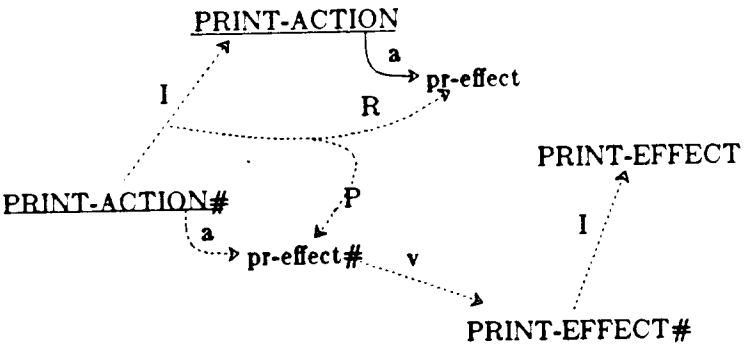


Figure 5: KODIAK PRINT creation

More options in the *builds* part of defpat are illustrated by the following (partial) example (for clarity, the words here are left unmarked)

```
(defpat (Q => do you know how to S)
   (builds
      (old (concept-of S) is1 HYPOTHETICAL
         with cause (i ACTION is1 HYPOTHETICAL
```

```
                    witha actor _actor))
.
.
.       ))
```

The ellipsis indicates that there is more that one expression in this **builds**. **builds** itself can be thought of as a LISP **progn** so that the last expression's value is what will be returned and put on the concept annotation for the **Q** chart edge.

The **(old (concept-of S) ...)** means refer to an already created concept rather than create a new one. Saying **X isi Y** means **(instance Y X)**. Finally, the aspectual *cause* is referred to by '**with**' as opposed to '**witha**' in the previous example. This means that the aspectual **cause** for (concept-of S) is inherited from a higher-up concept (presumably *causal-event*) rather than have a new instance of that aspectual be created. S, by the way, could refer to a concept such as DELETE-ACTION17 (an instance of a DELETE-ACTION which may have been created by the sentence "Do you know how to delete a file?"). The KODIAK calls for the above examples are

```
(relation 'ACTION#)
(instance ACTION ACTION#)

(aspectual 'actor#)
(argument ACTION# actor#)
(role-play actor actor#)
(value actor# _actor)

(instance HYPOTHETICAL ACTION#)

(role-fill (concept-of S) cause ACTION#)

(instance HYPOTHETICAL (car *REL-namestack*))
```

which graphically looks like Figure 6.

Other options to the *builds* macros include referring to the most recent instance of a concept by **(newest** *concept*). Also, one can indicate a *dominates* relation by using **isa** as in **(old** *concept* **isa** *concept-class*) which expands to **(dominates** *concept-class concept*).

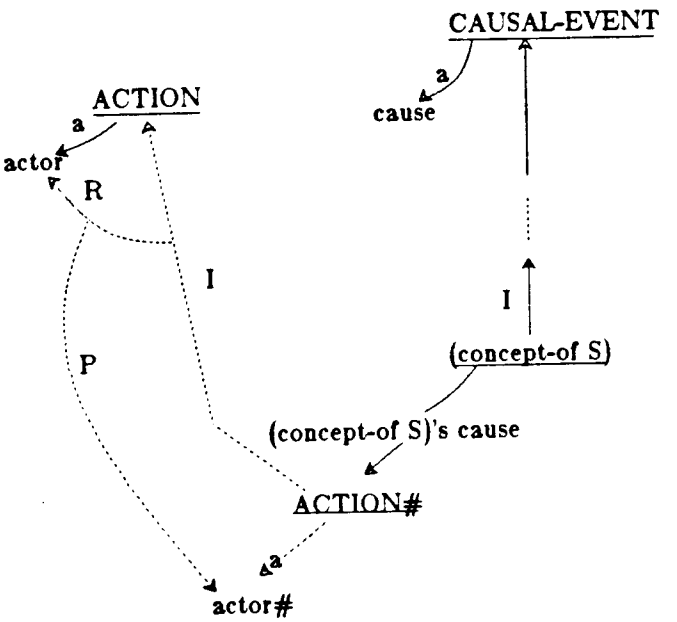


Figure 6: KODIAK Causal Inheritance

2.3.2 TESTs processing

At the time of this writing, not much study had been put into *TEST*s and very little has been implemented. What is implemented for them is the automatic checking for them. If a *TEST* exists and it does not return non-nil, the chart edge corresponding to the pattern-name is not added. Here is an example with a *TEST*.

```
(defpat (S => Person Trans-VP)
   (TEST (number-and-person-agreement Person Trans-VP))
   (builds
      (old (concept-of Trans-VP)
         witha actor (concept-of Person))))
```

Just as in the *builds* discussion the *TEST* function, in this case `number-and-person-agreement` is applied to the chart edges corresponding to the matched Person and Trans-VP. If the *TEST* is successful (i.e. returns non-nil), then the *builds* is evaluated and stored on S's chart edge. Otherwise, S is not added to the chart, and this pattern will be discarded from further consideration.

2.3.3 Miscellaneous defpat options

There are a few other features I have added to defpat to make pattern writing slightly easier. One option is the ability to assert conceptual information without adding a chart edge. An example of when you might want to do this is when you have matched a modifier pattern whose *builds* modifies an existing KODIAK concept. It may not be desirable to add a new edge representing that concept since it would only redundantly fire off many patterns that had previously been triggered the first time that concept's edge was inserted into the chart. The way to achieve this option is to put a minus sign (-) in front of the pattern-name. Here is a specific example.

```
(defpat (-NP => NP named *anyword*) ;; add a name to NP
   (builds                          ;; without adding any
      (i HAS-NAME                    ;; new chart edges
         witha named-obj (concept-of NP)
         witha name (old (absolute (word-of *anyword*))
                         isi NAME))
      (concept-of NP)))
```

In this defpat, we are building a HAS-NAME relation which relates an object (denoted by `(concept-of NP)`) to a NAME (which is matched by the wildcard `*anyword*`). `*anyword*` matches any *word* from the input. There is another special pattern component called `*unkword*` which is supposed to match any unknown word. This is useful for patterns that look for components such as names which may not be in the analyzer's dictionary, but are still expected. I say "supposed to" because deciding if a word is unknown or if it fits into the pattern is very complicated to figure out since it is very likely that known words may be used to name objects (such as *file* in the phrase `the file` *file*). Thus the code for handling `*unkword*` is currently exactly the same as that for `*anyword*`.

Another defpat option is the ability to define multiple patterns within a single defpat. This is done as follows

```
(defpat (name1 => name2 => ...  => namen => c1 ...  cm)
  (builds ...  ))
```

which is exactly the same as writing the following n patterns:

```
(defpat (namen => c1 ...  cm)
  (builds ...  ))

(defpat (namen-1 => namen)
  (builds (concept-of namen)))

    .
    .
    .

(defpat (name1 => name2)
  (builds (concept-of name2)))
```

This option's use is illustrated in the next example, which also shows how one can access the results of a partially completed pattern outside of the current pattern being matched. This is done by setting global variables during the matching phase. A variable (atom) is included as part of a pattern component. If that component is matched, the atom's value is set to the concept associated with that component's edge. For example, a pattern exists in the UC pattern database as follows:

```
(defpat (S => Intrans-S => (_actor Person) Intrans-VP)
   (TEST (number-and-person-agreement Person Trans-VP))
   (builds
```

```
        (old (concept-of Intrans-VP))))
```

This defpat says that if Person is found in the course of matching this pattern, the global variable `_actor` is set, and can be used in other patterns such as the *builds* part of

```
(defpat (Intrans-VP => VP => get Phys-Ailment)
   (builds
      (i HAPPENS-TO
         witha patient _actor      ;; set patient to global
                                   ;; value of _actor
         witha condition (concept-of Phys-Ailment))))
```

Here the S pattern has not been completed yet, but the actor is already known and can be gotten by the VP pattern (subpattern of S). Using an underscore to prefix variable names is simply a convention (borrowed from PROLOG) for naming variables.

2.4 Sharing Linguistic Knowledge with the Generator

One of PHRAN's greatest strengths came from its declarative style knowledge base. This style of knowledge base allowed PHRED to be built and run using PHRAN's linguistic knowledge. Part of the reason PHRAN was able to use this declarative structure was that all PHRAN had to do was build frames, which corresponded to LISP lists. In the KODIAK world, things are different in that the knowledge that gets passed around from one routine to another is not in the form of LISP lists, but rather in the form of pointers into a giant semantic network representing the state of the program's understanding of current situations. Moreover, analyzer knowledge is not the knowledge to just plug fillers into slots. The analyzer has to dynamically create instances of concepts and relations and leave pointers to these newly created objects.

My solution to building KODIAK structures fitting into the main semantic net, and at the same time, have this knowledge structured declaratively, has been to impose a strict style on the *builds* functions and have them be *interpreted*. For my analysis needs, up to now, this format has been fine. However, if a concept needs to be built that does not follow a more or less hierarchical form, i.e. having something like

```
(i X
   witha (i Y witha . . . ))
```

then one has to resort to using some kind of trickery such as declaring local variables. Resorting to such trickery loses the flavor of the strict declarative format (even though the analyzer will still work). Furthermore, such trickery diminishes the hopes of getting a generator up that can use the same knowledge.

A proposal, by Dekai Wu, has been made that the concepts of pattern-concept pairs should be represented as pointers to template sections of KODIAK networks. These templates could be filled out by an interpreter which would take links set up between pattern components to pointers into the template and instantiate that template into the main KODIAK network.

What is nice about these forms is that they allow much more generality than my *i* macro would give, plus they could be shared with the generator. Wu's proposal is appealing. Making such a modification to the current ALANA should be relatively straightforward since building the concepts is centralized to one function.

3 ALANA's Processing

ALANA processes a sentence one word at a time, left to right. The analyzer is highly integrated with memory and refererence systems so that other processes that use the analyzer will be able to make analyzer-time inferences without waiting for the analysis to finish. An example that has come up in UC involves the question `How do I print a file on the line printer?`. The other components of UC depend on the fact that after the analysis, an instance of a PRINT-FILE-ACTION concept be created with the appropriate denotation that the destination is the line printer. When reading the sentence, after we have seen "How do I print a file...", it is expected, at analysis-time, to have this PRINT-FILE-ACTION. It is not reasonable, however, for the analyzer to generally know that a PRINTing of a file is any different from PRINTing anything else, so all the analyzer produces is a PRINT-ACTION relation with a print-file being a FILE node. At this point, however, it is reasonable for a concretion[13][Wil83] mechanism to take over and infer that a PRINT-FILE-ACTION needs to be created. The analyzer can then take this PRINT-FILE-ACTION instance and associate with it the fact that it is an action directed to the line-printer.

[13] Concretion is the process whereby one can realize a more specific concept given that one already knows a set of more general concepts.

Ultimately, ALANA should be set up as a coroutine so that the rest of the system can make complex inferences while ALANA is still processing. This phenomenon also happens in people when they are able to answer a question or finish someone else's thought before processing the whole sentence being input to them.

ALANA simulates this ability by asserting facts in the knowledge base as it finds them out. Of course, much of the information may become invalidated if later parts of the sentence contradict inferences made in the earlier parts. In an ideal system, however, these inferences could be retracted or avoided just as a person would do when reading text in the ALANA style. ALANA can be set up to allow other parts of the system to add or delete language analysis facts, i.e. chart edges or annotations associated with these edges. In this way, ALANA can allow other parts of the system the ability to provide knowledge that it needs in order to do its analysis.

Currently, ALANA is not running as a coroutine since it has not been integrated into a system which supports language analysis as a coroutine. I believe, however, that adapting ALANA to a coroutine system would be relatively straightforward. The whole analyzer runs as two main loops which would need to be broken up so that state could be saved as the other components operate. Until a whole non-pipelined understanding system is set up so that ALANA can run on an agenda or coroutine basis (where it would fetch input and build simple concepts), it can and is being used to pattern match input against patterns and to use its patterns to create partial or complete concepts that the rest of the system can use for the other understanding or planning tasks. Also, after each concept is built, whether it be major or minor, a concretion mechanism (being built by Dekai Wu) is called to further concrete the concept if necessary.

3.1 Chart Parsing

We now concentrate on how the pattern matching part of the analyzer works. A basic idea I wanted to incorporate into the analyzer was that any pattern in the database should match whenever all the components were present in the input and in the right order. In the extreme, patterns could be written in such a detailed and low level way that there would be one pattern for every sentence that the analyzer could understand. My analyzer does not rule out this way of

storing patterns[14]. Yet, since it instantiates a pattern whenever that pattern is seen, we can have different 'levels' of patterns. High level patterns would be, for example, general syntax rules or language definition rules. A "language definition" rule is what I am considering to be among the most abstract of the high-level patterns. These are the ones which may declare a language to be, for example, an SVO (Subject-Verb-Object) language. Lower level patterns would be fixed word expressions such as idioms and collocations (or discontinuous collocations). The lowest level would be forms for words or lexemes. This type of pattern matching is handled using a data structure known as a *chart*.

The need for a chart comes up in the following simple example:
Given the grammar

The "high-level" patterns
NP → Art Nominal
Nominal → Adj Nominal
Nominal → N

The "low-level" patterns
Art → the
Adj → big
Adj → red
N → ball

The correct and only parse for the unambiguous phrase `the big red ball` is pictured in Figure 7. We need to avoid all patterns getting instantiated whenever all their components appear but are not properly positioned, as indicated by the dotted lines in Figure 8.

What is needed to match all the possible patterns is a way to keep track of where the matched components are relative to all the other components. An ideal data structure for representing this situation is a *chart*.

Like a tree, a chart is a graphical representation of an input string parse. Unlike a tree, however, a single chart can represent multiple parses of a sentence. In fact, a chart is a generalization of a tree in that it can represent all possible parses of the string along with all possible parses of substrings.

[14]In fact, I needed to rely, to some degree, on this method of storing patterns for UC patterns since I did not have the benefit of a concretion mechanism working with the analyzer to aid in building the concepts that were needed by the rest of UC. Thus some of my patterns are more "canned" than they should be right now.

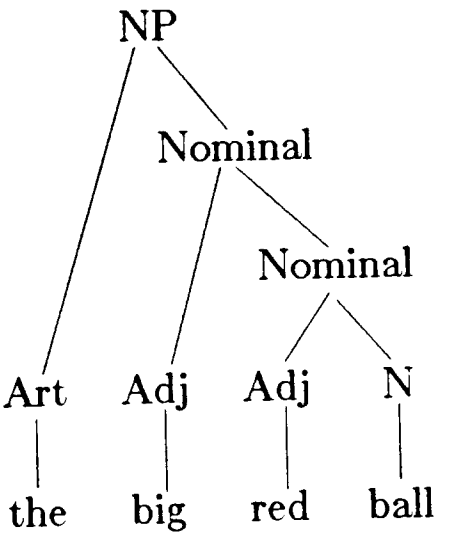


Figure 7: Correct Parse

Because of this flexibility of being able to represent multiple parses in a non-mutually conflicting way, I have chosen to use a chart as the basic data structure in my analyzer.

A chart is simply a labelled directed graph. Nodes are meant to be thought of as markers between the original words of the phrase. The edges indicate pattern instantiations with the edge labels being the name of the pattern instantiated. For convenience, the words themselves form edge labels on edges to indicate a linear ordering of the nodes. An example of the "initial" chart, i.e. one that has not had any patterns matched against it is shown in Figure 9.

The patterns, then, are matched against the initial chart by looking at the labels on the edges. An edge indicates all the nodes that are included in a particular match. In the simple case, each word belongs to a category, so an "intermediate" chart might look like Figure 10. Note that from a given node, one has access to all the edge labels coming from that node. So, for node 2, we can access the word `red` just as easily as we can the part of speech (or pattern label) `Adj`. This feature makes matching all levels of patterns easy and is not available if we just are building a strictly hierarchical tree structure.

In the case where a pattern matches two or more components as in Nominal → Adj N, a single 'Nominal' edge is added that starts at the initial node of the first component edge and ends at the final node of the last compo-

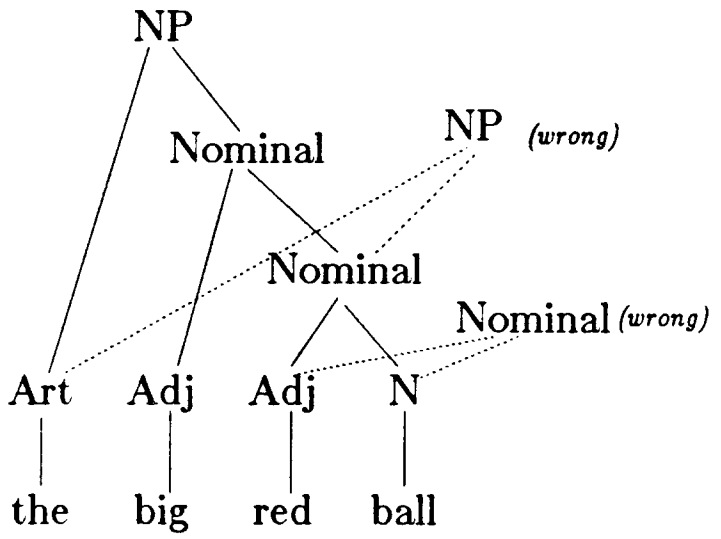
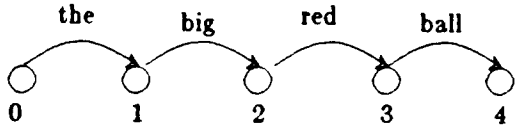


Figure 8: Bad Parse

nent. Thus the next intermediate chart containing the instantiation of the Nominal → Adj N pattern may look like Figure 11.

The final chart using the above grammar for our example is in Figure 12.

We look back at Figure 8 and see that the wrong grammar rule instantiations are not possible in the chart since they represent edges which skip nodes. Not only are wrong grammar rule instantiations not possible, but we have the benefit of accessing all possible parts of the instantiated patterns from each node.



Vertices are numbered with words labelling the edges. The nodes are labelled for ease of reference.

Figure 9: Initial Chart

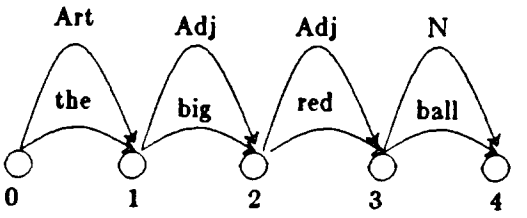


Figure 10: Intermediate Chart

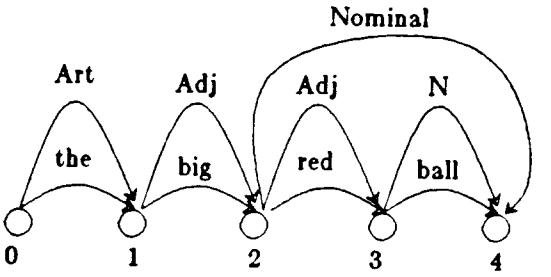


Figure 11: After Nominal → Adj N

3.2 Matching Defpats

Let us look again at the basic form for a pattern-concept-TEST triple to see how they can be matched against the analyzer input. Recall the basic form of a defpat:

(**defpat** [*pattern-label*] (*pattern-name* => $component_1$ ... $component_n$)
(TEST (LISP-FCN $component_1$ ... $component_n$))
(**builds** (LISP-FCN $component_1$ ... $component_n$)))

A pattern matches if each $component_i$ is present and adjacent in terms of the chart that gets built up. When each component is found to be present, a new chart edge gets added starting at the initial vertex of $component_1$ and ending at the terminating edge of $component_n$. So, if the chart looks like Figure 13, the pattern-name is then added as the label of a new edge representing a new component as in Figure 14 leaving all other edges in place. The

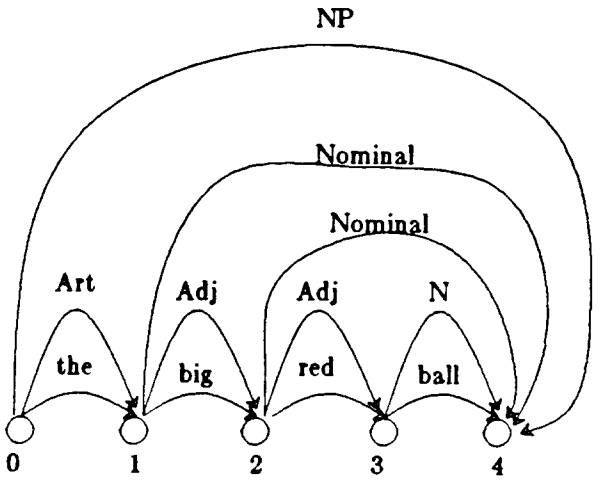
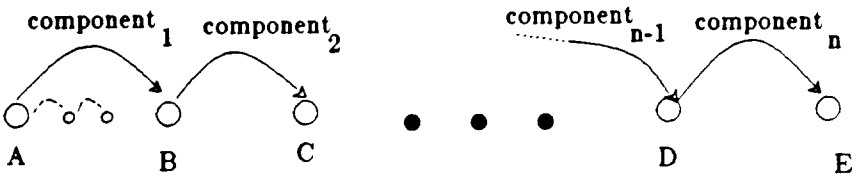


Figure 12: Completed Chart



Dotted edges indicate arbitrary edges

Figure 13: Sample Chart

pattern-name may be then just the component that a higher level pattern is looking for.

Each *component*$_i$ in the defpat refers to exactly one chart edge and so that chart edge can have extra information or annotations associated with that component. More importantly, using the chart edge, we can have an entry point to the chart and through it, we can get to as much arbitrary information about the parse as we may need in concept and *TEST* processing. For example, if we matched a NP, and we wanted to know what the words of that NP were, we easily have access to that information by following all the appropriate edges of the chart vertices between the beginning and ending vertices of the NP edge. In processing a *TEST* or *builds*, one may need access

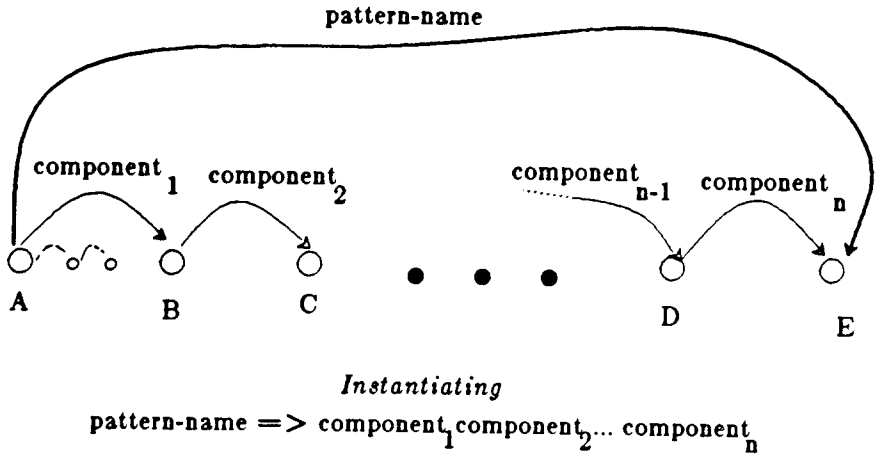


Figure 14: Sample Chart with new edge added

to arbitrary pieces of information from the already built chart. We will now look at *TEST* and *builds* processing.

3.3 Left to Right Parsing

Most computer language parsers read its input (i.e. source program) by reading one lexeme at a time and building parse trees. At the lowest level, the input is read one lexeme at a time, and although processing on that lexeme may have to wait until later input is read and processed, no lexical item is read until all the lexemes to the left are read. Likewise, most natural language analyzers read one word at a time where the leftmost words will set up expectations for the words further to the right. A case where a word sets up an expectation for input already read in is "NATIONALITY restaurant", where only after reading **restaurant** would it be reasonable to look to the left and check if the "previous" word refers to a nationality.

People, however, do not necessarily read one word at a time. Speed reading courses teach students to read down the centers of pages using peripheral vision to see words along the sides. The eyes (which are peoples' input sensors) do not need to move across each word. A machine, however, with only a single read mechanism, needs to ultimately read each character of each word.

My belief is that people, when clumping together recognized phrases, read

faster and more efficiently than when they are constrained to reading one word at a time. Thus if one were to write a analyzer that reads as people read, that analyzer would have to have (in addition to some sort of visual capability) an ability to recognize phrases quickly that jump out as units to people. In this way, individual words can be skipped, or given less analysis than other words. This type of analyzer would go way beyond analyzers that have been built in the past, and would require several years of research in vision and perception as well as in natural language recognition.

Yet, people are still able to read one word at a time since those, who know how to read, are able to process the news sign at New York's Times Square or the light signs at BART stations which have words "traveling" across the sign. The words start at the right edge of the sign and stay visible as they move across the sign until they disappear off the left edge. Words are presented in order, at a constant rate, where the left-most word of a passage will appear first. People reading the sign then have to be able to read and analyze the sentence one word at a time. If we are to write a natural language understanding system that is limited in how it collects its input (i.e. one word at a time), we should strive to simulate how *people* read input one word at a time, and how expectations and concepts are built.

For me personally, one word at a time type of reading is slow and can be very frustrating. I can sense expectations being set up so that the need for reading the rest of the words in a phrase becomes somewhat reduced. I am only looking for a few more keywords to finish the idea. It is this idea of setting up expectations from the left-most words, and having the phrases be filled out, or instantiated, that I wanted to capture in ALANA. My way of capturing the idea of expectations is to have PHRAN-like patterns. What I want in addition to just patterns is to have all different levels of patterns being matched at the same time so that both specific and general information can be ascertained from the input at the same time.

There is a class of sentences which would tend to defy this single word, left-to-right analysis style. These are the *Garden Path* sentences. A Garden Path sentence is a sentence in which after you have read most of the sentence, you reach a word or phrase at which point, you realize that you have to back up and re-analyze some of the sentence. An example is **The secretary reported seeing a bug, which caused a massive CIA investigation.** In this sentence, we are parsing along, and when we see the word **bug**, people usually first instantiate it to the small multi-legged animal that people find unpleasant having in office situations[GEH84]. Things seem fine until we see

the word CIA. Not being able to make sense of this, people sometimes have to stop here to figure out that **bug** probably referred to a small listening device used by an adversary to hear what is going on. This step required backing up and reassigning a meaning to **bug**, which then involves re-analyzing the sentence from the word **bug**. In fact, one may go back further to realize that **secretary** may not refer to the usual sense of secretary (i.e. a person employed as an aide to a superior to perform tasks such as typing, organizing, and generally being helpful), but may refer to a government secretary such as the Secretary of State or Secretary of Defense.

The type of backing up required by the understanding components in this example includes the parser, but is not completely done by the parser. The parser is only involved in taking the input text and matching it against patterns. It has to work in an integrated fashion with an understanding mechanism that performs concretion and other tasks. Rather than having the parser finish its task before the understander takes control, the parser and the understander should work together to form the total analyzer. In case the input is ill-formed, the analyzer can still provide a meaning of a sentence by reporting to the understander, phrases that match. The understander, then, can potentially help the parser by providing extra information that the parser can use to instantiate a pattern even if the components were not in the correct position within the sentence. This way of parsing is known as *Integrated Parsing*. Other integrated parsers are described in papers [RM85] [SLB80].

3.4 Pattern Matching

The main goals for the design of the pattern matcher were that it should be simple and general. The pattern matcher should not try to know anything about analysis, only about instantiating patterns and calling the attached *TEST*s and concepts (*builds* functions). The main purpose of the pattern matcher is to build the parsing chart used by it and other parsing routines. It should also instantiate, or at least test for instantiation any pattern all of whose components are found to be adjacent on the parsing chart. This section describes the pattern matcher and how all these goals are met.

3.4.1 Pattern Storage

As with PHRAN, the patterns are stored in a *Discrimination Net* [CRM80]. In ALANA, the patterns are stored in a special type of discrimination net more accurately referred to as a discrimination tree since there are not any

cycles and each node in the tree has exactly one parent. I will use the terms *discrimination net*, *discrimination tree* interchangeably. The discrimination net is actually used to store the concepts (*builds*) and *TEST*s with the pattern components acting as an index to them. As the matcher matches a pattern, it follows the path in the discrimination net. When it finds that a particular discrimination net node has information attached, it calls the routine **do-any-new-actions** which will either build a concept, call a *TEST* for feasibility, set a global variable, or all of the above. Figure 15 shows an example of how a simple pattern is stored in the database.

```
(defpat (Q => how Aux S)
   (builds concept))
```

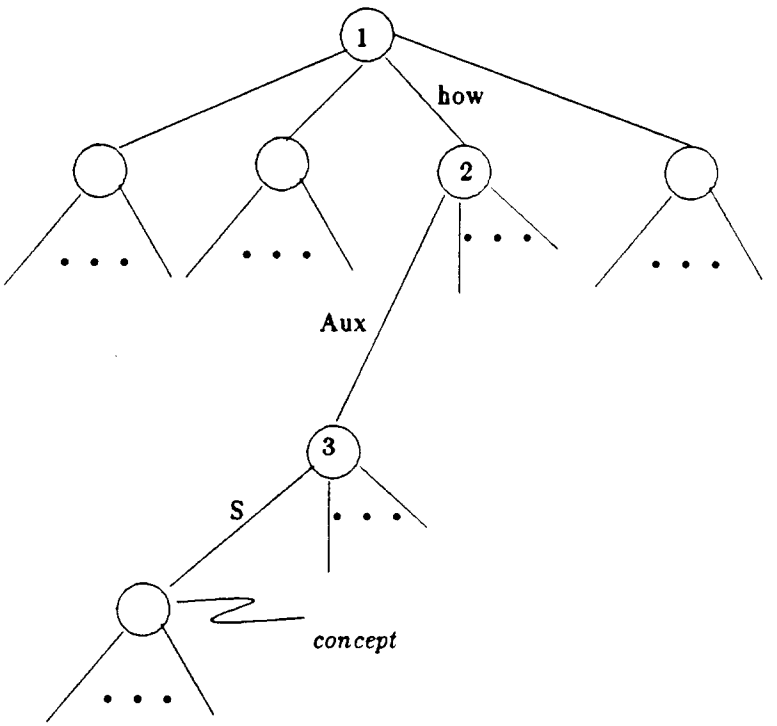


Figure 15: Discrimination Net of **how Aux S**

After seeing the component **how**, the matcher advances from node 1 to 2 and searches for components starting from 2. After seeing an **Aux**, the matcher

will advance to 3. When all the pattern components are seen and we make it to 4, the matcher will check if a *TEST* function exists. If one is attached to the discrimination net node, it will call the *TEST* function as described in § 2.3.2. If the *TEST* fails then the pattern describing this path is discarded (more about this later). If the *TEST* succeeds, then a new chart edge is added to the chart, and the *builds* function is called to assert new conceptual knowledge to the network. It would be at this point that a coroutine might take over and decide what to do with the new conceptual knowledge.

The success of a discrimination net highly depends on its shape and how many children nodes have. If a highly used node (such as the top node) has many children, then much time could be spent just searching through those children to see if the matcher can advance through one of them. In the current implementation, ALANA just does a linear search at each node. It could easily be speeded up here by using either a hashing or binary search mechanism. Up to now, however, it has not been worth the overhead. One optimization I have made, though, is that patterns with only one component (i.e. look like (**defpat** (*Name* => *Component*) ...)) not be indexed under the discrimination net, but rather be indexed under the name of the atom component itself (using its property list). This way we keep the top level of the discrimination net small, and get to use LISP's internal hashing mechanism to find these single-component patterns.

3.4.2 Open and Closed Patterns

To keep pattern matching as general as possible, we need to have a way to match the components no matter where they may begin or end in an input sentence. We conveniently have available to us an ability to point to any position of the input in the form of the parsing chart which includes the words themselves. Combined with the ability to point anywhere in the patterns discrimination net, we are able to have a special data structure called an *open-pattern* which ties the discrimination net to the parsing chart. Such a data structure keeps track of where a pattern match began in the input, where it ends, and the matchings of edges to pattern component labels.

Abstractly, the structure looks as diagrammed in Figure 16. Here an open-pattern recognized a C in the middle of the chart. Since the pattern discrimination net indicates that C is a valid way to begin a pattern (i.e. a (**defpat** (*name* => C ...) ...)), the pattern matching process is begun. An open-pattern gets created and starts at the chart vertex where C

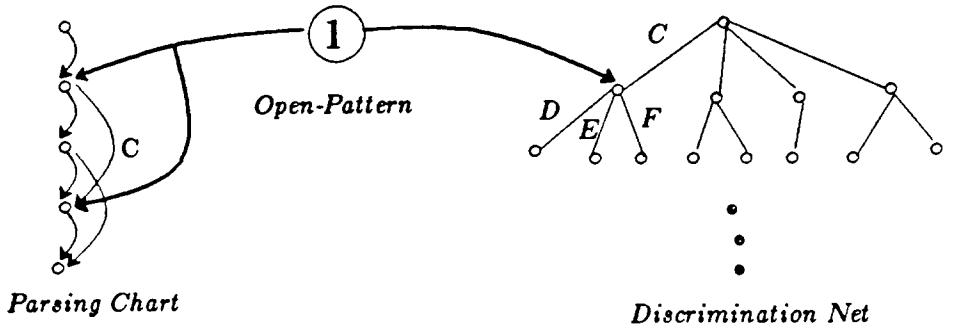


Figure 16: Abstract Diagram of how an open-pattern ties the chart to the discrimination net

was started, and ends where C ends. At the same time, it advances down the discrimination net along the C path.

The next components that can follow a C are D, E, and F. If one is seen, or gets added, following the C on the parsing chart, then a copy of the open-pattern is made and the copy will advance simultaneously along the chart and the discrimination net. The reason we make a copy is that there may be more than one valid component following the C (i.e. another D, E, or F), and we want to follow them all simultaneously. So, a possible next state of affairs may look like in Figure 17.

Open-pattern 2 now captures the complete C-E pattern. Open-pattern 1 is waiting for a*nother* occurrence of D, E, or F to show up. Although, not shown in the picture open-pattern 1 will not follow the same E since open-patterns actually point to edges, and once an *edge* within a vertex is considered, it is never considered again. If there were more than one possibility following the C-E, then open-pattern 2 will be copied (I refer to this as *cloning* an open-pattern) to consider all of them.

Before going to a more full example, recall that not every pattern is stored in the discrimination net. Those which had just one component are indexed under the atom itself. In this case, the pattern-name, *builds*, and *TEST* information is all stored in the node atom's property list. Because of the separation of modules between matching and database management, the matcher does

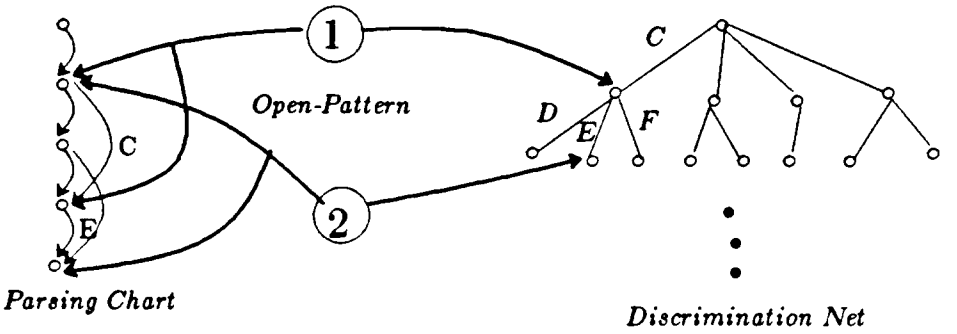


Figure 17: Two Open-Patterns

not need to know too much about how single-component patterns are stored except that they are not on the discrimination net. Thus, for every vertex, there is a single special open pattern known as a *single-component special* which slides one-way along the edges of that vertex and checks to see if that edge's label indexes a single component pattern. If it does, then the actions taken are just the same as if a multi-component pattern had completed (i.e. it calls TEST, **builds**, and adds a chart edge if appropriate).

Let us go partially through a detailed example to show how a parse using *just* the pattern matcher is actually performed. We will use the above example of parsing **the big red ball**. This example will also point out more accurately what the data structures look like to **ALANA**. We will proceed through a series of snapshots of the data structures.

In Figure 18, we see the set-up before any processing has begun. The chart starts off as a single chart vertex (labeled in a circle with 0) with no edges. The object pointed to by the vertex is a dummy empty edge which is pointed to by the open-patterns. When we want to add a new edge, we change the dummy empty edge so that the open-patterns will see changes made to the object that they point to. This is kind of like changing things under an open-pattern's feet. We will see more of this. The open-pattern 1 is initialized to begin and end at the vertex and also point to the top of the patterns discrimination net. Also initialized with the chart vertex is a *single-component special* open-pattern which is distinguished from a regular open pattern only by the fact

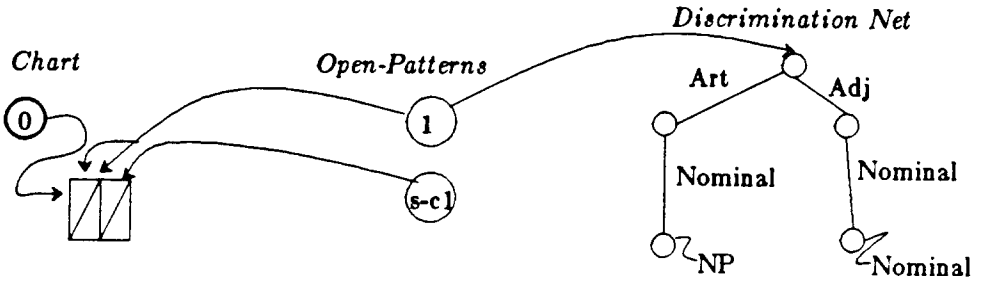


Figure 18: Snapshot 1

that it does not point into the patterns discrimination net. This open-pattern will be marching along the edges and checking to see if the edge labels index single-component defpat's. Every chart vertex is initialized with two open-patterns in the way shown above. Thus we are able to have patterns begin matching at any arbitrary point of the input.

After entering the word, **the**, as in Figure 19, we see that two more open-patterns got added and initialized in the same way as was done for vertex 0. We also note that open-patterns 1 and s-c1 now point to non empty edges. It is at this point, when open-patterns point to something non-nil, that instantiations occur as shown in the next snapshot (Figure 20)

At this point, open-pattern 1 checked **the**, and found that there was nothing at that level in the discrimination net that matched **the**. So, it slid along to the next edge, which was the empty edge for the 0 vertex. The s-c1, however, did recognize **the** as an index to a single-component defpat. At this point, the *TEST* and *builds* functions (if present) would be called to validate the instantiation and contribute to the KODIAK network respectively. Figure 20 shows that s-c1 added an edge to end at the same point that **the** ends, and then advanced itself. That is when snapshot 3 is being taken. Note that s-c1 indicated that **Art** should be added, an edge changed under open-pattern 1's feet, so that open-pattern1 will now consider the new edge.

In Figure 21, we see that open-pattern 1 recognized **Art** and so it made a copy of itself (open-pattern 3), which got slid to the next (empty) edge to wait in case another edge gets added that matches a component at the top level of the discrimination net. Then, one of the pointers of open-pattern 1 advanced

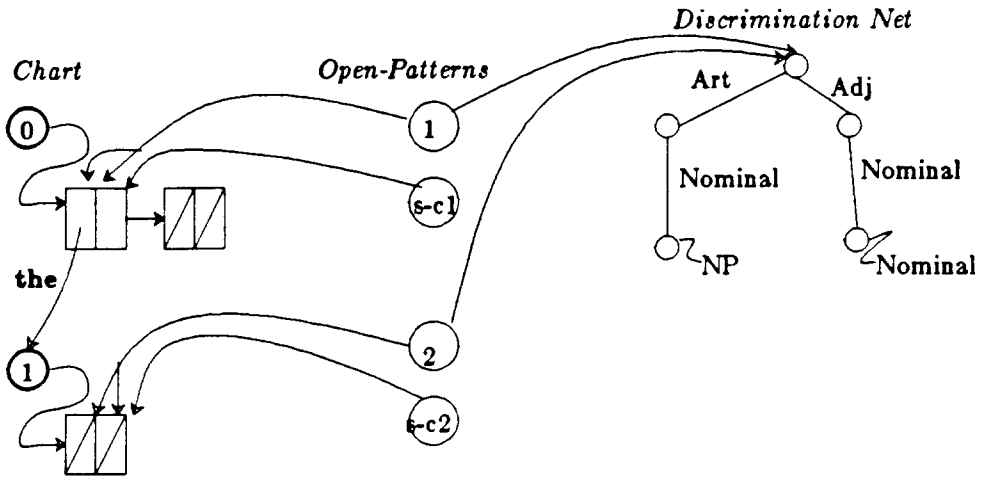


Figure 19: Snapshot 2

to point to the first edge of the vertex which matched the discrimination net link. Simultaneously, the pointer to the discrimination net advanced so that it now considers all the patterns that start with Art.

As we can see, continuing with such an example makes for messy diagrams very quickly. Figure 22 on page 63, however, is the ending diagram (with some editting) showing only the patterns that matched completely (it is shown only for completeness, it is not necessary to digest the whole thing right away to understand the main points). The diagram shows that the list of open patterns now corresponding to "closed" patterns (i.e. completely matched open-patterns). For this diagram, these closed-patterns are not numbered.

Once an open-pattern has traversed to the bottom node of the discrimination net path, it is removed from consideration by moving it to a *closed-patterns* list. Currently, once a pattern is in the closed list, it just stays there. While the closed pattern could be used as an inspector for those patterns that completely matched to get at the nodes and edges that matched, the current implementation does not take advantage of this feature.

Thus, the pattern matcher, which is what guides the analysis to be performed, does not read any new words until all the pattern matching that *can* be done *has* been done on all previous words. This is how we achieve our one-word-at-a-time, left-to-right reading. The patterns firing off simulates

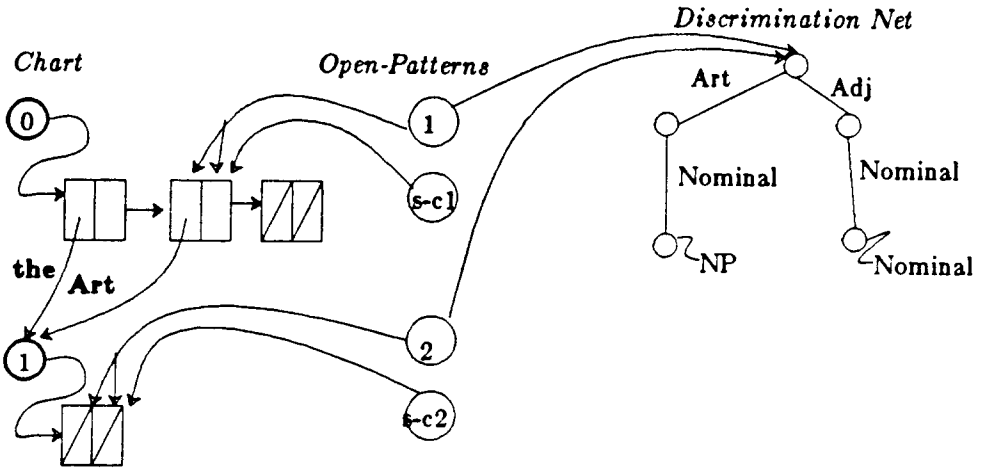


Figure 20: Snapshot 3

generating and/or fulfilling expectations. The way patterns get fired off is to simply go through the list of open-patterns as many times as it takes until none of them change.

3.4.3 Spelling Corrector

Any time a new word is entered, we expect at least one open-pattern to advance. If none can advance, we very likely have an unknown or misspelled word. By use of wildcards, such as `*anyword*` and `*unkword*`, we can expect unknown or arbitrary words at certain points and open-patterns containing such wildcards in their patterns would automatically advance. Thus if no open pattern can advance, we truly have an unknown word at a place we did not expect to find one. Such an unknown word is very likely to just be misspelled. If the correct spelling could be known, we could insert the correct word into the chart and try to match against it.

In UC, ALANA does make use of a spelling corrector [implemented by Jim Mayfield based on the algorithm used in Teitelman's DWIM[Tei78]]. When ALANA is stopped at a word where no open-pattern can advance, ALANA gathers all the words it expects and calls the spelling corrector with the unknown word and the list of expected words. Since ALANA technically can

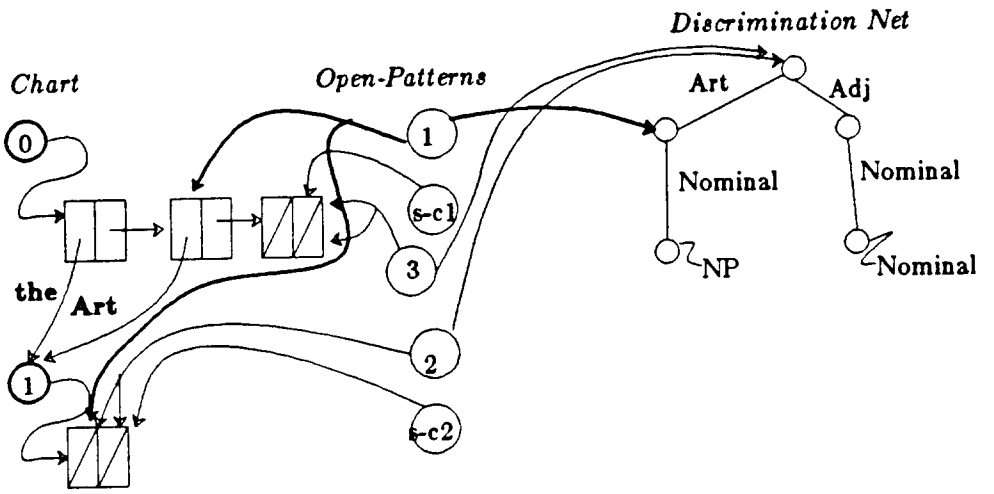


Figure 21: Snapshot 4

expect any word at any time, the list of expected words is made up only from the patterns that have already started matching.

The corrector then returns a list of valid candidates from the list of possibilities. To keep in the general spirit of parsing all that we can, all of the possibilities are entered into the chart at the point where the misspelled word is located. Then, if patterns are able to match more than one word, open-patterns will be cloned and those corresponding to nonsensical phrases will presumably be ruled out at a deeper level of analysis.

3.4.4 Timing Statistics and Further Optimizations

ALANA was developed entirely in Franz Lisp (Opus 38.91). The total number of lines (not including the pattern database), including comments is 1035 of which 175 is for pattern definition code, 569 is for the pattern matcher, 240 is for the database manager, and the rest is for miscellaneous interface functions. It is difficult to obtain timing statistics for ALANA since it depends on the speed of the representation functions.

Running only as a pattern matcher, in as fast a mode as possible for Franz Lisp (translink set to on), and using the current patterns list, sentences parse in around 115-380 ms. Running in UC, where ALANA needs to make calls to

KODIAK primitives, and execute *builds* functions, times are around 500-1400 milliseconds for a complete analysis depending on the sentence's complexity. So far, time has not been a critical problem. As previously mentioned, there are many areas where speed up could be achieved by using hashing at the discrimination net nodes.

I leave this area open for optimization in case the time-critical path of the analyzer is through inspecting the open patterns. Ideally, there should not be a problem here since the number of patterns should be small. However, without an understanding or concretion mechanism, the burden may fall on the analyzer to create the concepts needed by the rest of the system which may result in many patterns, hence we would have many many open-patterns.

I believe that there are lots of interesting ways this pattern-matching method could be optimized including checking for recency of use. We could move recently accessed open-patterns to the beginning of the open-patterns list in a way similar to Tarjan's linear splaying algorithm[Tar83]. In this way, we are likely to consider those open-patterns most likely to change with higher regularity. Before adding a new word, however, we would still want to sweep through the whole open-patterns list to make sure all matching had been done.

4 Example Trace

I now present a trace of ALANA running inside of UC. Currently there is a LISP top level which acts as a front end to UC. This top level reads the input into a list and passes the list of words (all in lower case) to the analyzer. Shown below is a briefly annotated trace of UC answering "How do I print a file?" with the analyzer verbose flag set so that we can watch its stepping through the parse.

```
Franz Lisp, Opus 42.15
(C) Copyright 1985, Franz Inc., Alameda Ca.
=> ?ld load
```

Loading in UC/KODIAK files

```
.
.
.
[fasl /na/bair/bair/UC/Source/Parser/aux.o]
[fasl /na/bair/bair/UC/Source/Parser/debug.h.o]
```

```
[fasl /na/bair/bair/UC/Source/Parser/UCparse.h.o]
[fasl /na/bair/bair/UC/Source/Parser/test.h.o]
[fasl /na/bair/bair/UC/Source/Parser/db.h.o]
[fasl /na/bair/bair/UC/Source/Parser/db.o]
[fasl /na/bair/bair/UC/Source/Parser/match.h.o]
[fasl /na/bair/bair/UC/Source/Parser/match.o]
[fasl /na/bair/bair/UC/Source/Parser/parse.h.o]
[fasl /na/bair/bair/UC/Source/Parser/parse.o]
[fasl /na/bair/bair/UC/Source/Parser/pat.h.o]
[fasl /na/bair/bair/UC/Source/Parser/pat.o]

Zeroing *patterns* database...
Warning:  adding to already established pattern ((w directory))
Warning:  adding to already established pattern ((w name))
Warning:  adding to already established pattern ((w mother))
Warning:  adding to already established pattern ((w in))
```

The above warnings just mean that there are multiple paths to the above words

```
(load.l)
=> (setq &verbose 2)
2
```

...so we can see the chart as it gets built

```
=> (parse '(how do i print a file))
```

The chart is printed (during the verbose mode of parsing) as an edge list with the vertices numbered from 0 on. Each edge is indicated by [label > vertex-number], where label is the edge's label, vertex-number is the vertex to which this edge is directed. The most recently added edge is denoted with + signs. The edges which caused this label to be added are indicated with @ signs.

```
Resetting matching chart...

Entering New Word: how

--------------
Current Chart:
```

```
0: +[how > 1]+

1:

--------------

Entering New Word: do

--------------
Current Chart:
0: [how > 1]

1: +[do > 2]+

2:

--------------
Calling function builds-493790904-187, which is compiled.
```

The builds *function is compiled. If it were interpreted, we would see its definition here which is a series of calls to the KODIAK interpreter. See § 2.3.1 for what such an expansion would look like.*

```
--------------
Current Chart:
0: [how > 1]

1: @[do > 2]@ +[Aux > 2]+

2:

--------------
```

At this point, we just entered the Aux *for the* how Aux S *pattern*

```
Entering New Word: i

--------------
```

```
Current Chart:
0: [how > 1]

1: [do > 2] [Aux > 2]

2: +[i > 3]+

3:

--------------
Calling function builds-493790904-185, which is compiled.

--------------
Current Chart:
0: [how > 1]

1: [do > 2] [Aux > 2]

2: @[i > 3]@ +[Person > 3:*USER*]+

3:

--------------
```

The chart printer prints concepts attached to edges as [*label* > *vertex-number* : CONCEPT]. *Hence, in the above chart, the* *USER* *indicates the concept associated with the chart edge for* Person.

.
.
.

(Skipping several steps)

```
--------------
Current Chart:
0: [how > 1]

1: [do > 2] [Aux > 2]
```

```
2: [i > 3] [Person > 3:*USER*] [Intrans-S > 4:PRINT-ACTION0] ...
   @[S > 4:PRINT-ACTION0]@ +[DeclS > 4:TELL0]+

3: [print > 4] [Print > 4:PRINT-ACTION0] [V > 4:PRINT-ACTION0] ...
   [Verbal > 4:PRINT-ACTION0] [Intrans-VP > 4:PRINT-ACTION0] ...
   [VP > 4:PRINT-ACTION0]

4:

--------------
```

At this point, the words how do i print *have been read. When* print *was read, a PRINT-ACTION instance (called PRINT-ACTION0) was asserted in the KODIAK network, and a pointer to that concept was initially entered on the chart as being associated with the* Print => print *pattern. This PRINT-ACTION0 can be seen to have propogated up as different patterns were fired that made use of the verb* print *resulting in being the concept of the sentence* I print.

The parsing chart now shows that we have a how *followed directly by an* Aux *followed directly by an* S. *Thus, the* how-question *pattern (below) will* build *an ASK instance (where ASK is an already defined KODIAK concept).*

```
(defpat how-question (Q => (w how) Aux S)
   (builds
      (old (concept-of S) is1 HYPOTHETICAL
         with cause (i ACTION is1 HYPOTHETICAL
                       witha actor _actor))
      (let ((Scause (get-value-of cause (concept-of S)))
            (Seffect (get-value-of effect (concept-of S))))
         (if Scause (old Scause is1 HYPOTHETICAL))
         (if Seffect (old Seffect is1 HYPOTHETICAL)))
```

The first part of this builds *states that the action of the* S *sentence is not an assertion of a fact, but rather an assertion of a desired effect. Hence, it is made a* hypothetical *instance to indicate that* S *is not to be taken as a literal assertion.*

The second part of the builds *creates a new instance of an ASK relation. The definition of an ASK is assumed to already be defined in the KODIAK network. We note that creating an ASK involves creating an instance of a QUESTION which makes access to part of the sentence assertion referred to by the* S *pattern component.*

```
        (i ASK
           witha asked-for (i QUESTION
                              witha what-is (get-value-of cause
                                                (concept-of S)))
           witha speaker _speaker
           witha listener _listener)))
```

Chart result of instantiating `Q => how Aux S`

```
Current Chart:
0: @[how > 1]@ +[Q > 4:ASK0]+

1: [do > 2] @[Aux > 2]@

2: [i > 3] [Person > 3:*USER*] [Intrans-S > 4:PRINT-ACTION0] ...
   @[S > 4:PRINT-ACTION0]@ [Decl S > 4:TELL0]

3: [print > 4] [Print > 4:PRINT-ACTION0] [V > 4:PRINT-ACTION0] ...
   [Verbal > 4:PRINT-ACTION0] [Intrans-VP > 4:PRINT-ACTION0] ...
   [VP > 4:PRINT-ACTION0]

4:
```

As can be seen in the above chart, ASK0 only represents the question `How do I print.` *Since ALANA is reading one word at a time, it doesn't look ahead, which is why it built the question. However, since there is further input, it will keep going and continue with the analysis.*

.
.
.

```
Current Chart:
0: [how > 1] [Q > 4:ASK0]
```

```
1: [do > 2] [Aux > 2]

2: [i > 3] [Person > 3:*USER*] [Intrans-S > 4:PRINT-ACTION0] ...
   [S > 4:PRINT-ACTION0] [DeclS > 4:TELL0] ...
   [Trans-S > 6:PRINT-ACTION0] @[S > 6:PRINT-ACTION0]@ ...
   +[DeclS > 6:TELL1]+

3: [print > 4] [Print > 4:PRINT-ACTION0] [V > 4:PRINT-ACTION0] ...
   [Verbal > 4:PRINT-ACTION0] [Intrans-VP > 4:PRINT-ACTION0] ...
   [VP > 4:PRINT-ACTION0] [Trans-VP > 6:PRINT-ACTION0] ...
   [VP > 6:PRINT-ACTION0]

4: [a > 5] [Unspec-Art > 5] [Art > 5] [File > 6:FILE0] ...
   [Unix-File > 6:FILE0] [Nominal > 6:FILE0] [NP > 6:FILE0]

5: [file > 6] [File > 6:FILE0] [Unix-File > 6:FILE0] [Nominal > 6:FILE0]

6:
--------------
```

At this point, we have read all the words, and have already analyzed a new sentence, `I delete a file`. *Since we have a new* `S` *(from vertices 2 to 6), we can once again use* `how Aux S` *to instantiate a new Question (or* `Q`*) pattern. This time, the ASK instance (*`ASK1`*) represents the whole question that we passed to the* `parse` *function.*

Since no more patterns will fire, ALANA's task is complete.

```
--------------
```

Final Chart

```
Current Chart:
0: @[how > 1]@ [Q > 4:ASK0] +[Q > 6:ASK1]+

1: [do > 2] @[Aux > 2]@

2: [i > 3] [Person > 3:*USER*] [Intrans-S > 4:PRINT-ACTION0] ...
   [S > 4:PRINT-ACTION0] [DeclS > 4:TELL0] ...
   [Trans-S > 6:PRINT-ACTION0] @[S > 6:PRINT-ACTION0]@ ...
```

```
   [DeclS > 6:TELL1]

3: [print > 4] [Print > 4:PRINT-ACTION0] [V > 4:PRINT-ACTION0] ...
   [Verbal > 4:PRINT-ACTION0] [Intrans-VP > 4:PRINT-ACTION0] ...
   [VP > 4:PRINT-ACTION0] [Trans-VP > 6:PRINT-ACTION0] ...
   [VP > 6:PRINT-ACTION0]

4: [a > 5] [Unspec-Art > 5] [Art > 5] [File > 6:FILE0] ...
   [Unix-File > 6:FILE0] [Nominal > 6:FILE0] [NP > 6:FILE0]

5: [file > 6] [File > 6:FILE0] [Unix-File > 6:FILE0] [Nominal > 6:FILE0]

6:

--------------
ASK1
=>  (show-all ASK1)
(ASK1 (listener6 = UC)
      (speaker6 = *USER*)
      (asked-for3 = (QUESTION3 (what-is3 =
                                  (ACTION3 (actor5 =
                                              *USER*))))))
=>
```

ASK1 is a pointer to the particular instance created by this analysis. The KODIAK function `show-all` *prints out the relation and all its aspectuals with their values.*

5 Conclusion

I have presented in this report not only a paradigm for natural language analysis based on the successes of past parsing and analysis efforts, including integrated parsing and using phrasal lexicon, but also a general tool that can be used for this paradigm. **ALANA** is representation independent, has a declarative style knowledge base, and because of the simplicity of its powerful ideas, is easy to extend and modify. It is hoped that rather than **ALANA** following the tradition of being scrapped immediately after the author's departure, that its core of ideas, and maybe even code, will branch towards different needs

for front-end or integrated parsing, or progress as a way of studying linguistic analysis.

I close this report by giving some of the strengths and weaknesses of my system, and finally by indicating my idea of what further directions this project was intended to head.

5.1 ALANA's Strengths

The advantages of ALANA over PHRAN stem mostly from ALANA's simplicity. This simplicity leads to a high degree of flexibility. For example, with PHRAN it was not always known that adding a pattern to the linguistic knowledge base make make PHRAN succeed in actually using that pattern to analyze (c.f., Problems with PHRAN, § 2.1.4). The reasons for PHRAN's problems were that PHRAN's mechanism tried to know a lot about what was going on in terms of analysis, and as a result, made many assumptions that reduced its flexibility. Not so with ALANA since every pattern that it knows about is guaranteed to match if the constituents are present. Thus, anyone adding knowledge to ALANA need only be concerned with the integration of new patterns with the database. The knowledge adder does not also need to fight the matching algorithm.

Even though there is a rigid concept defining structure to the pattern concept pairs, one can have arbitrary pieces of LISP there to allow for complete flexibility. Using LISP directly is undesirable, but since it is available, I did not have to write an interpreter for things that were already available in LISP such as having the ability to define local or global variables. Since the processing of concept functions is centralized, one can very easily change the representation of concepts without changing the analyzer. It is my hope that a system, such as proposed by Wu (discussed in § 2.4) will fit into my analyzing tool as an extension.

5.2 ALANA's Weaknesses

I think that the best way to sum up ALANA is by saying that it is a "weak", but general method of parsing, just as depth-first search, best-first search, hill-climbing, etc. are "weak" methods of problem solving. By itself, ALANA does not capture much of what I consider to be AI.

Much of the "AI" in natural language processing, I have pushed to more specialized, but as of yet, largely unwritten, routines that know about specific pieces of knowledge and can make inferences and concretions if handed the

right things from a parser. However, at the linguistic analysis phase, there are lots of things ALANA does not do.

5.2.1 Ill-formed Input

If a user of the system makes grammatical errors, crucial patterns may fail to become instantiated. A tricky problem comes if the misspelled word is a recognized word. This comes up if the user typed the sentence `How do I delete flies?` where `flies` should be `files`. Using the ALANA paradigm, the analyzer would go ahead and parse `flies`. However, the ill-formed semantics of this sentence could possibly be caught at a *TEST* of a high level pattern (if it were to check for semantics). Some other understanding mechanism, however, could indicate to the analyzer, in the form of a signal, that a certain chart edge is not instantiated correctly, and needs to be re-evaluated. This signal, or message could indicate that a word does not make sense, and could possibly be misspelled. The spelling corrector may then re-evaluate the word of that edge, and analysis would be restarted from the reading of that word.

In general, the solutions ALANA would have for ill-formed input is that the analyzer does the best it can by matching the patterns it can and building the associated concepts. If concepts are not combined because a particular pattern could not instantiate, there may be enough redundancy in the semantic network from patterns already matched and the concepts already built that the understander can still figure out what the user meant.

5.3 What Needs To Be Done

The analyzer needs some sort of morphological analyzer. The UNIX spell program has a number of morphological rules which can be used to determine the roots and tenses of words. It is not clear if these rules are in the public domain and whether they could be published, but a weaker set of rules (actually a flowchart) can be found in Winograd's book [Win72], page 72. Such a routine can be used to strip off prefixes and suffixes and construct the root of a word.

The main part of ALANA that is unspecified in current implementations, is the real AI of the system, the understanding/concretion mechanism. This is the part that may, for example, take a subject and verb predicate and find out if the pair can be made more specific to a defined concept in the semantic network. For example, in UC, if it is known that the user wants to delete a file, a DELETE-FILE-ACTION instance must be created for the rest of the

system to be able to handle inferences on deleting files. However, the most the analyzer should be expected to do is to create a DELETE-ACTION with the delete object being a file. In other words, seeing the word delete with an object should create the DELETE-ACTION of that object. The analyzer should not have to know that a DELETE-FILE-ACTION is what needs to be created just because other parts of the system depend on it. Rather, the concretion mechanism should concrete DELETE-ACTION of a FILE to a DELETE-FILE-ACTION. At this point, however, such a system does not exist and the analyzer has to be kludged to create the specific DELETE-FILE-ACTION by having a special Delete-File Verb Phrase pattern. This solution is unsatisfactory. However, work by Dekai Wu is being done to add a mechanism that would run in conjunction with the analyzer so that the phrasal patterns can be made less specific. I am hopeful that this approach to analysis will allow for not only a general model of understanding, but also a "knowledge-based" AI system into which new knowledge can be easily added.

Another part of ALANA's design that needs to be more incorporated into its implementation is its coroutine structure with other programs being used to form inferences at "read" time. As an example, to show ALANA's cooperative/coroutine nature and how other routines can affect the way ALANA's matcher analyzes a sentence, there may be a pattern <PRINT> <N>, which would match the verb phrase of a sentence such as "How do I *print* a *file*?". At the same time, one would want to interpret "How do I get a listing of a file?" as having the same meaning as the previous sentence. One solution for recognizing that *print* is equivalent to *get a listing of* is to have two patterns sharing a common concept so that PRINT is asserted whenever either the word sequences **print** or **get a listing of** are seen. This is in fact how ALANA currently recognizes both of the above sentences.

The more ideal possibility for recognizing the conceptual equivalences of the above phrases is not to just store patterns for each phrase, but to have the analyzer initially and naively analyze *get a listing of* as a receiving of some listing object. Some concretion or understanding mechanism would then recognize that *get a listing of* is a PRINT, and simply assert that PRINT so that the matcher could substitute it for the words *get a listing of* and be able to fire off <PRINT NP> patterns. In this way, understanding and inference mechanisms can take place during analysis time, which I feel is closer to how humans understand (c.f., § 3.3).

6 Appendices

6.1 How to Use

As of January 17, 1986, the source to ALANA can be found on the UC Berkeley **kim** VAX 11/780 in the **bair/UC/Parser/** directory. To run, one simply starts up a Franz Lisp session and, at the prompt, execute the function *(load 'load)*. All of ALANA's modules are then loaded. At this point, the pattern matcher alone can be exercised by calling the function *parse* with a list of words as the argument. If the KODIAK interpreter and a knowledge base is also loaded in the lisp environment, one can set the *&use-kodiak* flag to actually make calls to build KODIAK concepts.

During processing, the variable *&verbose* can be set to different numeric values where the higher values indicate more verbosity. To see the final chart produced, one can call the function *(&print-chart)* which will print the chart as formatted in the example in § 4.

7 Acknowledgements

Each member of the BAIR project has inspired me in some way that helped to shape this project. The current and past members of this group include Anthony Albert, Rick Alterman, Yigal Arens, Mike Braverman, Margaret Butler, Dave Chin, Joe Faletti, Paul Jacobs, Marc Luria, Jim Martin, Jim Mayfield, Peter Norvig, Lisa Rau, Nigel Ward, and Dekai Wu. I also would like to acknowledge inspiration I have received from students and faculty in Psychology and Linguistics, especially to Prof. George Lakoff for giving me insights into the nature of understanding.

Thanks also to Nina Herrera and Clarice Cox for helping with some of the proofreading.

Finally, thanks to Prof. Robert Wilensky for supporting me during graduate school and reviving my inspiration for this project when my spirits were low.

This work was supported by Office of Naval Research (ONR) contract number N0014-80-C-0732, by Defense Advanced Research Project Agency (DARPA/DoD) contract number N00039-82-C0235.

References

[Are81] Y. Arens. How to Write PHRAN Patterns. 1981. This report is to appear as a chapter in Arens' Ph.D. Thesis.

[BW80] R.J. Bobrow and B.L Webber. Psi-Klone: parsing and semantic interpretation in the BBN natural language understanding system. In *Proceedings of the Third Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, Stanford University, Stanford, 1980.

[CRM80] E. Charniak, C. Riesbeck, and D. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.

[DH82] R. Douglass and S. Hegner. An Expert Consultant for the Unix System: Bridging the Gap Between the User and Command Language Semantics. In *Proceedings of the Fourth National Conference of Canadian Society for Computational Studies of Intelligence*, University of Saskatchewan, Saskatoon, Canada, 1982.

[GEH84] R. Granger, K.P. Eiselt, and J.K. Holbrook. The Parallel Organization of Lexical, Syntactic, and Pragmatic Inference Processes. In *Proceedings of the First Annual Workshop on Theoretical Issues in Conceptual Information Processing*, Atlanta, GA, 1984.

[Ger77] A.N. Gershman. Conceptual analysis of noun groups in English. In *Proceedings of the fifth International Joint Conference on Artificial Intelligence*, pages 132–138, Cambridge, Massachusetts, 1977.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison Wesley, 1978.

[Jac83] P. Jacobs. Generation in a Natural Language Interface. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, Germany, 1983.

[Joh83] R. Johnson. *Parsing with Transition Networks*, chapter 4, pages 59–72. Academic Press, 1983.

[Lak69] G. Lakoff. On Generative Semantics. In *Papers from the Fifth Regional Meeting of the Chicago Linguistic Society*, Linguistics Department, University of Chicago, 1969.

[LB81] M. Selfridge L. Birnbaum. *Conceptual Analysis of Natural Language*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981.

[Rie78] C. Riesbeck. *An expectation-driven production system for natural language understanding.* Academic Press, New York, 1978.

[RM85] C. Riesbeck and C. Martin. *Direct Memory Access Parsing.* Technical Report, Yale University, Report No. YALEU/DCS/RR 354, Yale University, 1985.

[SA77] Roger Schank and Robert Abelson. *Scripts Plans Goals and Understanding.* Lawrence Erlbaum Associates, 1977.

[SLB80] R. Schank, M. Leobowitz, and L. Birnbaum. An Integrated Understander. *American Journal of Computational Linguistics*, 6(1):13ff., 1980.

[SR82] S. Small and C. Rieger. *Parsing and Comprehending with Word Experts (A Theory and its Realization).* Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1982.

[Tar83] R. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[Tei78] W. Teitelman. *The Interlisp Reference Manual.* Technical Report, Xerox PARC, October 1978.

[WA80] R. Wilensky and Y. Arens. *PHRAN—A Knowledge Based Approach to Natural Language Analysis.* Technical Report, University of California at Berkeley, 1980.

[WAC84] R. Wilensky, Y. Arens, and D. N. Chin. Talking to UNIX in English: An Overview of UC. *Communications of the ACM*, 27(6), 1984.

[Wal78] D. L. Waltz. An English language question-answering system for a large relational database. *Communications of the ACM*, 21:526–539, 1978.

[Wil83] R. Wilensky. *Planning and Understanding.* Addison-Wesley, Reading, MA, 1983.

[Wil84] R. Wilensky. KODIAK, A Representation Language. In *Proceedings of the First Annual Workshop on Theoretical Issues in Conceptual Information Processing*, Atlanta, GA, 1984.

[Win72] Terry Winograd. *Understanding Natural Language.* Academic Press, 1972.

[WKN72] W.A. Woods, R.M. Kaplan, and B. Nash-Webber. *The lunar sciences natural language information system: Final report.* Technical Report, Bolt Beranek and Newman, Report No. 2388, Cambridge, Massachusetts, 1972.

[WM81] R. Wilensky and M. Morgan. *One Analyzer for Three Languages.* Technical Report, Electronics Research Laboratory, Report No. UCB/ERL M81/67, College of Engineering, UC Berkeley, 1981.

[Woo70] W. A. Woods. Transition network grammars for natural language analysis. *Communications of the ACM*, 13:591–606, 1970.

[Woo80] W.A. Woods. Cascaded ATN grammars. *American Journal of Computational Linguistics*, 6(1):1–12, 1980.

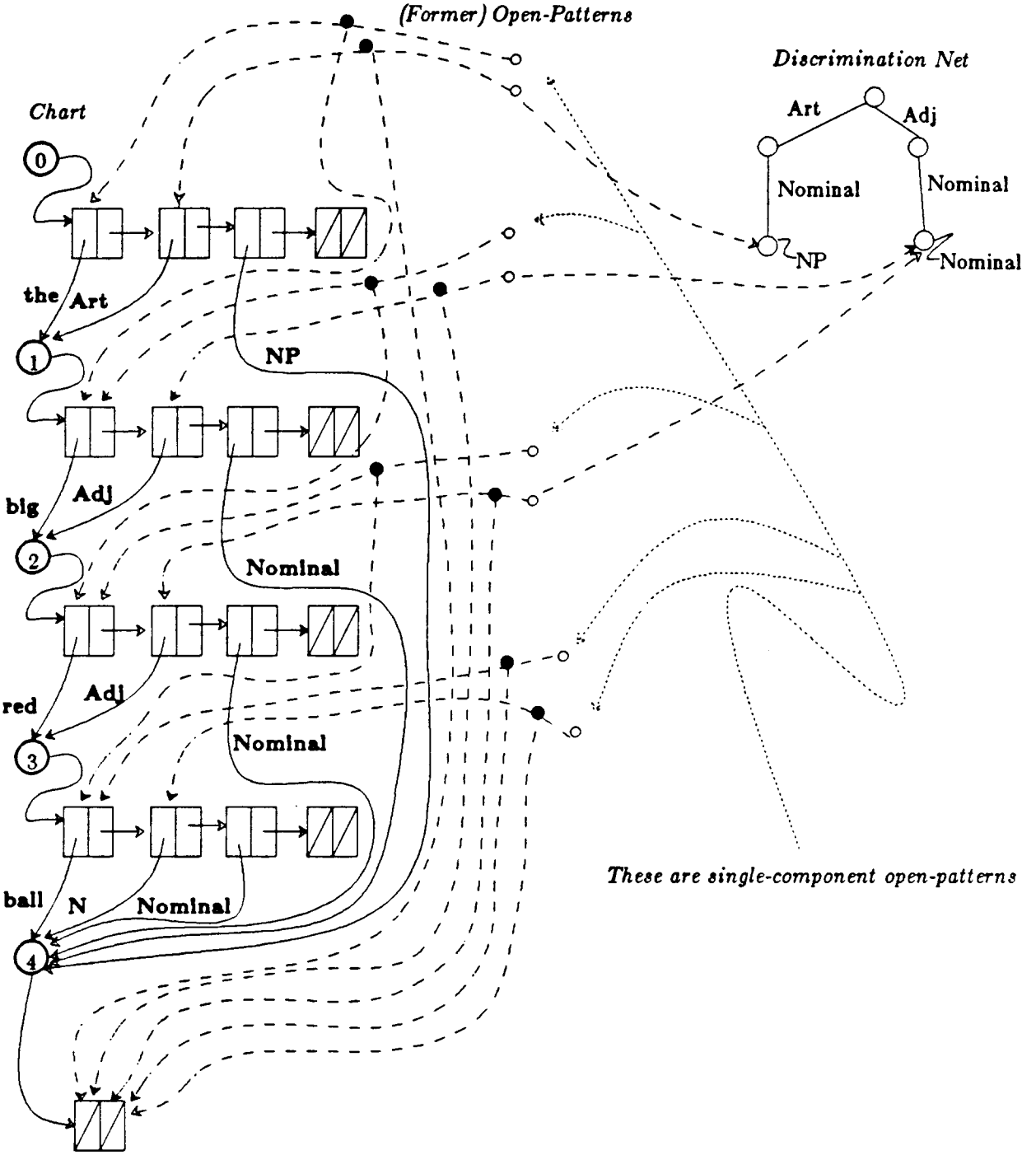


Figure 22: Snapshot 5