MULTIPAR: a Robust Entity-Oriented Parser[1]

Jill Fain, Jaime G. Carbonell, Philip J. Hayes and Steven N. Minton
Computer Science Department, Carnegie-Mellon University,
Pittsburgh, PA 15213, USA

1. Objectives

The phenomenon of human language comprehension has posed a considerable number of challenging problems for linguists, psychologists, philosophers, and artificial intelligence researchers alike. One particularly important aspect is the robust manner in which people are able to comprehend novel utterances, many of which deviate from the abstract notion of grammatical correctness. Whereas most linguists and philosophers focus on formulating *competence* theories of the idealized speaker, our objective is to model human-like *performance* in robust comprehension of naturally-occurring utterances. Within that general objective, this paper presents a concrete computational model of language interpretation capable of tolerating grammatical deviations, approximately to the extent exhibited in human language comprehension. Unlike the psychological approach that strives to model the fine-structure internal process of a cognitive task, we seek only to match observed human performance. Eliminating the constraint of strict adherence to human processing — to the extent that it can be measured or theorized — enables us to make much more rapid progress in developing an effective computational model.

Thus, our objectives can be summarized as follows:

- Understand the information processing requirements of robust natural language comprehension, including the integration of syntactic, semantic and pragmatic knowledge.
- Build an effective computational model for experimentation, refinement, validation, or refutation of our theoretical precepts. This requires the formulation of flexible control strategies and fairly rich knowledge representation formalisms.
- Apply the robust language interpreter to human-computer natural language interfaces, and investigate the extent to which such powerful, human-like abilities enhance the communication process.

In order to realize these objectives, we developed an experimental parsing system called MULTIPAR [4] based on earlier work at Carnegie-Mellon University, such as the CASPAR and DYPAR parsers [3, 9, 4, 5]. Unlike previous work on ATN-based robust parsing [14, 13, 11, 10], our approach is based on more flexible and interpretive case-frame instantiation methods. The case-frame approach enables us to integrate diverse syntactic and semantic knowledge into a universal data structure, which can be accessed by different parsing strategies. In essence, the computational model elaborated below consists of finding the best possible parse satisfying all the syntactic, semantic and pragmatic constraints and expectations. If one fails to find such a parse — perhaps due to missing function words, misspellings, other grammatical errors, semantic constraint violations, etc. — the constraints are gradually relaxed *in a principled manner* to find the best possible parse(s). A second theme interwoven with the progressive relaxation process, is the notion of invoking multiple parsing strategies, depending upon expectations of grammatical structure, semantic role, and suspected deviations from the correct grammar. The bulk of this paper presents the methods and knowledge sources that make robust parsing an effective computational process. The principles underlying our approach are:

- ***Grammatical tolerance levels***: The search space of possible interpretations of an utterance can grow prohibitively large if one allows compounding of all possible grammatical and semantic deviations. Therefore, the search is carried out using a least-deviant-interpretation-first strategy. This requires the postulation of additive *tolerance levels*. Each grammatical deviation is assigned a tolerance level, and multiple deviations in the same utterance correspond to the sum of the tolerance levels.
- ***Multiple strategies***: Each known deviation from grammatical well-formedness indexes one or more recovery strategies, as well as the required tolerance level increment. An agenda-based control structure allows strategies to post hypotheses of suspected deviations (and their associated recovery strategies), which are examined subsequently

[1]This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551, and in part by the Air Force Office of Scientific Research under Contract F49620-79-C-0143. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the Air Force Office of Scientific Research or the US government.

only in the case that no fully grammatical, or less deviant interpretation is found.

- ***Entity-oriented parsing***: An entity is a generalized case-frame. In addition to storing the semantic roles, headers, and case markers, it may represent declaratively multiple syntactic mappings from the surface input to the semantic representation.

The power of the MULTIPAR system lies in the integration of these principles into a cohesive computational model. The central objective of the system is to comprehend less-than-perfect utterances much as humans would, rather than to hallucinate all possible nonsensical interpretations. The first stage is to develop the general computational model, and the second is to tune it to find only those interpretations corresponding to human-tolerable grammatical deviations. After outlining the continuation-passing control structure and the entity-based knowledge representation, we present in some detail one example of parsing a plausible but imperfect utterance.

The MULTIPAR approach differs from earlier inference-intensive systems that sought to complete a parse in the presence of an unknown word [7, 2] in that it provides a uniform computational framework to recover from all grammatical difficulties and deficiencies. Moreover, it differs from text-scanning approaches in that it does not resort to ignoring large segments of input it judges either incomprehensible or uninteresting [12, 6].

2. MULTIPAR Control Structure

Like most natural language parsers, MULTIPAR operates by searching through a virtual space of possible parses for a given input. The size of the search space depends on the number of local ambiguities that are encountered during the parsing process. Because MULTIPAR is expected to parse ungrammatical input, it is typically confronted with a search space that is much larger than that explored by conventional parsers. Unlike its more conventional predecessors, MULTIPAR cannot simply reject a partial parse when a grammatical constraint is violated. Instead, various recovery techniques are applied.

If the best parse is to be found in a timely manner, the exploration of alternative paths in the search space must be carefully controlled. For example, spelling correction should be tried before hypothesizing a missing word, and hypothesizing one word is preferable to hypothesizing five. Moreover, if a strategy fails to find a correct parse for some input, it does not mean that the recovery actions should be invoked immediately. It may be that some other strategy will find a grammatical and semantically consistent parse. If so, that other strategy should be given its chance before any recovery actions are attempted.

To control the exploration of the search space, competing alternatives within a strategy are explicitly specified using SPLIT statements. When a SPLIT statement is encountered, the computation divides into separate branches; each branch has a *flexibility increment* indicating the additional degree of ungrammaticality tolerated by the associated continuation. A partial parse is generated along each branch, and each computation may proceed from the SPLIT statement independently of the others.[2]

```
(Split (+0 continuationA)
       (+1 continuationB)
       (+3 continuationC)...)
```

Thus, a SPLIT statement such as the one above produces a three-way branch in the search tree. Continuation A does not result in a gain in flexibility of the associated parse (implying that this continuation requires no additional relaxation from grammatical correctness). Continuations B and C, if they are ever pursued, will result in gains of one and three respectively (implying that different degrees of additional relaxation from grammatical correctness are required to continue each parse).[3] All continuations in a SPLIT statement are posted in a global agenda, indexed by their global *flexibility level* (explained below). The structure of a global agenda looks like:

```
[Agenda: (0  Continuation-A  Continuation-B ...)
         (1  Continuation-C ...)
         (4  Continuation-D ...) ...]
```

---

[2] MULTIPAR is implemented in Common Lisp.

[3] The non-determinism provided by the SPLIT statement is also exploited for handling normal ambiguities. When a strategy finds that a parse is ambiguous, a separate branch is created for each alternative parse. +0 flexibility increments are used for each locally-ambiguous continuation.

MULTIPAR starts with an agenda containing one or more top-level strategies at the 0-flexibility level, and a global flexibility level set to 0 (i.e. full grammaticality). As each strategy is applied it can post additional continuations in the agenda at the 0 or higher flexibility levels. Continuations are posted when SPLIT statements are encountered, and the flexibility levels at which they are posted correspond to the sum of the global flexibility level and the flexibility increment of the continuation. When a continuation completes execution it is removed from the agenda.

MULTIPAR pursues all the continuations at the 0-flexibility level. Ideally, exactly one of these yields a complete and consistent parse, indicating that the input was grammatical and unambiguous. If more that one parse is generated, we have true ambiguity, which must be resolved by external means. If no parses are generated at the 0-flexibility level, the input is ungrammatical (or semantically inconsistent), and MULTIPAR goes on to the next-lowest flexibility level in the agenda, selecting this one as the new global flexibility level. The process is repeated at this flexibility level (perhaps resulting in more continuations being posted on the agenda), and if, again, no parses are found, the global flexibility level is once more augmented. In this fashion, the search for a possible parse continues, seeking the least deviant interpretations first. If the global flexibility level reaches some preset threshold, MULTIPAR judges the input to be incomprehensible and abandons its parsing attempts.

3. MULTIPAR Entities

MULTIPAR is an entity-oriented parser. As described in [8], entity-oriented parsing is an approach to restricted domain natural language processing in which the parser is driven by a set of definitions of domain entities (objects, operations, and states). The entities are defined at a high level of abstraction, primarily in terms of other component entities. This approach to parsing is well adapted to dealing with ungrammatical input; it allows a parser to interpret the abstract entity definitions in a variety of ways so that it can look for the entity components in places other than the syntactically correct ones.

A simplified example of an entity definition used by MULTIPAR is:

```
(EntityName     MoveCommand
 SemanticCases  (
        Object      (FileObjDesc or DirectoryObjDesc)
        Source      (DirectoryObjDesc or LogicalDeviceObjDesc)
        Destination (FileObjDesc or DirectoryObjDesc)
        Location    (DirectoryObjDesc or LogicalDeviceObjDesc))
 Constraints    (
        (Destination FileObjDesc > Object FileObjDesc)
        (Object DirectoryObjDesc > Source LogicalDeviceObjDesc) ...
        (Required Object Destination))
 SurfaceForms   (
        (SFName Icf-Canonical
         Head           <movehead>
         DirectObject   Object
         Cases          (
                (Preposition <sourcepreps>
                 Bind Source) ...)))
 InstanceTemplate (
        Action  'MOVE
        Deviations Deviations
        Source (
                IsA (IsA in Object)
                Name (Name in Object)
                Extension (Extension in Object)
                Directory (Directory in Object or ... or Directory in Location)
                LogicalDev (LogicalDev in Object or ... or LogicalDev in Location)
                ObjDesc  (Description in Object)
                SourceDesc (Description in Source)
                LocDesc (Description in Location))
        Destination (
                IsA (IsA in Destination)
                Name (Name in Object or Name in Destination)
                Extension (Extension in Object or Extension in Destination)
                Directory (Directory in Destination or Directory in Location)
                LogicalDev (LogicalDev in Destination or LogicalDev in Location)
                DestDesc (Description in Destination)
```

```
LocDesc (Description in Location))))
```

This defines the "move" command for an operating system interface. Like all entity definitions, it has four main parts:

- **SemanticCases**: this defines the basic structure of the entity in terms of the other entities that are its components. MoveCommand, for instance, has an object (the file[4] or directory to be moved), a source (the directory or logical device to move it from), a destination (the file or directory to move it to), and a location (the directory or logical device in the context of which the move takes place). These semantic cases (collectively constituting a case-frame) do not tell MULTIPAR how to recognize the entity in an input utterance, they just define what other entities may or must be found in the input as part of finding the entity defined. The general format is a list of case names and filler types.

- **Constraints**: For any specific instance of an entity, the constraints specify both relations that are required to hold between cases, and predicates on individual cases. The first constraint for MoveCommand says that if the Destination case is filled by a file, then the Object case must also be filled by a file. Many constraints of this kind may be needed to fully specify an entity. The final constraint says that the Object and Destination cases are required to be present for any instance of MoveCommand. MULTIPAR enforces constraints of both types insofar as it can. In other words, it prefers parses in which the constraints are satisfied, but will accept (as a deviation) inputs in which the constraints are violated.

- **SurfaceForms**: This component of an entity definition tells MULTIPAR how an entity can be described in English. That is, it tells where in the input to find words identifying the entity itself (e.g. "move" or "transfer" in our example), as well as where to find fillers for the SemanticCases. The information about where to look for the SemanticCases is implicit in the specific parsing strategies associated with the SurfaceForm name (Icf-Canonical, or "imperative case frame canonical," above). Although there is a 1-to-1 mapping between SurfaceForm and strategy in the current implementation, it is not necessary that this be the case. A strategy can know how to parse more than one surface form and a SurfaceForm can be used by more than one strategy.

  The SFName is the only attribute common to all SurfaceForms. The other attributes are specific to particular surface forms. In the above example, the Head attribute defines the imperative verb to be used to identify the MoveCommand, the DirectObject attribute indicates which SemanticCase is the syntactic direct object of the imperative verb, and the Cases attribute defines which prepositions are used to mark the other SemanticCases in English input. Symbols like <movehead> are non-terminals in a grammar used by DYPAR [3, 1] (our pattern-matcher) and expand in the course of computation (e.g. <movehead> -> move | transfer). SurfaceForm slot fillers that are not surrounded by <>'s are SemanticCase names and tell the strategy which SemanticCase to bind the information to. Consider the DirectObject case of the Icf-Canonical SurfaceForm in the MoveCommand. The strategy that knows how to parse this kind of surface form will be passed an instance of the MoveCommand entity definition and an input segment to work on. When it finds a noun phrase unmarked by prepositions, it will check the surface form to find out what SemanticCase it should be trying to fill. Since the Object SemanticCase can be filled by either a FileObjDesc or DirectoryObjDesc, it will try to parse the unmarked segment as each of these. This will result in calls to strategies that know how to handle the SurfaceForms in the entity definitions of FileObjDesc and DirectoryObjDesc respectively. In general, an entity may have more than one surface form corresponding to different forms of surface expression for the same underlying semantic cases. The entity definition for FileObjDesc, for example, has two SurfaceForms: Ncf-Canonical[5] and Ncf-System. The former is designed for recognition of input like "the files with extension lsp in directory [c410jf90]," while the latter is used for recognition of descriptions that include proper names, such as "foo.lsp in [c410jf90]".

- **InstanceTemplate**: This information is used when a strategy has finished parsing an entity. It tells MULTIPAR the final representation to use for the entity instance thus produced. It is essentially a method for reformatting, "canonicalizing," and pulling information out of subordinate entity instances to compose the current one. The slot fillers in the InstanceTemplate act as directions to a routine that uses the bindings to the SemanticCases produced by the strategy. A single word means the value of the slot is exactly what is bound to the SemanticCase. A list without "or" means the value of the slot is whatever is found in the slot with that name in the InstanceTemplate bound to the SemanticCase. (e.g. IsA in Object says look at the IsA field in whatever kind of entity is bound to Object — if the field is not found, the slot = nil). Finally, directions that have one or more "ors" act as deterministic disjuncts. Each

---

[4] FileObjDesc means file object description; other abbreviations are similar

[5] Ncf is an abbreviation for NominalCaseFrame

instruction is followed until a non-nil value is found. In this way, "move [c410jf90]foo to my directory" produces the same InstanceTemplate as "move foo from [c410jf90] to my directory." In the former, the directory name for the source of the move is found in the FileObjDesc InstanceTemplate bound to the Object SemanticCase. In the latter, the directory name for the source of the move is found in the DirectoryObjDesc InstanceTemplate bound to the Source SemanticCase. Finally, each InstanceTemplate has a special slot called "deviations" which has no corresponding SemanticCase. This slot acts as a repository of information about the recovery actions taken by strategies.

4. An Annotated Example

To make clear how all the components of MULTIPAR combine and interact to parse both well-formed and ungrammatical input, we present the following extended example. Readers are advised to refer frequently to the figures in this section, and to the entity definition shown in section 3. In the discussion that follows, function calls and strategy names are written in boldface type; entity slots and values are given in italics. The notation "$word_1 ... word_n$" refers to the portion of the input utterance starting with $word_1$ and ending with $word_n$.

Consider the behaviour of the system when the user types:

Move the accounts directory the file Data3.[6]

Step 1. Initialization.

Before any part of the input is examined, the control tree is initialized as in Figure 1, label (a)[7]. Here, MULTIPAR sets up a branch for each of the commands in its current vocabulary. Note that no flexibility increment is added to the initial value of zero because no deviation has occurred. The control chooses the first level 0 branch on the agenda for continuation.

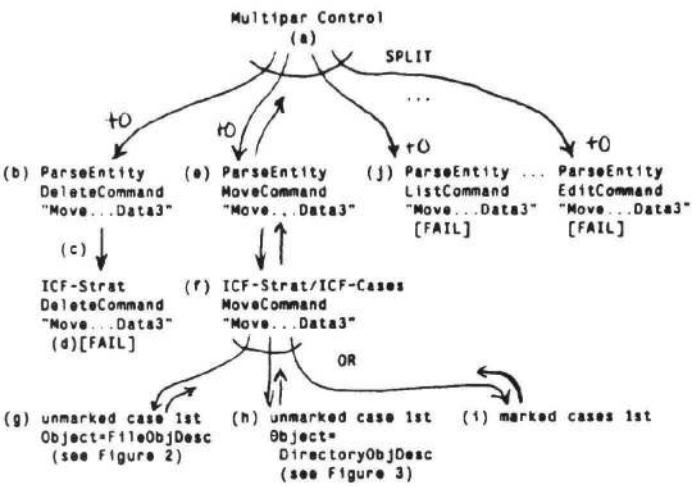

Figure 1.
Annotated Example, Steps 1 4

Step 2. Try to parse "Move...Data3" as a *DeleteCommand*.

ParseEntity (Figure 1,(b)) is a function that maps entity types to strategies. A request for a command entity could result in the trial of a number of different strategies. At present, only top-down versions of the parsing strategies exist, and calls to **ParseEntity** that look for commands are always mapped into calls to the imperative caseframe strategy (Figure 1,(c)). **ImperativeCaseFrame-Strat** will be unable to find an appropriate verb for the *DeleteCommand* entity and will fail without scheduling any alternate branches (i.e. this can be viewed as a non-recoverable error for the top-down strategy). Failure means that processing is continued by the control structure which eliminates this branch (Figure 1,(d)) and chooses another (Figure 1,(e)). Of course, the new branch also contains a call to **ParseEntity**; this call is mapped as above (Figure 1,(f)).

Step 3. Try to parse the input as a *MoveCommand*.

ImperativeCaseFrame-Strat knows how to use the *Icf-canonical SurfaceForm* to interpret the input. In the *MoveCommand* entity definition shown in section 3, this is the only *SurfaceForm*. As linguistic coverage is extended to include, for example, declaratives and interrogatives, new *SurfaceForms* must be defined. **ImperativeCaseFrame-Strat** could be expanded to interpret all top-level forms or each form could be provided with its own "expert".

The first action that **ImperativeCaseFrame-Strat** takes is to use the *Head* field in the *SurfaceForm* to search for a legal

[6] The grammatically correct version of this sentence is "Move to the accounts directory the file Data3."

[7] Hereafter, simply Figure 1,(a).

verb.[8] It finds "Move" and the unparsed segment is reduced to "the accounts directory the file Data3". The next step is to call a routine to fill the *SemanticCases* of the *MoveCommand*.

Step 4. Use **ImperativeCaseFrame-Cases** to fill *MoveCommand*'s *SemanticCases*.

ImperativeCaseFrame-Cases (Figure 1,(f)) is not a strategy itself but only a part of the top-down imperative caseframe strategy. The distinction is important because the responsibility of a strategy is to return an "instance list," i.e. one or more instantiated *InstanceTemplates*. **ImperativeCaseFrame-Cases** will return lists of consistent *SemanticCase* bindings which **ImperativeCaseFrame-Strat** will use to fill in *InstanceTemplates* when building its instance list. Even if **ImperativeCaseFrame-Cases** fails to fill any cases, **ImperativeCaseFrame-Strat** may return a non-empty instance list.

When filling cases we impose no order on their appearance in the utterance, nor do we fill required cases first (doing so would eliminate possible parses at flexibility levels greater than 0). As we try to expand a partial parse the still-unfilled cases may be constrained in the kinds of values they can take on by the values bound to those cases already filled. The *Constraints* field of the entity definition specifies the requirements. Of course, at this point no cases have been filled and no constraints apply. The unparsed segment, "the accounts directory the file Data3", is examined in two ways:

a. The first case we try to fill is the direct object which is unmarked. (Figure 1,(g) and (h))
b. The first case we try to fill is one of the marked cases. (Figure 1,(i))

Consider step 4.a. We are attempting to interpret some portion of the segment as the *Object SemanticCase* of the *MoveCommand*. Since the entity definition shows that an *Object* can be an instance of either a *FileObjDesc* or a *DirectoryObjDesc*, we will try each of these in turn (Figures 2,(a) and 3,(a)).

Step 5. Continue step 4.a.; try to parse *Object* as a *FileObjDesc*.

ImperativeCaseFrame-Cases wants to find a *FileObjDesc* in "the accounts directory the file Data3". To do so, it must call **ParseEntity** with a request for a noun phrase that can be interpreted as a *FileObjDesc*. Since there are two *SurfaceForms* for parsing nominal caseframes, each with its own associated strategy, this call to **ParseEntity** results in a SPLIT (Figure 2,(b)) We will examine only the path labelled **NominalCaseFrame-Strat-Canonical** (Figure 2,(c)).

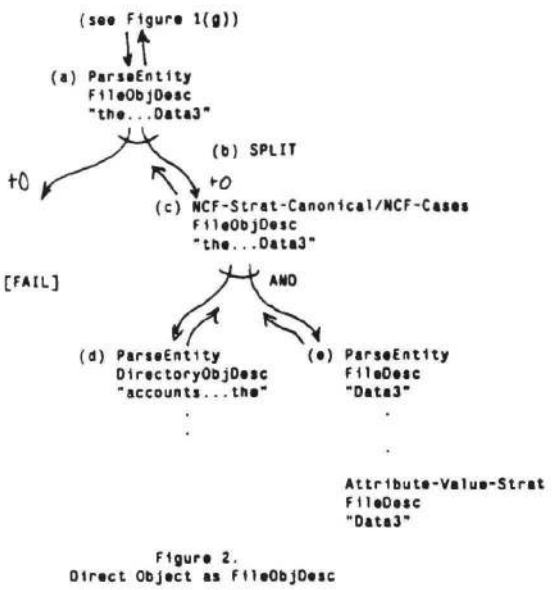

Figure 2.
Direct Object as FileObjDesc

Step 6. Find the *Head* and *Quantifier* of the noun phrase.

As with imperative caseframes, the first action taken to fill a nominal caseframe is to locate the *Head*. **NominalCaseFrame-Strat-Canonical** finds "file" as a possible head and breaks the remainder of the input into prenominal and postnominal segments. The strategy then looks for a quantifier or determiner at the left end of the prenominal segment and finds "the". "Accounts directory the" now constitutes the prenominal segment and "Data3," the postnominal segment. We call **NominalCaseFrame-Cases**, a sub-routine of **NominalCaseFrame-Strat-Canonical**, to begin filling cases from the prenominal segment (Figure 2,(c)).

Step 7. Parse the prenominal segment, "accounts directory the".

The only *FileObjDesc* case we will be able to fill from this segment of the input is *FileDirectory*, an instance of a *DirectoryObjDesc*. After a sequence of calls (Figure 2,(d)) a sub-invocation of **NominalCaseFrame-Strat-Canonical** will return an instance list with a single instance:

```
(IsA DIRECTORY
```

[8] The reader should keep in mind that at each step it is possible to have more than one interpretation of the input. Thus, if our input had been, "Move the file called transfer to dskb," the initial scan for a verb would have resulted in two partial parses -- one catching "move" and the other catching "transfer".

```
Name (accounts))[9]
```

Thus, we return to step 6 (Figure 2,(c)) with one case filled and the word "the" unused.

Step 8. Parse the postnominal segment, "Data3".

We are interested in extending any partial parses created in step 7 by using the postnominal segment to fill any unbound *SemanticCases*. Figure 2,(e) shows that a succession of calls results in filling the *FileName* using the Attribute-Value strategy.

Step 9. Return to step 4, Figure 1,(g).

Steps 7 and 8 produced one consistent set of bindings for two *SemanticCases* in the *MoveCommand* with one word leftover. Although there are unfilled cases, we have run out of input on the right. Thus, the instance list of *FileObjDescs* returned by the call to **ParseEntity** in Figure 2,(a) has only one element:

```
(IsA    FILE
 Name   (Data3)
 Directory  (accounts)
 Description (
       Quantifier (the)))
```

Referring to step 4.a. (Figure 1,(g)), the possible interpretations of the input such that the *Object* is a *FileObjDesc* have been exhausted. It remains to examine what happens when we look for an *Object* that is a *DirectoryObjDesc*. Again, we will call **ParseEntity**, split and suspend one of the calls, and examine the branch labelled **NominalCaseFrame-Strat-Canonical** (Figure 3,(a) through (c)).

Step 10. Continue step 4.a.; try to parse the *Object* as a *DirectoryObjDesc*.

We pick up the *head*, "directory," and the quantifier/determiner case as in step 6. This leaves the word "accounts" in the prenominal segment and "the file Data3" in the postnominal segment. The Attribute-Value strategy finds "accounts" as the *Name*. No other *DirectoryObjDesc* cases can be filled, so this step returns:

```
(IsA DIRECTORY
 Name (accounts)
 Description (
       Quantifier (the)))
```

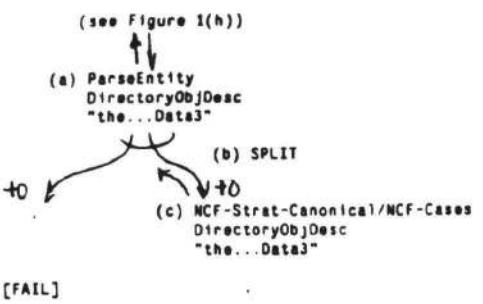

Figure 3.
Direct Object as DirectoryObjDesc

Step 11. Return to step 4.

Consider Figure 4,(a) which corresponds to the "OR" in Figure 1,(f). The computations shown in Figures 2 and 3 have given us two partial parses with the direct object filled; once by a *FileObjDesc* with the word "the" leftover and no input left to fill the other cases of the *MoveCommand* (Figure 4,(b)), and once by a *DirectoryObjDesc* with "the file Data3" leftover (Figure 4,(c)). As we try to extend this second partial parse, we have only marked cases remaining in the *SurfaceForm*. Since there is no marker at the beginning of the remaining segment we have encountered our first violated expectation. The recovery action associated with this failure is to hypothesize the existence of the missing case marker. Thus, for each of the three remaining *SemanticCases* we schedule a continuation that charges three flexibility points for the deviation (Figure 4,(d)). Note that if some other branch of the search tree with cumulative flexibility less than three succeeds in consuming the entire input segment, the branches just spawned will never be reactivated.

---

[9] The *InstanceTemplate* has many more fields in it; only those with a non-nil value are shown.

Step 12. Return to step 4.b. (Figure 4.(e))

Consider what happens when we try filling the marked cases of the *MoveCommand* first (Figure 1.(i) corresponding to Figure 4.(e)); the situation is identical to the one just outlined. For all cases other than the direct object, the segment "the accounts directory the file Data3" must have a left marker. Since it has none, we hypothesize a branch for each marked case at the current flexibility level plus three (Figure 4.(e)).

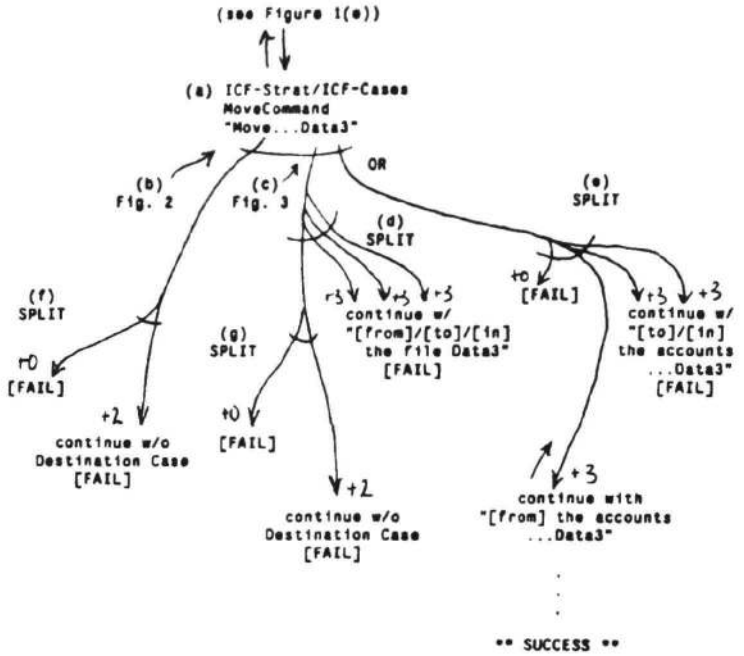

Figure 4.
Continuation of Figure 1 After
Processing Completed by Figures 2 and 3

Step 13. All the possibilities for the *MoveCommand* have been examined.

We have succeeded in finding two ways to fill the cases of the *MoveCommand*. Before we can return the partial parses from **ImperativeCaseFrame-Cases** to **ImperativeCaseFrame-Strat** we must check whether the *Required* cases, as specified by the *Constraints* field, have been filled. Indeed, each of the partial parses is missing the required *Destination* case, a violated expectation. The recovery action associated with this error is to suspend each of these partial parses and charge two flexibility points per missing required case for their continuation (Figure 4,(f) and (g)). Since no parses had all the required cases, the level 0 continuation of **ImperativeCaseFrame-Cases** returns a failure signal to **ImperativeCaseFrame-Strat.**

ImperativeCaseFrame-Strat returns an instance list with a single instance whose *Source* and *Destination* fields are nil. This signifies that the only part of the strategy that succeeded at level 0 was finding the verb. Since there is unused input, the top-level of MULTIPAR interprets this instance as a failure and signals this to the control.

Step 14. Exhaust level 0 of the agenda looking for a non-deviating parse.

The control structure takes over and continues in turn each branch suspended at level 0. Those containing requests for imperatives (Figure 1,(j)) fail immediately as in step 2. The other level 0 branches were left suspended by the SPLITs in Figures 2 and 3; these also fail.

Step 15. Exhaust levels 1 and 2 of the agenda.

Having tried all the branches in level 0 without success, the flexibility level is incremented and the control structure tries to choose a path suspended at level 1. Our example did not spawn any level 1 branches (single spelling corrections), so the flexibility level is incremented again. There are two branches at level 2, both in the same predicament (Figure 4, (f) and (g)); each has a missing required case and leftover input. If there had been no leftover input (as in "Move foo"), they would have succeeded at this level.[10] However, since no further recovery actions apply, each of these branches fails without adding to the control tree.

Step 16. The control increments the flexibility level to 3.

There are two sets of branches at this level:

a. The *Object* case is filled and the missing marker has been hypothesized before "the file Data3". (Figure 4,(d))
b. No cases are filled and the missing marker has been hypothesized before "the accounts directory the file Data3". (Figure 4,(e))

[10] Although consuming the entire input would have guaranteed success, note that if some branch with cumulative flexibility of 0 or 1 had succeeded, these branches would never have been retried.

Step 17. Continue 16.a. (Figure 4.(d))

The *Object* case of the *MoveCommand* has been bound to a *DirectoryObjDesc* with the *Name* field bound to "(accounts)". Hypothesizing the appropriate kind of marker for each of the remaining cases gives:

a. *Source*: Move the accounts directory [from] the file Data3.
b. *Destination*: Move the accounts directory [to] the file Data3.
c. *Location*: Move the accounts directory [in] the file Data3.

The single recovery action of hypothesizing a missing marker is not enough for any of these branches to succeed. Each would be rescheduled at least one more time. If MULTIPAR allowed the control structure to search the space indefinitely, 17.a. would eventually succeed at flexibility level 8 (3 points for a missing marker, 3 points for a constraint violation[11], and 2 points for a missing required case). 17.b. would eventually succeed at level 6 (1 missing marker, 1 constraint violation) and 17.c. at level 8 (1 missing marker, 1 missing required case and 1 constraint violation).

Step 18. Continue 16.b. (Figure 4.(e))

There are three branches remaining, one for each of the marked cases in the *MoveCommand*, with no cases yet filled. Allowing indefinite expansion, the following would occur:

Source:

a. Move [from] the accounts directory the file Data3.
   eventually succeeds at +5; 3 for missing marker, 2 for missing case
b. Move [from] the accounts directory [to] the file Data3.
   eventually succeeds at +8; 2 missing markers, 1 required case
c. Move [from] the accounts directory [in] the file Data3.
   eventually succeeds at +13; 2 missing case markers, 1 constraint violation and 2 missing required cases

Destination:

a. Move [to] the accounts directory the file Data3.
   *** SUCCEEDS at this level (level 3) ***
b. Move [to] the accounts directory [from] the file Data3.
   eventually succeeds at +8; 2 missing case markers and 1 missing required case
c. Move [to] the accounts directory [on] the file Data3.
   eventually succeeds at +13; same as *Source* c.

Location:

a. Move [in] the accounts directory the file Data3.
   eventually succeeds at +5, 1 missing marker and 1 missing case
b. Move [in] the accounts directory [to] the file Data3.
   eventually succeeds at +8, 2 markers, 1 case
c. Move [in] the accounts directory [in] the file Data3.
   eventually succeeds at +13, same as *Source* c.

Step 19. A successful path is found at level 3.

Hypothesizing the existence of a marker for the *Destination* enables **ImperativeCaseFrame-Strat** to continue the second branch of Figure 4.(e). Now "the accounts directory" can be picked up as the *Destination* and "the file Data3" as the unmarked direct object. Since both required cases are bound and no input remains, MULTIPAR return the following instance as its representation of the input:

```
(Action MOVE
 Deviations (MissingMarker Destination)
 Source (
       IsA FILE
       Name (Data3)
       Description (
              Quantifier (the)))
```

[11] In *Constraints:(Object DirectoryObjDesc > Source LogicalDeviceObjDesc)*.

```
Destination (
        IsA FILE
        Name (Data3)
        Directory (accounts)
        Description (
                Quantifier (the))))[12]
```

5. Conclusion

MULTIPAR is not a detailed cognitive model of human language processing. It is an attempt to emulate the performance of humans in comprehending natural language utterances that deviate from strict grammatical standards. MULTIPAR uses multiple parsing strategies, and is driven by a "grammar" of descriptions of entities relevant to the domain of discourse. The multiple strategies are able to interpret the entity definitions in a variety of ways. Some of the ways depend on the surface language constituents being in the grammatically correct place. Other ways, though more computationally expensive, relax the grammatical constraints, and are thus able to handle grammatically deviant input. To control the potential for exponential growth in the search space of a parser that accepts ungrammatical as well as grammatical input, MULTIPAR incorporates a control mechanism that allows possible parses to be explored in order of increasing degree of ungrammaticality. All of these features of MULTIPAR are essential to its performance as a robust natural language parser.

References

1. Boggs, W. M. and Carbonell, J. G., Monarch, I., and Kee, M., "The DYPAR-I Tutorial and Reference Manual," Tech. report, Carnegie-Mellon University, Computer Science Department, 1985.
2. Carbonell, J. G., "Towards a Self-Extending Parser," *Proceedings of the 17th Meeting of the Association for Computational Linguistics*, 1979, pp. 3-7.
3. Carbonell, J. G. and Hayes, P. J., "Dynamic Strategy Selection in Flexible Parsing," *Proceedings of the 19th Meeting of the Association for Computational Linguistics*, 1981.
4. Carbonell, J. G. and Hayes, P. J., "Recovery Strategies for Parsing Extragrammatical Language," *American Journal of Computational Linguistics*, Vol. 9, No. 3-4, 1983, pp. 123-146.
5. Carbonell, J. G. and Hayes, P. H., "Robust Parsing Using Multiple Construction-Specific Strategies," in *Natural Language Parsing Systems*, L. Bolc, ed., Springer-Verlag Publishers, 1985.
6. Dejong, J. F., *Skimming Stories in Real Time: An Experiment in Integrated Understanding*, PhD dissertation, Yale University, May 1979.
7. Granger, R., "FOUL-UP: A Program that Figures Out Meanings of Words from Context," *Proceedings of IJCAI-77*, 1977, pp. 172-178.
8. Hayes, P. J., "Entity-Oriented Parsing," *COLING84*, Stanford University, July 1984.
9. Hayes, P. J. and Carbonell, J. G., "Multi-Strategy Parsing and its Role in Robust Man-Machine Communication," Tech. report CMU-CS-81-118, Carnegie-Mellon University, Computer Science Department, May 1981.
10. Kwasny, S. C. and Sondheimer, N. K., "Ungrammaticality and Extra-Grammaticality in Natural Language Understanding Systems," *Proc. of 17th Annual Meeting of the Assoc. for Comput. Ling.*, August 1979, pp. 19-23.
11. Kwasny, S. C. and Sondheimer, N. K., "Relaxation Techniques for Parsing Grammatically Ill-Formed Input in Natural Language Understanding Systems," *American Journal of Computational Linguistics*, Vol. 7, No. 2, 1981, pp. 99-108.
12. Lebowitz, M., *Generalization and Memory in an Integrated Understanding System*, PhD dissertation, Yale University, Oct. 1980.
13. Weischedel, R. M. and Sondheimer, N. K., "Meta-Rules as a Basis for Processing Ill-formed Input," *Computational Linguistics*, Vol. 10, 1984.
14. Weischedel, R. M. and Black, J., "Responding to Potentially Unparseable Sentences," *American Journal of Computational Linguistics*, Vol. 6, 1980, pp. 97-109.

[12] The reader may have noticed that the directions for filling in the *MoveCommand*'s *InstanceTemplate* would give different values for the *Destination* fields. The *InstanceTemplate* shown in section 3 has been simplified for illustrative purposes. The real *InstanceTemplate* has considerably more complex directions for filling the fields; from those directions (which include conditionals that test other fields) the instance shown above would be constructed.