An Efficient Method for Parsing Erroneous Input

Stuart Malone and Sue Felshin
Athena Language Learning Project
Massachusetts Institute of Technology

The Athena Language Learning Project receives major funding from the Annenberg/Corporation for Public Broadcasting Project.

Abstract

In a natural language processing system designed for language learners, it is necessary to accept both well-formed and ill-formed input. This paper describes a method of maintaining parsing efficiency for well-formed sentences while still accepting a wide range of ill-formed input.

1. Introduction

The Athena Language Learning Project is developing advanced educational software for foreign language learners. One of the tools we are developing is a natural language parser for use by first through fourth semester students of various languages. This parser must be able to recover from and correct a wide range of morphological, syntactic, and semantic errors, and yet still run in real time. We have designed a system where all of these errors can be handled by the parser uniformly and efficiently.

2. Description of the Parser

Our parser is a nondeterministic LALR(1) parser, written in Common Lisp, similar to that of Tomita [1] but differing in several significant ways.

- First, we associate a **reduction function** with each rule in the grammar. Whenever a reduce action is performed by the parser it calls the corresponding reduction function, which constructs the new node of the parse tree from the nodes on the right side of the production. As it does this, it may perform various tests on its input and either mark errors on the new node or, rarely, return NIL to fail. This is similar to the system of relaxation of predicates used by Weischedel and Black [2] and others; we mark errors where they allowed predicates to be relaxed, and return NIL from reductions where their predicates failed.

- Second, in order to properly handle linguistic phenomena like movement and binding, we needed to make the parser context sensitive. We did this by associating context sensitive information about a parse with each parse stack. This information is passed in to each reduction function, which examines and modifies the information as appropriate in order to build the new node.
- Third, we did not want the parser to return *every possible* parse of the student's input, given the relaxed rules of our grammar. A strict grammar for a natural language already has to consider many possible parses of the input—allowing erroneous input increases the problem by an order of magnitude or more. Computing all these parses would be a waste of time, and would make the system unusably slow. Instead, we only want the parser to return the "most likely" parses.
- Fourth, we decided that it was essential for our parser to perform semantic analysis at the same time as syntactic analysis in order to reduce ambiguity. Even though syntactic errors are common for language learners, semantic errors are more unusual. If parsing can be guided by semantic constraints as well as syntactic ones, then we can expect to come up with the better interpretations of the student's input with less work.

Adding context sensitive information to each parse stack had the significant disadvantage that it became impractical to use some of Tomita's more sophisticated techniques such as graph-structured stacks and local ambiguity packing.[1] However, abandoning these techniques allowed us to take advantage of a different one: a best-first searching strategy. Creating graph-structured stacks requires the use of breadth-first search in order to keep all of the parse stacks synchronized on the input. Without graph-structured stacks, it becomes possible to advance different parses of a sentence at different rates, forging ahead with parses that look promising, and postponing work on less likely ones.

3. Marking Errors

In our parser, every word and every node in the parse tree contains an error-count which is initially zero. Whenever an error is detected, our code increases the error-count of the word or node and attempts to generate a plausible corrected node.

Four kinds of errors are detected by the lexical analysis pass of the parser, and are marked on individual words before they are parsed.

errors in the lexicon	Some errors are so common that we have anticipated them by entering them directly into the lexicon. For instance, use of the wrong gender ending on a noun in Spanish, e.g., "abriga" for "abrigo" (*"overcoat"*). Lexical lookup returns a word marked with an error.
spelling errors	When regular lexical lookup fails, we run a spelling checker to search for known words with similar spellings. Each misspelled word is marked with an error.

[1]This was because, in order for two parse stacks to be merged, the context sensitive information associated with each stack had to be compatible. This situation was so rare that the bookkeeping involved wasted more time than was saved.

blocked word errors — Irregular forms of words are stored in the lexicon as subentries of their regular forms. After lexical lookup, a second pass checks the irregular subentries of the returned word to make sure none of them should have been used instead. If it determines that one should have, it marks the word with an error before returning it. For example, the Spanish word "tenió" in place of "tuvo", or the English word "haved" in place of "had".

surface filter errors — The surface filter looks at the stream of words returned by lexical lookup and performs arbitrary surface operations, such as splitting "compound" words into their components, combining single meanings given by more than one word, and insuring that words are properly contracted. As it does this, it marks any errors it finds on the appropriate words. In English, for instance, a surface filter checks for correct "a/an" alternation as in "a dog" vs. "an apple".

It is important to understand that the lexical analysis pass may return several different interpretations for a single word, some of which may have errors while others may not. For instance, in English the word "seed" could either be the correct singular form of the noun "seed" or the incorrect (blocked) past tense of the verb "to see".

Three other kinds of errors are detected during parsing and are marked on nodes by the reduction functions that create those nodes.

structural errors — The grammar productions anticipate certain structural errors, similar to the way that the lexicon anticipates certain lexical ones. For instance, Spanish detects improper use of preposition-like words, e.g., "encima la mesa" instead of "encima de la mesa" (*"on (top of) the table"*).

agreement errors — The reduction functions mark errors as appropriate for any context dependent and/or independent syntactic requirements which are violated by the current constituents. In English, the noun phrase "a books" would be marked with an agreement error and assumed to be plural.

semantic errors — The reduction functions also access the case frame interpreter, which builds semantic structure and marks any necessary errors. Even though the semantic structures are separate from the syntactic nodes of the parse tree, semantic errors are marked on the nodes of the parse tree so that they will be visible to the parser.[2]

4. How Parsing Proceeds

Now that we have explained how we mark errors on the words and nodes of a parse tree, we can explain how these errors are used by the parser to direct parsing. At this point it is helpful to introduce a term for the information that is stored about a partially-completed parse. We call this infomation the parse state, or **pstate** for short. A pstate contains the following information:

---

[2]The semantic structures built by the case frame interpreter introduce a new level of ambiguity—each represents any number of possible semantic interpretations of the constituent. The error-count of a case frame is the error-count of its best interpretation.

- The traditional LALR parse stack of alternating nodes and state numbers.
- The current word of the input, which is the cu nt look-ahead token for the LALR parser and will be the next word shifted onto the parse stack.
- An error-count, which is used to determine which pstate is "best" in the best-first search for a successful parse.
- Context-sensitive infomation which varies from language to language.

The most interesting part of a pstate for this discussion is the error-count, which is used to direct the best-first parsing. The error-count of a pstate is the sum of the error-counts of the nodes in its parse stack, plus the error-count of its current word.

The parser keeps a sorted list of pstates. Pstates with the same error-count are ordered arbitrarily. Each step through the parser pops the first (best) pstate off of this list and looks up the next actions for the pstate in the LALR tables. For each action, the parser does the following:

- If the action is a shift action, it shifts the current word onto the parse stack. A new pstate is created for each possible following word, and the following words are made the current words of the new pstates. Each new current word's error-count is added to the error-count for its pstate.
- If the action is a reduce action, then the arguments to the reduction, which are the right side constituents, are popped off the parse stack and passed to the reduction. If the reduction constructs a result node, a new pstate is created, the node is pushed onto its parse stack, and the node's error count is added to the pstate's error-count.

Each pstate created during the above procedure is inserted in its proper position in the list of pstates, and the procedure is repeated with the new best pstate. This continues until either the list of pstates becomes empty, in which case parsing has failed, or enough pstates parse to completion that the remaining (worse) pstates are simply thrown away. To determine when to throw away pstates, we maintain a range which we call the **style threshold**. Whenever the error-count of a pstate becomes larger than the error-count of the best successful parse plus the style threshold, that pstate is removed from the list of pstates and thrown away. However, no pstates are thrown away until there is a successful parse.

4.1. An Example

As an example of how this system works, we'll describe the parsing of the sentence "Dije donde llovió," which is incorrect Spanish for *"(I) said where (it) rained."* "Donde" with no accent is a subordinating conjunction, as in "I'll go *where* you go." "Dónde", with an accent, is a pro-PP introducing a complement clause, as in "I said *where* it rained." Unsurprisingly, students of Spanish use the wrong form quite often. Lexical analysis of "donde" in our system returns two words, the subordinating conjunction and the pro-PP, the latter marked with an error-count of 500 for the lack of an accent.

The parser starts with a single pstate, call it A, where the current word is "dije", the first word in the input sentence. The grammar first pushes various empty nodes onto the parse stack, including an empty COMP and a pro subject, and eventually shifts "dije". Since the next word, "donde", is ambiguous, the parser must now split this pstate into two new pstates, B and C. Pstate B receives the subordinating conjunction as its current word, and has its error-count

increased by 0. Pstate C receives the pro-PP, and has its error-count increased by 500. Processing of pstate C is then postponed because it's not the best available pstate.

Parsing continues with pstate B. The subordinate clause "donde llovió" is completed and attached to the S node dominating "dije". Now the sentence is ready to be finished off. But finishing it off precludes the possibility of more arguments being parsed, and the verb "dije", which requires a direct object or complement clause, has received neither. The case frame interpreter marks an error of 600 for a missing argument, and when this is added to pstate B, it is no longer the best pstate. Thus pstate B is now postponed in favor of pstate C.

Parsing of pstate C now resumes at the point it was left off, and by parsing "donde llovió" as a complement clause, continues to a successful completion. If the style threshold is less than 100, parsing will stop, and pstate C, with an error-count of 500, will be returned. Otherwise, parsing of pstate B will resume until it is successfully completed with an error-count of 600, and both pstates B and C will be returned.

5. Anticipated Errors

Because the error-count of a pstate determines whether or not the pstate should be actively pursued, postponed, or thrown away, it is vitally important for error-counts to be accumulated as soon as possible.

Take, for example, the sentence "You drink too much beer." This sentence has two interpretations: the obvious one, and an erroneous one where a case-blocking modifier has been placed between the verb and the direct object (this reading should be "You drink beer too much"). The error in the second interpretation is in the *placement* of the ADVP "too much" within the VP—there is nothing wrong with the ADVP itself. Conceptually, therefore, the error should be marked on the VP. But before this VP can be built, the erroneous modifier and the direct object must be parsed; this will waste a lot of work before this reading's pstate is postponed in favor of the first reading, and eventually discarded unfinished due to the style threshold. Alternatively, we can build a modifier node around the ADVP node and mark the error there, saving the time it takes to parse the direct object. Or best of all, before starting to build the ADVP in the first place, we can build an empty node and mark the error on it. Thus this pstate will be postponed as soon as the empty error node is created—before either the ADVP *or* the direct object have been parsed.

We call these errors, which are marked on empty nodes *before* the erroneous input, **anticipated errors**. In many ways, anticipated errors are the most important category of error, because they have the greatest influence on the speed of parsing. Anticipated errors allow us to postpone or discard a pstate *before* we have wasted a great deal of time on it.

We handle many structural errors by writing explicit productions to parse ill-formed input, and these errors can always be anticipated. For instance, in our system we can write a rule such as:

```
VP => VBAR MOD,BAD-MOD? OBJ
```

This rule says: "To parse a VP, parse a VBAR, optionally followed by a MOD marked with the BAD-MOD error, followed by an OBJ." This is automatically expanded by our LALR table generator into:

```
VP => VBAR MOD,BAD-MOD? OBJ

MOD,BAD-MOD? =>
MOD,BAD-MOD? => MOD,BAD-MOD

MOD,BAD-MOD => BAD-MOD MOD

BAD-MOD =>                          (create an empty error node)
```

These rules will mark the partially completed parse with an error as soon as the parser decides that there is a MOD after the VBAR, *before* the MOD, OBJ, or VP has been created. Since parsing is best first, this partial parse will be postponed until it is the best parse available—which may never happen. However, if the parse *does* become the best available, the work done to construct the VBAR will not have been wasted. Processing of this partial parse will continue right where it left off.

Some structural errors are too complex to anticipate through the use of the LALR table generator's error facility. For example, in Spanish, infinitive sbar complement clauses must be introduced with one of two different complementizers, "a" or "de"; or with no complementizer at all, depending on the higher verb. English speakers, who are accustomed to always using the particle "to", frequently choose the wrong complementizer in Spanish. Since all three structures (either complementizer or none at all) are potentially correct in Spanish, there is no place in any production to mark the error.

We can still anticipate the error, however, by having the reduction function called when the complementizer is reduced look at the higher verb. We can find the higher verb using the context-sensitive information that is kept with each pstate. We call the case frame interpreter to determine whether the higher verb allows an SBAR complement clause, and if so, whether the verb allows the given complementizer. We push a small error if the wrong complementizer was used and a very large error if no clause is allowed at all.

6. Conclusions

We have written grammars for Spanish, English, French, German, Russian, and Classical Greek. The most comprehensive of these is for Spanish, where the grammar contains over five hundred context-free productions.

The following data demonstrates the time saved by using best-first parsing and by trimming unlikely pstates through use of the style threshold. Time is measured by the total number of reductions performed during parsing the input, counting both successful and trimmed pstates. We generally set the style threshold at 30 to allow for slight variations in interpretations of the input. Setting the style threshold to a very large number (such as 10,000) approximates what would happen if an equivalent grammar were run using Tomita's parser.

Dije donde llovió.

"(I) said where (it) rained."

Style threshold	# of parses	# of reductions
0	1	680
30	1	680
100	2	1,010
10,000	19	3,549

Hace buen tiempo en Bogotá.

"(It) makes good weather in Bogota."

Style threshold	# of parses	# of reductions
30	1	831
10,000	22	7499

Acknowledgements

We wish to thank Professor Robert C. Berwick and Dr. Janet H. Murray of MIT for supporting our research and convincing us to write this paper.

References

[1] Masaru Tomita.
Efficient Parsing for Natural Language.
Kluwer Academic Publishers, Boston, 1986.

[2] Ralph M. Weischedel and John E. Black.
Responding Intelligently to Unparsable Inputs.
American Journal of Computation Linguistics 6(2):97–109, April–June, 1980.