

A Parser from Antiquity¹

Aravind K. Joshi and Phil Hopely

¹ We want to thank Lauri Karttunen, Mark Liberman, Mitch Marcus, Mehryar Mohri and B. Srinivas for their valuable comments during the preparation of this paper. We are grateful to William Schmidt for providing some information about Univac 1 and also to the Hagley Museum, Wilmington, Delaware for providing access to some early documentation about Univac 1. The first author wishes to thank Eric Brill for providing him an opportunity to make an early report of this work at the Empirical Issues in Natural Language Processing Workshop, held at the University of Pennsylvania, in May 1996, as a part of an year-long celebration of the 50th anniversary of Eniac. Many comments on the report at this workshop were also very helpful in the preparation of this paper.

Abstract

This paper describes the key aspects of a parser developed at the University of Pennsylvania from 1958 to 1959. The parser is essentially a cascade of finite state transducers. To the best of our knowledge, this is the first application of finite state transducers to parsing. This parser was recently faithfully reconstructed from the original documentation. Many aspects of this program have a close relationship to some of the recent work on finite state transducers.

1 Introduction

A parsing program was designed and implemented at the University of Pennsylvania during the period from June 1958 to July 1959. This program was part of the Transformations and Discourse Analysis Project (TDAP) directed by Zellig S. Harris. The techniques used in this program, besides being influenced by the particular linguistic theory, arose out of the need to deal with the extremely limited computational resources available at that time. The program was essentially a cascade of finite state transducers (FSTs). To the best of our knowledge, this is the first application of FSTs to parsing. The program consisted of the following phases:

1. Dictionary look-up.
2. Replacement of some ‘grammatical idioms’ by a single part of speech.
3. Rule based part of speech disambiguation.
4. A right to left FST composed with a left to right FST for computing ‘simple noun phrases.’
5. A left to right FST for computing ‘simple adjuncts’ such as prepositional phrases and adverbial phrases.
6. A left to right FST for computing simple verb clusters.
7. A left to right ‘FST’ for computing clauses.

In Section 2 we will describe the different phases of the parser in some detail and also briefly discuss several aspects of the parser that have a close relationship to some of the recent work on finite state transducers. An illustrative example is provided in Section 3 showing the output of each phase. This is followed by a brief description of the reconstruction and evaluation of the parser in Section 4. Finally, some historical information is provided in Section 5.

2 Some details of the parser

We will describe here the various phases of the parser, illustrated with a few examples for each phase.

Phase 1: Each word is assigned one or more parts of speech (POS). If a word is assigned more than one POS, then sometimes they are ranked, the less frequent POS first and then the next. Thus for example, for example, *show* has N ranked before V and *book* has V ranked before N. There are about 14 subcategorization frames for verbs. Since the Prepositional Phrases (PPs) are marked with the

specific prepositions, there are effectively over 50 verb subcategorizations. The parser does not handle unknown words.

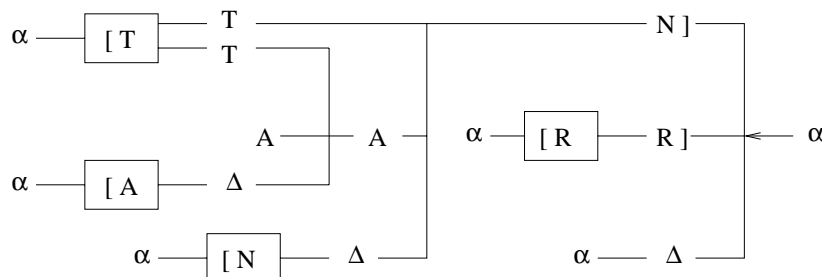
Phase 2: A ‘grammatical idiom’ such as *of course* is replaced by a single POS for adverb, *per cent* by POS for noun, etc. Certain ‘grammatical idioms’ such as *per cent* were replaced by a single POS with certainty. Others, such as *of course*, were replaced by a single POS with an indication that this replacement is tentative because, for example, *of* and *course* may belong to different phrases. For each ‘grammatical idiom’ one word of the idiom is marked as an index into the idiom-dictionary which specifies the local environment (words to the left and to the right of the index word). Phase 2 operations consist of finite transductions.

Phase 3: Rule based disambiguation techniques are used for POS disambiguation. There are about 14 tests: N-eliminating tests, V-eliminating tests, etc. If the POS for a word are ordered, for example, for *show* N before V, then the N-eliminating tests are applied first. If they fail, then the V-eliminating tests are applied. If these also fail then the ambiguity remains. Most tests look for bounded contexts to the left and to the right; thus these are finite transductions. Some tests use contexts specified by simple (i.e., no Kleene-* over another Kleene-*) regular expressions and thus they are also finite state transductions. The ordered set of tests, X - eliminating tests for a POS X, are cycled until no further disambiguations can be made.

Phases 4,5 and 6: The strings (phrases) in these phases are called first-order strings as they do not involve proper nesting, as compared to the strings (clauses) computed in Phase 7, which are called second-order strings as they may involve proper nestings. The computation in Phase 7 is, strictly speaking, not a finite state computation.

In Phase 4 a right-to-left (RL) FST is composed with a left-to-right (LR) FST for computing simple noun phrases. A highly simplified diagram of a RL FST for simple noun phrases in the original notation is shown in Figure 1.

The sentence is scanned from right to left. A closing bracket] is placed as soon we meet a symbol (a part of speech (POS)) that allows us to enter the graph in Figure 1. The possible entrance points are N and R. Once we enter the graph, control flows according to the symbols encountered while scanning from right to left. If the graph terminates on a box then we place the opening bracket [; if the graph terminates on a POS symbol, we loop back to the place on the graph where that POS appeared before for the first time. Note that the longest path strategy is built into the graph, in the sense that once we enter the graph we try to absorb as many of the POS symbols encountered while traversing the graph. This longest path strategy is also used (not shown in Figure 1) if a word is encountered with ambiguous POS. In this case the POS selected is the one which allows continuation of the phrase; however, information about the alternatives is left behind for later processing. Thus some unresolved POS ambiguities from Phase 1 are temporarily resolved during the computation of these first-order strings. Similarly, if a conjunction is encountered and the phrase can be continued by looking one



N: noun, R: pronoun, T: article, A: adjective,
 Δ : any symbol other than those that appear above Δ .

Figure 1: A highly simplified diagram of a RL FST for simple noun phrases (original notation)

symbol to the left of the conjunction (for the RL FST) then it is continued.

If we represent the graph in Figure 1 in terms of the modern notation for sequential FST we obtain a RL FST as shown in Figure 2. This FST is strictly not sequential. It is subsequential because upon exiting from some final states the FST produces an output.

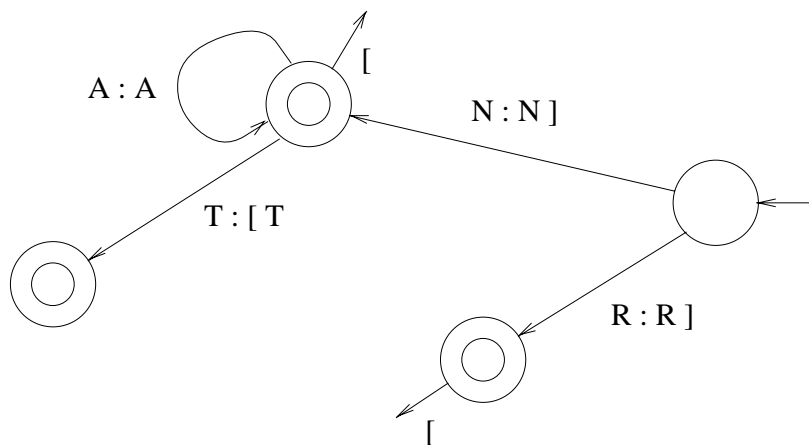


Figure 2: RL subsequential FST corresponding to Figure 1

This RL FST computes simple noun phrases, enclosed in [], whose rightmost end is determined. This transduction is followed by a LR FST which computes

simple noun phrases such as *the rich* where the leftmost end is reliably determined. Of course, if the actual string was *the rich man* the first RL FST would have already enclosed this string in [], thus making the string *the rich* ignorable in the second LR FST.

The total number of states in the RL FST is approximately 160 and in the LR FST approximately 50.

In Phase 5 a LR FST computes simple adjuncts such as prepositional phrases and adverbial phrases, enclosed in (), for example:

(very clearly)
 (rapidly)
 (to [date])
 (in [increased production])

In (*in [increased production]*) the LR FST cannot place (before *in* unless it is followed by a noun phrase []. However, by composing this FST with an RL FST (for simple noun phrases discussed earlier), where both FSTs are sequential, this can be easily accomplished. The total number of states in this FST is approximately 40.

In Phase 6, a LR FST computes simple verb clusters. If simple adjuncts, enclosed in (), are encountered during the computation, they are skipped over. Explicit attachment of adjuncts is not shown in the output. Simple verb clusters include verbal and infinitival complements of verbs. This FST checks to see that these complement requirements are satisfied. Simple verb clusters are enclosed in { }. This LR FST has 65 states approximately. Some examples are:

{went}
 {has gone fishing}
 {may have been published}
 {may have been (already) published}
 {have been observed and reported to be}
 {wants to leave}
 but not
 {wants [the man] (from [Philadelphia]) to leave }
 which will be bracketed as
 {wants} [the man] (from [Philadelphia]) {to leave}

Phase 7: The computation in this phase is strictly speaking not a finite state computation. The strings (phrases) computed in this phase are called second order strings as they may involve nesting. In the original parser there were two implementations of this phase. The first implementation is equivalent to a push-down automaton. In the second implementation we look for the most embedded clause, compute this with a LR FST and then repeat this process, starting from the beginning of the sentence, until the main clause has been computed. Clauses already computed are skipped while computing clauses that embed them. In either case, as far as strings enclosed in () are concerned, they are either skipped as

adjuncts or they are used as complements if appropriate. Attachment of clauses as well as attachment of adjuncts is not shown explicitly in the output.

While looking for verb complements during the computation of clauses the longest complement is preferred—that is the longest path strategy is followed. The end of a complement is marked by +. Complements of a verb cluster are taken as the complements of the last verb in the cluster. Clauses are enclosed in <>. Clauses headed by *that* and sentential subjects and complements are enclosed in /\ . The main clause is not enclosed in any brackets. Some examples are:

[Those] <who {read} [newspapers] + > {waste} [their time] +
<Since [the conviction] was based (on [evidence]) + > ...

The FSTs described in Phases 1 through 6 were made ‘effectively’ deterministic by (1) choosing the direction of the scan (left to right or right to left) and adopting the longest path strategy, (2) cascading right to left and left to right transducers, and (3) using the delimiting characters to allow for some minimal nondeterminism. These aspects of the program have a close relationship to some of the recent work on FSTs such as subsequential machines (Mohri 1996; Schützenberger 1977), decomposition of a FST into two sequential FSTs (Elgot and Mezzi 1965; Mohri 1996) and the work on ‘directed replacement’ (Karttunen 1996). The parsing style itself has resemblance to Abney’s chunking parser (Abney 1991). The first-order strings in Phases 4, 5, and 6 (enclosed in [], (), and { } respectively) are ‘chunks’ in Abney’s sense.

3 A detailed example

The overall objective of the program was to prepare the text for tasks such as abstracting. However, the 1958-59 program only performed the parsing task. Besides parsing a large number of test sentences (about 200), the program processed about 25 sentences from a journal paper in biochemistry. Although the FSTs compute more structure, the final output shows relatively flat structures. Adjuncts are never explicitly attached. Here is an example from the original set of sentences:

(1) We have found that subsequent addition of the second inducer to either system after allowing single induction to proceed for 15 minutes also results in increased reproduction of both enzymes

There are no grammatical idioms in this example. In Phase 3, *results* (N/V) is resolved to V. After the first three phases, in Phase 4 the first (right to left) FST identifies the following simple NPs in this example, enclosed in [].

(2) [We] have found that [subsequent addition] of [the second inducer] to [either system] after allowing [single induction] to proceed for [15 minutes] also results in [increased reproduction] of [both enzymes]

The next (left to right) FST does not identify any new simple NPs in this example. It would have found NPs such as [*the rich*].

Then the next (left to right) FST (Phase 5) identifies the following simple adjuncts in this example, enclosed in ().

(3) [We] have found that [subsequent addition](of [the second inducer]) (to [either system]) after allowing [single induction] to proceed (for [15 minutes]) (also) results (in [increased reproduction]) (of [both enzymes])

The final (left to right) FST (Phase 6) identifies the following simple verb clusters in this example, enclosed in { }.

(4) [We]{ have found } that [subsequent addition](of [the second inducer]) (to [either system]) after { allowing } [single induction] {to proceed} (for [15 minutes]) (also) {results} (in [increased reproduction]) (of [both enzymes])

Then the left to right scan (strictly not finite state) of Phase 7 identifies the clauses, enclosed in < > and / \ . The main clause is not enclosed in any brackets. + indicates the end of a complement. Thus the final output is as follows:

(5) [We] {have found} / that [subsequent addition](of [the second inducer])(to [either system]) < after {allowing} [single induction] to proceed + > (for [15 minutes]) (also) {results} (in [increased reproduction]) + > \+ (of [both enzymes])

In each one of these phases the longest path criterion is used. This results in maximal extensions of simple NPs, simple adjuncts, simple verb clusters and clauses. While looking for verb complements the longest complement is preferred.

4 Reconstruction and evaluation

Recently this program was faithfully reconstructed, a collaborative effort with Phil Hopely¹, from the original documentation (about 200 pages consisting of about 50 flowcharts and 150 pages of typed text), which fortunately exists (TDAP 1959-60), and is in sufficient detail to make the reconstruction possible. During the reconstruction we encountered a few places where there was some ambiguity in the documentation. In these cases, we did the reconstruction conservatively, i.e., only up to a level justified by the test sentences and their corresponding outputs as described in the documentation. The same criterion was used when filling

¹ A user interface method available for Uniparse is provided as a 'common gateway interfacing (cgi)' script, which permits live access to uniparse via html browser. As of this writing the script may only be run by local users, but negotiations are underway to allow relatively unrestricted access from outside of Penn's firewall. The latest information about the cgi is available from <http://www.cis.upenn.edu/~phopely/tdap-fe-post.html> An optional extension to the original system, which will be available soon via the web-page interface, permits Uniparse to deal with unknown words by using the XTAG lexicon for generating Uniparse dictionary entries (to be sure, only approximately sometimes), thus avoiding having to enter this information manually. The web page contains the latest availability information.

in for the faded parts of some flowcharts, which happened in two cases. The reconstructed parser (now called Uniparse (for Univac/UPenn Parser), for future reference) has been implemented in the C language in order to preserve the architecture of the original program. Uniparse features a highly pipelined architecture, where each element of the pipeline corresponds approximately to one program tape of the original system.

Uniparse has also been tested on about 50 sentences from each of the three corpora—Wall Street Journal (WSJ), IBM computer manuals, and ATIS. Since Uniparse does not handle unknown words, all words not known to Uniparse had to be provided with POS and subcategorization information. This information was limited to exactly the information allowed by Uniparse. This was necessary because all the grammatical information is built in Uniparse—wired in, in a sense. In principle, Uniparse cannot use any information that is not already built in. However, we did not provide any information for a new word that was not in Uniparse in order to avoid unintended interactions of this information with the reconstructed program. Limiting the information for each new word in this way, of course, both hurts and helps Uniparse. It hurts when the intended reading of a sentence requires information that is not available to Uniparse. It helps when the information required for the intended reading is built in Uniparse and Uniparse does not deal with other readings requiring information that is not built into it.

Evaluation of performance was carried out in two parts—for simple noun phrases and for parses.

Simple Noun Phrases:

- WSJ— 50 sentences (average length: 12 words)
- Simple Noun Phrases in Penn Treebank: 179
- Simple Noun Phrases found by Uniparse: 175
- Out of the 175 phrases found by Uniparse, 151 are exact matches to the corresponding phrases in Penn Treebank.
 - Exact matches found by Uniparse: approximately 81 %
- Of the remaining 24 phrases found by Uniparse, 22 phrases match the corresponding Penn Treebank phrases at one or the other end of the phrase but not at both ends.
- Corresponding performance for exact matches with respect to Penn Treebank for
 - IBM computer manuals: approximately 84 %
 - ATIS: approximately 89 %

Parses:

Evaluation of parses is somewhat problematic. As we have pointed already, the attachment of adjuncts as well as clauses is not shown explicitly by Uniparse. Further Uniparse computes verb clusters and not verb phrases, thus systematically producing more crossing brackets when compared to the Penn Treebank. Hence for the purpose of the evaluation, we consider a parse given by Uniparse

as being consistent with the corresponding parse in the Penn Treebank if, modulo attachments of adjuncts, clauses match exactly.

In the original program only a very limited exploration of alternatives (backtracking) was carried out, although in each phase, while following the longest path strategy, the FSTs leave behind information about possible alternatives. The current implementation optionally permits more exploration of alternatives, in particular, a parallel breadth first search over alternative subcategorizations and first-order brackets is performed, although still consistent with the spirit of the original program because only the information available within the original system is used. There is no particular significance to the order in which these alternatives are explored².

In order to make a fair comparison for evaluating Uniparse, we considered two criteria: (a) The ‘preferred’ parse, i.e., the parse that would have been produced by the original program (b) The ‘preferred’ parse and only the first parse returned during the exploration of the alternatives. By ‘first’ we do not mean anything special. It is just the first parse returned by the alternatives exploration system.

- WSJ– 50 sentences (same as above)
- Criterion (a): Only the ‘preferred’ parse of Uniparse is considered.
 - Parses consistent with the Penn Treebank: approximately 50 %
- Criterion (b): The ‘preferred’ parse and only the ‘first’ parse returned during the exploration of the alternatives are considered.
 - Parses consistent with the Penn Treebank: approximately 75 %
- Corresponding performance for IBM computer manuals and ATIS
 - IBM computer manuals: Criterion (a): 54 % Criterion (b): 75 %
 - ATIS: Criterion (a): 56 % Criterion (b): 79 %

ATIS corpus has many imperatives, which Uniparse does not handle. These sentences were parsed by explicitly adding *you* to these sentences.

² The reconstructed program has supplementary code embedded in Phase 7 that allows greater utilization (relative to the original program) of information available within the original system. This permits a parallel breadth-first exploration over (a subset of all) the potential alternative first-order bracketings and second-order subcategorization possibilities, implemented via a process cloning method. This cloning strategy seems to work quite well due to the small size of the second-order engine (phase 7) in the pipeline, which is a consequence of the cascaded nature of the overall architecture of original program as well as Uniparse. For example, (from the original documentation), *While he was reading papers were flying everywhere.* fails because of the preference for the longest verb complement. The alternatives extension, using only information available from the original system, permits the intransitive possibility to be explored in parallel. In *The apparatus plans control operations.*, due to the preference for maximally extending the first-order bracketings, the entire sentence is bracketed as a single noun phrase [*the apparatus plans control operations*] and the original computation fails. But the alternatives system (again using only information left behind by each phase in the original system) permits alternative fragmentation of the over-extended noun phrase, generating both: [*the apparatus plans*] { *control* } [*operations*] + and [*The apparatus*] { *plans* } [*control operations*] + (in addition to the parallel exploration of 25 other (invalid) bracketing data interpretations and subcategorization possibilities).

The evaluations presented above are only for the purpose of giving some idea about the capabilities of Uniparse and they should be seen only in the context of (1) the criteria for evaluation and (2) the fact that information about each word was limited to only the kinds of information available to (and wired-in) Uniparse. Any large scale evaluation would not make much sense. We did however test Uniparse on a small set (about 30 sentences) of long sentences (about 20 to 30 words). The performance for NP's (computed in Phase 4) was degraded by about 5 to 7% (these percentages do not mean much as the set size of these sentences is very small). This is partly due to errors in POS disambiguation, as the tests for POS disambiguation are much fewer than those in modern rule based POS disambiguators and further Uniparse does not use any statistical information. Such information was simply not available at that time. The performance for clauses was more degraded (by about 10%, again, not too much should be read into this number), primarily because the grammar available to Uniparse is rather small as compared to those available to modern wide coverage grammars, in terms of sub-categorization frames and the variety of constructions.

5 Historical notes

The original program was implemented in the assembly language on Univac 1, a single user machine. The machine had acoustic (mercury) delay line memory of 1000 words (100 mercury channels, 10 words/channel) with input/output tape speed, 60,000 bits/sec and add/subtract speed, 1 million bits/sec. Each word was 12 characters/digits, each character/digit was 6 bits.

Lila Gleitman, Aravind Joshi, Bruria Kauffman, and Naomi Sager and a little later, Carol Chomsky were involved in the development and implementation of this program. A brief description of the program appears in (Joshi 1961) and a somewhat generalized description of the grammar appears in (Harris 1962). This program is the precursor of the string grammar program of Naomi Sager at NYU, leading up to the current parsers of Ralph Grishman (NYU) and Lynette Hirschman (formerly at UNISYS, now at Mitre Corporation). Carol Chomsky took the program to MIT and it was used in the question-answer program of Green, BASEBALL (1961). At Penn, it led to a program for transformational analysis (kernels and transformations) (1963) and, in many ways, influenced the formal work on string adjunction (1972) and later tree-adjunction (1975).

References

- Abney, Steven. 1991. Parsing by chunks. In Berwick, R. *et al.* (eds.), *Principle-based Parsing* Kluwer Academic Publishers.
- Elgot, Calvin C. and Mezzi, J.E. 1965. On relations defined by generalized finite automata. *IBM Journal of Research and Development* **9**
- Harris, Zellig S. 1962. *String Analysis of Sentence Structure* Mouton & Co. The Hague.
- Joshi, Aravind K. 1961. Computation of Syntactic Structure. *Advances in Documentation and Library Science*, vol III, part 2, Interscience Publishers.
- Karttunen, Lauri. 1996. Directed replacement. *Proceedings of the 34th Annual Meeting of ACL*, Santa Cruz, CA.
- Mohri, Mehryar. 1997. Finite state transducers in language and speech processing. To appear in *Computational Linguistics*.
- Roche, Emmanuel 1994. Two parsing methods by means of finite state transducers. *Proceedings of the 16th International Conference on Computational Linguistics (COLING'94)*, Kyoto, Japan.
- Schützenberger, Marcel P. 1977. Sur une variante des fonctions séquentielles. *Theoretical Computer Science*.
- Transformations and Discourse Analysis Project (TDAP) Reports, University of Pennsylvania, Reports #15 through #19, 1959-60. Available in the Library of the National Institute of Science and Technology (NIST) (formerly known as the National Bureau of Standards (NBS)), Bethesda, MD.