INCREMENTAL ENVIRONMENT FOR SCORED MACHINE TRANSLATION SYSTEMS

Jing-Shin Chang[*] and Keh-Yih Su[**]

[*]BTC R&D Center
2F, No. 28, R&D Road II
Science-Based Industrial Park
Hsinchu, Taiwan, R.O.C.

[**]Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan, R.O.C.

ABSTRACT

In a Machine Translation System a wide variety of data and rule bases are involved. Some of these data bases must be constructed, trained or learned from the *well-formed* output of the system. Such data bases are sensitive to the changes in the other data bases in the system, and the reconstruction of these data bases will be time-consuming if everything is reconstructed from scratch.

In this paper, we propose an *incremental* strategy to this problem. The *basic operations* in constructing an incremental interface are outlined and the associated topics are discussed. An illustrative example will be given to show the basic methodology behind the philosophy of incrementality. Some general consideration and extension about incremental approach will be discussed as well. The same strategy can be extended to handle large data base which is subject to change in a much more efficient way.

With an incremental approach, the data bases which are sensitive to changes can be reconstructed by making only *local change* to the original ones. Thus, the *time and cost* for reconstruction will be reduced significantly.

Introduction

In a Machine Translation System (MTS), a wide variety of data and rule bases ("data bases" for short) are involved. Typical data bases are the *underlying grammar* of the system (possibly augmented), *transformation rules*, *generation rules*, knowledge-based *heuristic rules* and the *lexicon information* for the lexicon items. They serve as the basis for driving the parsing mechanism and other analysis modules. All these data bases are constructed by linguists to solve most of the linguistic problems that can be generalized.

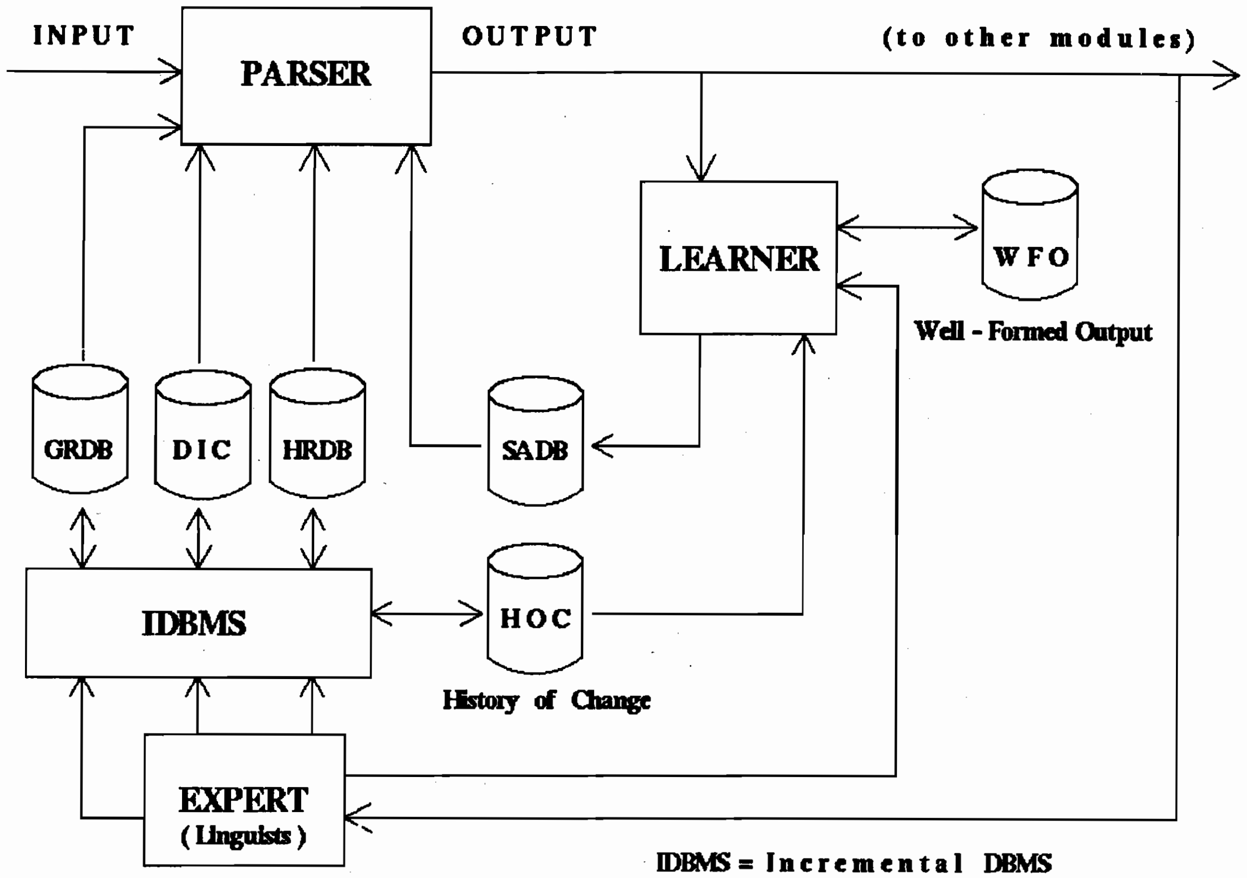
However, the linguists can not supply every *subtle* and *detail* information for the system. For instance, the detailed co-occurrence restrictions on all combinations of the syntactic categories may not be completely or explicitly described in the rules proposed by the linguists. Furthermore, in many cases the preference for different interpretations in language processing involves *uncertainty* such that even a distinguished linguists may have difficulty in providing a precise decision rule. Determination of the preference for these cases may requires *statistic* measurement instead of *subjective* judgement from linguists. Therefore, tuning the linguist-constructed data bases is usually essential to the system.

One way to tune the linguist-constructed data bases is to introduce other components capable of providing default information induced from the *previous behavior* of the system. This will create another type of data bases which are based on the *well-formed output* of the system. Data bases of this type can be used as the *feedback* to the system for *self-learning* or as the *statistic* data bases for decision making. Supplementary information for tuning linguists' rules can also be acquired from these data bases.

The **scored parsing mechanism** proposed by [Su 87], for example, uses the well-formed syntax trees observed to take part in the **score function** [Su 88]. The score function is then used for truncating potentially inappropriate parsing paths *during* scored parsing, or for computing relative preference to the ambiguous parse trees *after* parsing. Under the scoring mechanism proposed, it can be shown that the preference measurement (the score) for ambiguous parse trees, which are semantically annotated, can be divided into *syntactic* and *semantic* components in a *statistic* sense. While the **semantic score** is reflected in the lexicon information, in the augmented part of the underlying grammar, in the heuristic rules and in the associated *subjective* score assigned to them, the **syntactic score** corresponding to the syntactic component of the score function can be extracted from the well-formed syntax trees observed. A typical system involving such data base can be modeled as in Figure 1.

This model closely resembles a closed-loop control system with the **Learner** and the **Expert** modules as the feedback components. The Expert Module corresponds to the linguists (*Linguistic Experts*), who specify the grammar, linguistic knowledge, lexicon information and heuristic rules based on their expertise. Because the Expert is likely to give *inexact* and *incomplete* rules to the system, *scored parsing*, involving the management of *uncertainty*, is required for evaluating the preference to different parsing paths or the annotated trees associated with the paths.

In such a system the Learner plays an important role in tuning the expertise of the linguists. It learns the preferred syntactic construction, extracts the information required for scored parsing from the well-formed output syntax trees and constructs a special data base which is used for tuning. We shall call this data base a **Self-Adaptive Data Base (SADB).** Probability is associated with the extracted information in this data base as a measurement



DIC = Dictionary
WFO = Well-Formed Output
HOC = History Of Change

IDBMS = Incremental DBMS
GRDB = Grammar Rule Data Base
HRDB = Heuristic Rule Data Base
SADB = Self-Adaptive Data Base

Figure 1 The Data Bases in a Scored MTS with Incremental Interface

of preference. The preference not explicitly specified over different syntactic structures is then embedded in this data base in a *statistic* sense. In general a linguist would not specify such data base entries explicitly in their rules, but they are required for the system to perform **default reasoning** when expert-proposed rules are not available. They are also required when competing interpretations are encountered such that these interpretations can be distinguished only in statistic sense. Note that we have used three typical data bases to model expert-constructed data bases just for simplicity. In a practical system, the number of data bases should be much larger.

Motivation for an Incremental Interface

The special type of data base just mentioned may introduce a problem when the other related data bases are changed. This problem results from the ever-changing characteristics of the data bases in an MTS. For a Machine Translation System, since the source language to be handled is invariant as well as the parsing mechanism, the feedback components must be changed gradually so as to obtain a stable system. However, the changes in the *expert-constructed* data bases, especially that of the *underlying grammar* will change the output for the original input which is used to construct the SADB. Because the expert-constructed data bases are subject to changes from time to time, any data base, such as the SADB, that is output-dependent must be reconstructed each time the other related data bases are changed. (The data bases which cause the output for the same input to be changed between two data base versions will be called the *change-inducing data bases* hereafter.)

To reconstruct the SADB when the other data bases are changed, two approaches can be adopted. The first approach is to reconstruct the data base *from scratch*; that is, the associated input of the outdated output is *re-parsed*, the correct syntax trees are *picked out* (by linguistic experts), and the required information is *extracted* from the new well-formed output. After the tedious procedures, the new data base is *reconstructed*. This process is quite *time-consuming* because *human intervention* is required. In addition, a lot of *computing power* is wasted merely for re-computation. Moreover, the old data base becomes useless once the other related data bases are changed, no matter how trivial the change may be. Therefore this approach is impractical for a large-scaled system with large data bases.

An alternative solution to this process is to adopt an *incremental interface* between the system output and the Learner module. The Learner can be switched between two modes, the normal mode and the incremental mode. In the *normal* mode, the Learner takes the set of output which is generated for the *first* time as its input, and constructs the required data base under the directions of the linguists. In the *incremental* mode, existing well-formed output which is previously used to construct the SADB is *patched* according to the *history of change* (**HOC**) of the grammar. Under such circumstance, the reconstruction of the data base will be much more efficient because only *local* patch is performed. This methodology is termed *incremental* because all one has to do is to take the *incremental change* of the change-inducing data base into account and reshape *part of* the outdated output and data base without restarting from scratch.

Basic Operations of an Incremental Interface

To construct an incremental interface, one must define a set of *primitive operations* to represent any possible change in the change-inducing data base before the incremental interface is set up. The incremental approach then involves the execution of the following basic operations when a local change is made in the change-inducing data base.

1. Compute the *incremental change* between the two versions of the change-inducing data base.
2. Identify the sequence of the *primative operations* involved in the change.
3. Compute *the effects of the change* on the *output.*
4. Find *the range to be patched* in the original output.
5. *Patch* the original output according to the predicted effects of the change and *reconstruct* the required data base.

For example, when the underlying grammar is changed, the syntax trees previously recognized as well-formed may change their syntax structures under the new grammar. To reflect the change and patch the old output, we must first compute the difference between the two versions of the grammar. The difference may show that the change is due to *addition, deletion or renaming of symbols*. In this case, the *substitution* of a new symbol for the old one is sufficient to patch up the original output. The change may also result from *addition or* *deletion of the* **Phrase Structure (PS) Rules**. Such change may cause part of the **state space** (or **transition network**) of the parsing mechanism to be changed. Under this circumstance, the incremental change in the state space is predicted, the original output is inspected to see *whether it falls within the altered states*, the range to be patched is then searched if necessary, and the structure in the range is patched accordingly.

In general, the effect of the grammar change will only be *local*. They can be precomputed, under certain type of **closure operation**, from the change in the grammar and a set of associated productions. Therefore, the effort to reconstruct the data base is no more than patching up the portion which has been changed. Time-saving is thus expected if the incremental approach is adopted.

An Example

To show the above idea more explicitly, consider two different versions of the underlying grammars named G and G′, respectively.

G :	**G' :**	**Difference (Delta-G)**	
S -> NP VP	S -> NP VP	G :	NP -> DET n
NP -> DET n	NP -> det n		DET -> det
DET -> det	VP -> v NP		
VP -> v NP		G' :	NP -> det n

In this example, the change in the grammar is simply the reformulation of the NP (Noun Phrase) part of a sentence (S). This change can be viewed as a deletion of the two productions in G followed by an insertion of a new production in G′ which also appear in Delta-G, the difference between G and G′. (Note, however, this expression of the change in terms of the sequence of insert/delete primitives is not unique.)

The **goto-graphs** [Aho 85] for G and G′ are shown in Figure 2. The graphs consist of a number of **set of items**, each corresponding to an internal state of the parser, and a set of arcs which show the direction of transition for a specific input. The productions in a set of items and the "dots" for the productions indicate the potential position of the input pointer if the parser enter the corresponding state. In some sense, these graphs corresponds to the transition networks in **ATN** Formalism [Woods 70].

These graphs do not show the effect of the individual insertion or deletion explicitly, but the net effect of the grammar change is apparent. In this example, the incremental change in the state space is the substitution of the portion labeled as **NP sub-Grammar** in the goto-graph for G with its counterpart in G′. By examining these two graphs carefully, one can also find that the portion labeled as **NP sub-Grammar** in either graph is closely related to the **closure** of the set of items corresponding to the productions in *Delta-G*. More specifically, the portion deleted and the portion patched (after deletion) can be generated from the corresponding productions in *Delta-G*. This observation confirms our previous intuition that *local change in the underlying state space of the parser can be precomputed from the local change in the underlying grammar incrementally.*

After the incremental change in the state space is patched, we will now examine how incremental approach can be adopted to patch the old syntax tree for reuse. The syntax trees

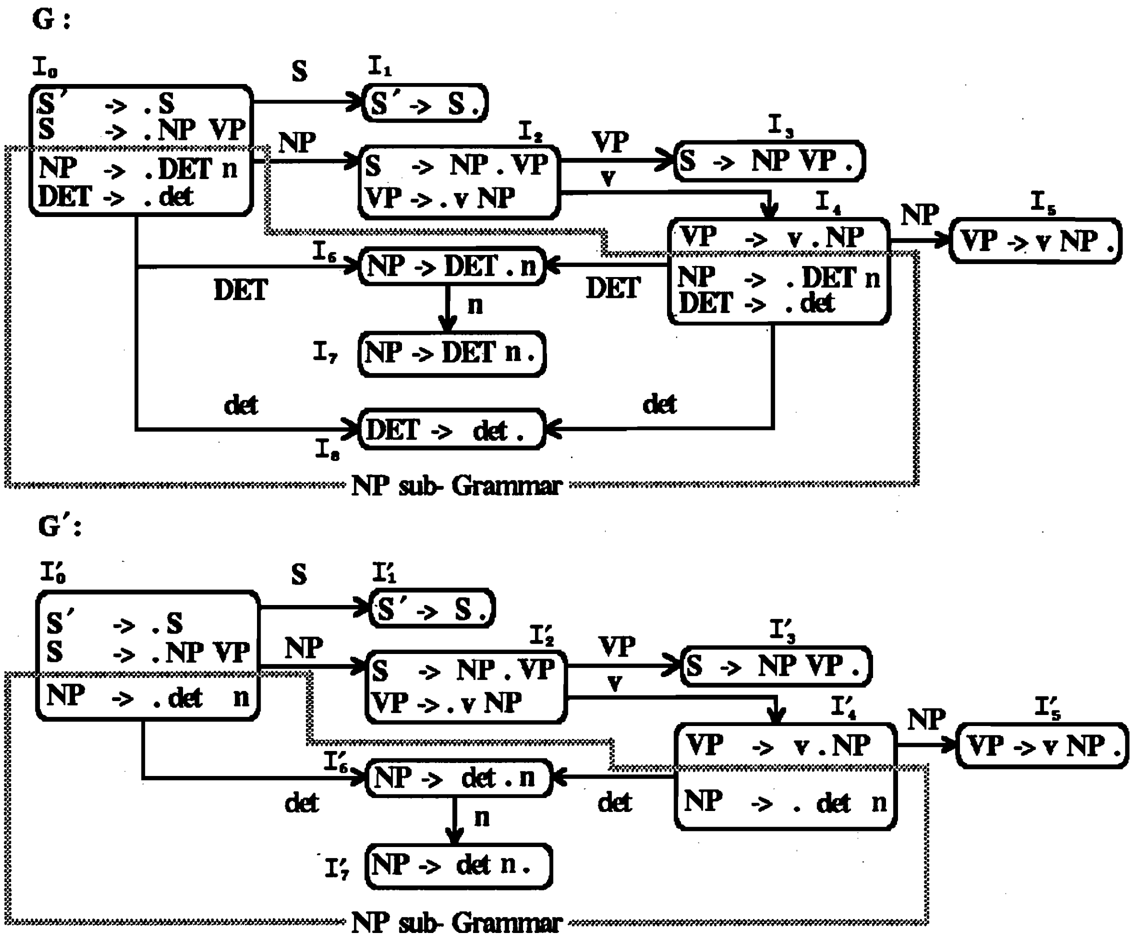


Figure 2 The GOTO- Graphs (Transition Diagrams) for G and G′

in Figure 3 show an instance of transformation from an old syntax tree to a new one, where syntax tree T is for the original grammar G and T′ for G′, respectively.

In Figure 3, the subtrees enclosed by the dashed lines are the parts to be or been patched. To see how the scopes of patching are identified, let's scan the terminal nodes from left to right. At t_0, the arrow in the figure for T is at the position preceding the nonterminals **S**, **NP**, **DET** and the terminal node **det**. If we view the arrow in T as the "dot" in the set of **LR(0) items** in the goto-graph for G, one can find without difficulty that this position corresponds to the set of items I_0. Since this set of items no longer exists in the new goto-graph, we know that this point is the *starting point* of the scope to be patched. At t_1, the state corresponding to the arrow position is also at an old state, namely I_6, not found in the new state space. Hence, it is still in the range to be patched. When the arrow is moved further to the right at t_2, the state is exactly the same as that of I_2. This set of items also appears at the goto-graph for G′ as I'_2. Hence the *end* of patching is found. It is also obvious, from the tree T, that the target to be patched is an NP. Scanning further in the same manner, one can find another scope to be patched in T, whose target is also an NP.

Keeping this information in mind, the remaining task is simply to reconstruct the NP subtrees instead of the whole sentence. This task can be done by traversing the new state space of the new grammar from an appropriate entry, namely I'_0, at the goto-graph for G′. Any parser capable of *partial parsing* should serve well for this purpose.

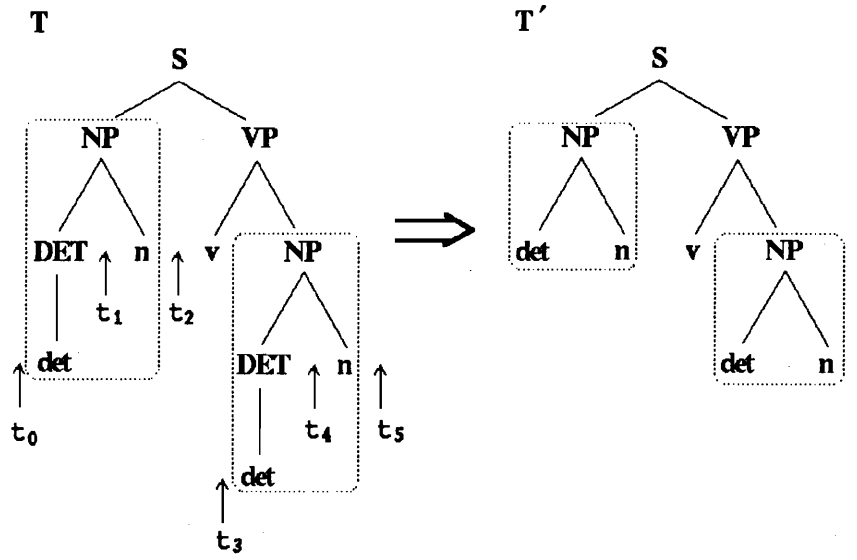


Figure 3 Syntax Trees before and after patching with incremental method

Although we have demonstrate this example in terms of some observations at the state space level, it is by no mean the sole and optimal way to address the methodology behind incremental approaches. Trade-offs for implementation will be left to a later paragraph.

It is important not to get confused with the concept of *incremental parser* [Ghezzi 80] when discussing the *incremental interface* just mentioned although both of them serve as tools to patch syntax trees. An incremental parser is used to patch a parse tree when its *input* (i.e., the *program*) is altered. The underlying *grammar* (or the *state space* of parsing) is kept unchanged. The incremental interface, on the contrary, is used to patch a well-formed syntax tree so that no re-parsing is required when the underlying *grammar* is changed. Therefore, the incremental interface is much more complicated than an incremental parser.

Basic Components in the Incremental Interface

To implement the incremental interface in the previous example, two basic components are required. The first is a **grammar editor** which records the incremental change in the symbols, grammar rules, and so forth. The second component is a **mini-parser** with *partial parsing* capability. It is used to predict the incremental change in the underlying state space and the effect of the grammar change to the parse trees. It is also used to patch up the original syntax trees if required. It is not necessary for the mini-parser to have the full power of the parser of the MTS because it only performs *local* patch. Its function can be a subset of the MTS parser, thus the name *mini-parser*. When possible, it can also be incorporated into the MTS parser and toggled in different modes.

Although we have use the dependency between the GRDB (Grammar Rule Data Base) and SADB as an example to show the idea of incrementality. This idea can be further extend to other data bases. The roles of the GRDB and SADB in the model in Figure 1 can, in fact, be played by any data bases in the model. For example, the deletion of a syntactic category from the dictionary (DIC) may cause some of the productions of the underlying grammar to become *useless*. In this case, an incremental approach can be adopted to modify the underlying grammar in a similar manner as long as it is cost-effective. The incremental change in the grammar is then handled as mentioned previously.

Hence, for a more general and integrated scheme, the grammar editor is involved in an *incremental data base management system (IDBMS)* which handles the access to the data bases in the system and maintains the history of change of the whole data base system. The components to patch up specific data bases are then incorporated into an incremental interface in a module similar to the Learner module of the more general learning system.

General Consideration

From the previous discussion we can summarize the basic philosophy behind incrementality for maintaining a large data base. The first point of greatest concern for incrementality is to make old data base entries *reusable*, and the second is to reduce the *cost of computation* for the reconstruction of the data base when local change arises.

To implement an efficient incremental interface, a few factors should be pointed out. The first important factor to consider is the selection of the set of *primitive operations* of the grammar change. It is obviously that these primitives must be able to represent or synthesize any sequence of modification to the underlying grammar. In addition, since the selection of the primitives and the representation of the change are not unique, *optimization* on the representation of the change in terms of the primitive operations would be significant when the requirement for efficiency is crucial. For example, we can *permute* the sequence of insertions and deletions in the demonstrative example without changing the result. But the *order* of the sequence of primitives may be a matter of great concern when the incremental change is patched on a *production-by-production* basis. Instead of using the deletion and insertion of a single production as the primitives, one can also take the deletion and insertion of *a group of related productions* as the primitives for incremental change. Closure operation for these productions can be performed to compute the net effect of the total change. The algorithm for patching in this way may be more complicated then the previous one, but the efficiency may be more rewarding. In either case, the IDBMS should provide a user-friendly interface for the linguists to specify the change in a linguistics-oriented manner.

Another important factor to consider is the efficiency of the mini-parser. Many strategies can be adopted to implement the mini-parser. One can *re-parse* the required subtrees for the original data base by making transition between the internal states of the mini-parser each time a syntax tree is read in. An alternative way is to *precompute* the transformation for the subtrees which are subject to change due to the local change of the grammar. The job of patching old syntax trees is then simply a task of *pattern matching*. While the first alternate requires more computation, the second one requires larger storage if the number of transformation is large. The choice is a trade-off between computing time and storage requirement. Thus a third way by mixing these two strategies is also possible. With the mixed strategy one can record the transformation when a new instance of transformation is found while re-parsing a required subtree. On the other hand, the required subtree to be patched can be replaced by the transformed pattern without re-parsing if the transformation has been recored.

Conclusion

In this paper, *incremental* strategy is proposed to resolve the problem of reconstructing the data base used in scored parsing. The algorithm for incremental patching of the data base is outlined. Fundamental components in the incremental interface are also discussed. We have pointed out a few important factors to considered about the efficiency of the incremental interface. With minor variation, the strategy can be easily extended to any self-adaptive data base which uses the output of the system as the source of the data bases. The *cost of computation* for reconstruction of the original data base can be reduced significantly with incremental approach, and the original data base will be made *reusable* without much effort for recomputation.

REFERENCES

[Aho 85] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers : Principles, Techniques and Tools,* Addison-Wesley Publishing Company, Reading, MA, 1985.

[Ghezzi 80] Carlo Ghezzi and Dino Mandrioli, "Augmenting Parsers to Support Incrementality," *J. ACM*, vol. 27, no. 3, pp. 564–579, July 1980.

[Su 87] K.-Y. Su, J.-N. Wang, W.-H. Li and J.-S. Chang,"A New Parsing Strategy in Natural Language Processing Based on the Truncation Algorithm," *Proc. of Natl. Computer Symposium (NCS) 1987*, vol. 2, pp. 580-586, National Taiwan University, Taipei, R.O.C., Dec 17-18, 1987.

[Su 88] K.-Y. Su and J.-S. Chang, "Semantic and Syntactic Aspects of Score Function," *Proc. of the 12th Int. Conf. on Comput. Linguistics, COLING-88*, vol. 2, pp. 642–644, Budapest, Hungary, 22-27 Aug. 1988.

[Woods 70] W.A. Woods, "Transition Network Grammars for Natural Language Analysis," *CACM*, vol. 13., no. 10, pp. 591-606, ACM, Oct. 1970.