A Logic-programming Approach to All-paths Parsing

Robert P. Goldman
Brown University Department of Computer Science
Providence, RI 02912, U.S.A.

Technical Report No. CS-90-17
August 1990

A Logic-Programming Approach To All-Paths Parsing

Robert P. Goldman*

Abstract

In this paper we show how all-paths parsing can be smoothly integrated into a unified semantic and pragmatic framework based on forward-chaining logic-programming. This approach combines the advantages of logic-programming parsing approaches with those of Tomita's efficient, all-paths parsing technique. The efficient representation of dependency-directed breadth-first search makes it well-suited for systems which integrate syntactic processing with semantics and pragmatics.

1 Introduction

In this paper we show how all-paths parsing can be smoothly integrated into a unified semantic and pragmatic framework based on forward-chaining logic-programming. This approach combines the advantages of logic-programming parsing approaches with those of Tomita's efficient, all-paths parsing technique. The efficient representation of dependency-directed breadth-first search makes it well-suited for systems which integrate syntactic processing with semantics and pragmatics.

Tomita's parsing technique [10] is based on LR parsing, and shares most of that technique's advantages, but can parse languages which are not LR. As with a true LR parser, there is no need to write a specific parser for a grammar: one uses a general parse table interpreter, and a parse table which is generated automatically from a user-supplied grammar. A Tomita-style parser efficiently produces multiple parses for ambiguous sentences. A great disadvantage is that it is difficult to connect this type of parser with semantic and pragmatic processing. Local ambiguity is efficiently packaged, but it is not clear how to present it to the components which do deeper processing.

On the other hand, communication is the greatest strength of logic-programming based parsing approaches. Unification provides parsers written in languages like Prolog with the ability to easily communicate between modules, and to augment their grammars with features which take into account the effects of context. Logic-programming parsers have been written using both bottom-up and top-down approaches [3, 7, 8, 9]. All of the systems known to us use a simple, chronological backtracking structure like that of the Prolog interpreter. When one wishes to find a single acceptable parse for a sentence this simple search strategy may be quite adequate. However, it is ill-suited to problems where one wishes to find all possible parses, where it takes a long time to detect errors made early in parsing, or where one can bring to bear constraints from loosely-coupled modules.

Our approach addresses these three problems. First of all, we use Tomita's approach to efficiently generate all possible parses of the target sentence. This is useful when syntax is not sufficient to distinguish a single preferred parse. If it takes a long time to detect errors made early in parsing, a search strategy which has some breadth-first aspect is

*This work has been supported in part by the National Science Foundation under grants IST 8416034 and IST 8515005 and Office of Naval Research under grant N00014-79-C-0529.

preferable (provided, as mentioned above, the cost of backtracking is high)[1]. Breadth-first search in combination with dependency information is useful when there are loosely-coupled modules. For example, if an incorrect choice has been made early in syntactic processing, it is not necessary or desirable to throw away subsequent, but unrelated syntactic, semantic or pragmatic processing.

In earlier work, we have taken a logic-programming approach to semantic and pragmatic interpretation[2, 5]. In this work, we used first-order logic to describe semantic interpretation. We developed an inference engine which takes this axiomatization and interprets it to carry out semantic interpretation. In this paper we will show that the inference engine used in our approach is capable of implementing Tomita's parsing scheme in a way that retains its efficiency, while gaining a logic-programming-style flexibility in communicating with morphological, semantic and pragmatic processing components.

The distinguishing feature of our approach is the way it treats ambiguity. Unlike earlier parsers, we combine a breadth-first search strategy with dependency information so that the consequences of incorrect decisions may be detected easily, and that all and only these consequences may be revoked. The essential contribution of this work is the encoding of Tomita's graph-structured stack. While we have adopted Tomita's extended LR parsing algorithm, recent work by Tomita [11] shows that the graph-structured stack can be used in other parsing algorithms, and our encoding should be equally adaptable.

2 Our programming language

This breadth-first approach is made possible by our adoption of a different kind of logic-programming. Our logic-programming is done in a language, Frail3, which uses DeKleer's Assumption-based Truth Maintenance System (ATMS)[4, 5]. There are two distinctive features of this language: 1) virtually all of the inference is done by forward-chaining application of modus ponens, rather than backward-chaining; 2) a single state of the database can represent many, not necessarily consistent, states of the world.

Frail operates by forward-chaining: as statements are added to its database, they fire forward-chaining rules, which add their consequents to the database. For example, the following rule:

```
(→ (word-inst ?instance sink)
    (OR (and (inst ?instance bath-furniture) (part-of-speech ?instance noun))
        (and (inst ?instance sinking) (part-of-speech ?instance verb))))
```

says that whenever an instance of the word 'sink' is asserted, we forward-chain to a statement that the word denotes an instance of either a piece of bathroom furniture, or a sinking event. For example, asserting (word-inst sink22 sink) yields

```
(OR (and (inst sink22 bath-furniture) (part-of-speech sink22 noun))
    (and (inst sink22 sinking) (part-of-speech sink22 verb)))
```

Note that a rule like this may lead to a disjunction of incompatible world-views.

The ATMS makes it possible for us to represent these disjunctions in a single database. Whenever a disjunction is asserted into the database, a special node called an *assumption* is created for each disjunct. Each statement in the database has a 'label', which indicates the set of assumptions which must be accepted in order for that statement to be believed. A network of data dependencies ensures that derived statements are given the appropriate

[1] DeKleer[4] gives a good discussion of the merits of various search strategies.

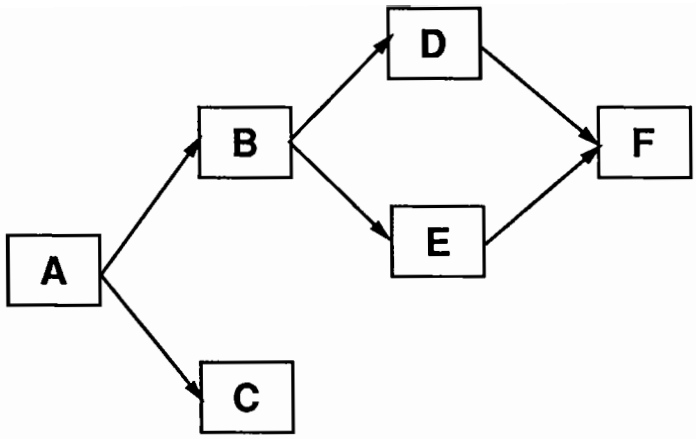

Figure 1: Pictorial representation of disjunctions

labels, and that inconsistent sets of assumptions (*nogoods*) are ignored. Here is an example of how some simple disjunctions come out.

A (→ A (OR B C))
(→ B (OR D E))
(→ D F)
(→ E F)

Formulas	Assumptions	Labels
A		(())
B	1	((1))
C	2	((2))
D	3	((1 3))
E	4	((1 4))
F		((1))

Note that the label of F is simply 1, since it follows from both alternatives from 1. Figure 1 represents this pictorially.

3 Tomita's Approach To All-Paths Parsing

In [10], Tomita presents a technique for efficiently producing all possible parses for an ambiguous input, using an LR parsing technique. A conventional LR parser has three components: a parser action table, which can be automatically constructed from a grammar, an interpreter program, and a stack, which records the state of the parser's computation. LR parsers differ only in their parser action tables; the interpreter is always the same. For reasons of space, we must assume familiarity with the LR parsing algorithm. For a detailed exposition, see [1].

Conventional LR parsers cannot handle ambiguity. Simulating non-determinism in a naive way (by creating multiple stacks to record the multiple states) costs exponential time. Tomita's approach is to have a digraph, rather than a stack, represent the state of the parse, and to fold together different parts of the parse forest.

Consider the example from Tomita's paper: "I saw the man in the park with a telescope." The (non-LR) grammar is reproduced in figure 2, and the parser action table in

1	S → NP VP
2	S → S PP
3	NP → n
4	NP → det n
5	NP → NP PP
6	PP → prep NP
7	VP → v NP

Figure 2: Grammar for Tomita's example.

	Actions					Goto's			
State	det	n	v	prep	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	acc		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	re6,sh6	re6		9		
12				re7,sh6	re7		9		

Figure 3: Parser action table for Tomita's example.

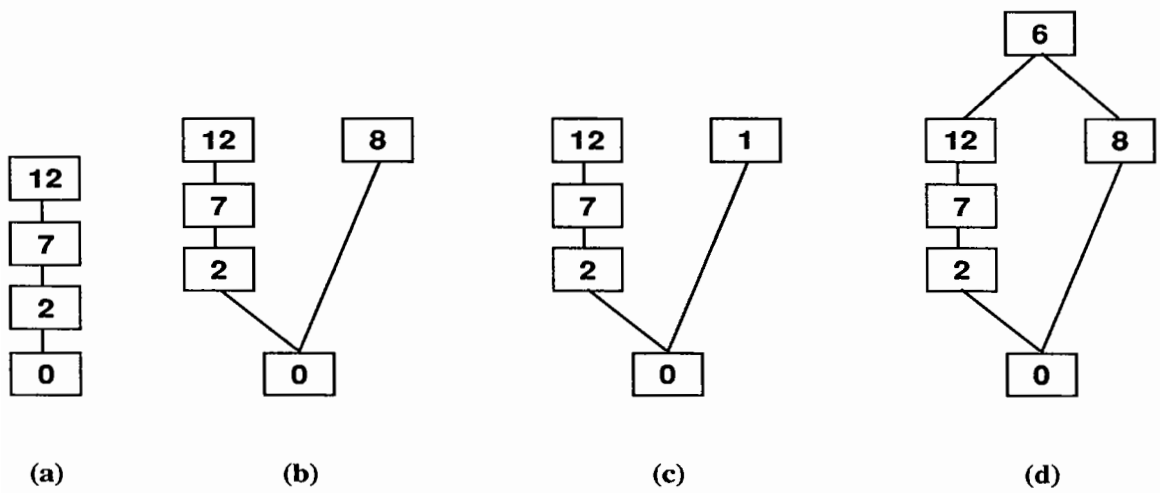

Figure 4: Stack sequence for Tomita's example.

figure 3. After reading "I saw the man," and reducing "the man" to NP, the stack is as in figure 4a. Here there are two possible actions: "reduce 7" and "shift 6". The reduction is done first, yielding figure 4b. This reduction is followed immediately by another, by rule 1, resulting in figure 4c. Now both tops of stack call for "shift 6". Only one new stack state is created, so we finish as shown in figure 4d.

4 All-paths parsing in an ATMS

In this section we will show how to implement a Tomita-style parser in our ATMS-based framework. This is done by showing how we can use our logical database to record the state of the computation; i.e., by showing how we can describe a graph-structured stack in the ATMS. The rest of the implementation follows straightforwardly: we use a parser-action table exactly like Tomita's, and the parser program is essentially the same. The only difference in the parser program is that when we have a choice of different actions to take, we create assumptions for each alternative. We cannot emphasize strongly enough that none of the following details need be understood in order to use the parser, or to construct a grammar for it.

We need to have statements in the database which will describe the stack states and top of stack pointers. The required predicates are as follows (in the following, *state* and *previous-state* are always used to denote unique names which pick out *particular* stack states):

- (stack-state *state state-num*) The stack state *state* is of the type given by *state-num*.
- (previous *state previous-state*) *previous-state* is the stack entry directly below *state*.
- (parse-tree *state parse-tree*) *parse-tree* is the parse-tree fragment associated with *state*.
- (top-of-stack *state time*) *state* is the stack entry at the top of the stack after reading the *time*-th input token.

To show how this works, we will return to the example from Tomita's paper. Having read "I saw the man," and with the stack as in figure 4a, there are two possible actions: "shift 6," and "reduce 7." So we create a disjunction of two assumptions (0 and 1), to label the two branches, (the shift and reduce actions, respectively). Doing the shift first, we assert the following statements into the database:

```
(stack-state foo23 6)          ;foo23 is the newly-created stack state
(previous foo23 foo22)         ;foo22 was the old top-of-stack
(parse-tree foo23 (prep in1))  ;in1 is the token of the word 'in'
(top-of-stack foo23 5)         ;after parsing 5 words, foo23 is the top-of-stack
```

At this moment, all of these statements will have ((0)) as their labels.

Taking the reduce path, after the two reductions we will have created a new stack state, foo27, as the other top-of-stack:

```
(stack-state foo27 1)
(previous foo27 foo24)
(parse-tree foo27 (S (NP (N I2)) (VP (V SAW3)(NP (DET A4)(N MAN5)))))
```

Each of these statements will have ((1)) as its label.

Now, when attempting to carry out a "shift 6", we find there is already a suitable state available, so rather than creating another stack-state of type 6, we re-use foo23. The following statements are shared between both possible states of the parse, so they will have the label (()):

(stack-state foo23 6)
(parse-tree foo23 (prep in1))
(top-of-stack foo23 5)

However, corresponding to the two states of the computation, there will be two pointers to the stack state below foo23:

(previous foo23 foo22) label: ((0))
(previous foo23 foo27) label: ((1))

5 Integrating different kinds of processing

We integrate semantic processing with syntax in the conventional way: by attaching production rules to the rules of the grammar. These rules will be fired when the parser executes the corresponding reduce actions. Because the parser operates within the same framework as the semantic and pragmatic processing, communication between these modules (and the lexical analyzer) is straightforward: if a constraint at any level is violated, a nogood is generated. Parser states, and states of semantic/pragmatic processing which rely on such a decision are discarded. We illustrate this principle with two simple examples:

Consider the sentence: "The sink overflowed." In this sentence, the word "sink" is ambiguous. It could be the noun referring to sinks, or the verb 'to sink.' Accordingly, when the word 'sink' is seen in the input, the lexical analyzer will assert the following two disjuncts:

Formulas	Label
(inst sink1 sinking) (part-of-speech sink1 verb)	((0))
(inst sink1 bath-furniture) (part-of-speech sink1 noun)	((1))

Now, when the parser finds that there is no action corresponding to having a verb in the input stream after a determiner, it will declare the assumption set (0) a nogood. All statements which rely on this assumption set will be ruled out, resolving the word-sense ambiguity.

It is also possible for the semantics to constrain the parse. For example, consider the sentence "Ross phoned the man with the limp."[2] The two attachment possibilities give rise to four possible semantic interpretations:

Formulas	Label
(== (accompaniment phone1) limp2)	((0 2))
(== (instrument phone1) limp2)	((0 3))
(== (attribute man3) limp2)	((1 4))
(== (accompaniment man3) limp2)	((1 5))

Note that the first two interpretations correspond to attaching the prepositional phrase "with the limp" to "phone", while the latter two correspond to attaching it to "the man". The two possible attachments are indicated by assumptions 0 and 1, respectively. Now, semantic considerations rule out the first two interpretations and the final one. Thus all environments containing assumption 0 become nogood, and this incorrect attachment is rejected on semantic grounds.

[2] I am indebted to Hirst[6] for this example.

6 Conclusions

We have shown that Tomita's all-paths parsing may be implemented in a forward-chaining logic-programming language. This implementation retains the advantages of Tomita's algorithm, and with the added advantage of integration with lexical, semantic and pragmatic processing. The parser described here has been implemented in Common Lisp on a Symbolics Lisp machine.

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] Eugene Charniak and Robert P. Goldman. A logic for semantic interpretation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pages 87–94, 1988.

[3] A. Colmerauer. Metamorphosis grammars. In G. Goos and J. Hartmanis, editors, *Natural Language Communication*, pages 133–189. Springer-Verlag, 1978.

[4] Johan deKleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[5] Robert P. Goldman and Eugene Charniak. A probabilistic ATMS for plan recognition. In *Proceedings of the Plan-recognition workshop*, 1988.

[6] Graeme Hirst. Semantic interpretation against ambiguity. Technical report, Brown University Department of Computer Science, 1983.

[7] Robert A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland, Amsterdam, 1979.

[8] Yuji Matsumoto, Hozumi Tanaka, and Masaki Kiyono. Bup: A bottom-up parsing system for natural languages. In Michel van Canegham and David H.D. Warren, editors, *Logic Programming and Its Applications*, pages 262–275. Ablex Publishing Corp., 1986.

[9] F. Pereira and D. Warren. Definite clause grammar for language analysis – a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[10] Masaru Tomita. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 756–765, 1985.

[11] Masaru Tomita. Graph-structured stack and natural language parsing. In *Proceedings of the 26th Annual Meeting of the ACL*, pages 249–257, 1988.