ソフトウェア基礎論 23－2
(1987. 12. 10)

A New Approach to Parallel Parsing for Context-Free Grammars

- An Example of Object-Oriented Concurrent Programming -

Akinori Yonezawa
Ichiro Ohsawa

Department of Information Science
Tokyo Institute of Technology
Ookayama, Meguro-Ku
Tokyo 152, Japan

Abstract

In our parsing scheme, a set of grammar rules is represented by a network of processor-like computing agents, each of which corresponds to an occurrence of a non-terminal or terminal symbol appearing in the grammar rules. Computing agents in the network work concurrently in processing partial parse trees. This scheme is shown to be fast (O(n+h) time for the first complete parse tree, where n is the length of an input sentence and h is the height of the parse tree) and useful in various modes of parsing such as on-line parsing, overlap parsing, on-line unparsing, pipe-lining to semantics processing, etc.

The code for the parser is given, which serves as an example program of the application of a programming language ABCL/1 for object-oriented concurrent programming.

1 Introduction

This paper presents a new approach to parsing for context-free grammars, which is conceptually very simple. The significance of our approach is supported by recent trends in computer-related fields. In computational linguistics, much attention has been drawn to parsing of context free grammars owing to the progress of context-free based grammatical frameworks for natural languages such as GPSG [Gazdar85], LFG [Kaplan82], FUG [Kay79]. Furthermore, many practical natural language interface systems such as [Hendrix77], [Brown82] are based on context-free (phrase structure) grammars. In computer architecture and programming, exploitation of parallelism has be actively pursued; innovative computer architectures utilizing a great number of processors [Hillis85] [Seitz85] [BBN85] [Gottlieb83] have been developed and accordingly new methodologies for concurrent programming [Yonezawa87] [Gelernter86] have been actively studied.

In our parsing scheme, a given set of context-free grammar rules is viewed as a network of terminal and non-terminal[1] symbols, and a corresponding network of processor-like computing agents (or simple processors) is constructed. The node set of the network has a direct one-to-one correspondence to the set of occurrences of symbols appearing in the grammar rules and the link topology of the network is directly

[1]In this paper, the term 'non-terminal symbol' should be interpreted in a broad sense. *Feature bundles* are considered to be non-terminal symbols.

derived from the structure of the set of grammar rules. Our parsing scheme produces all the possible parse trees for a given input string.

Since the notion of *objects* in object-oriented concurrent programming naturally fits the computing agents composing the network, this parsing scheme has been implemented in an object-oriented language for concurrent programming ABCL/1[Yonezawa86b] in a very simple and direct manner. The code for a simplified version of the parser and a sample session of the use of the parser are given in Appendix.

2 The Basic Scheme

2.1 A Symbol as a Computing Agent

Our approach is basically bottom-up. Suppose we have a context free grammar rule such as:

```
VP --> V NP                (1)
```

In bottom-up parsing, a usual interpretation of this kind of rule is:

> In a substring of an input string, if its first half portion can be reduced to a category (terminal/non-terminal symbol) V and subsequently, if its second half portion can be reduced to a category VP, then the whole substring can be reduced to a category VP.

This interpretation is implicitly based upon the following two assumptions about parsing process:

- a single computing agent (processor or process) is working on the input string, and
- non-terminal or terminal symbols such as VP, V, and NP are viewed as passive tokens or data.

Instead, we will take a radically different approach, in which

- more than one, actually, a number of computing agents are allowed to work concurrently, each performing a rather simple task,
- for each occurrence of a non-terminal or terminal symbol in grammar rules, a computing agent is given,
- such a computing agent receives data (messages), manipulates and stores data in its local memory, and also can send data (messages) to other computing agents that correspond to non-terminal or terminal symbols, and
- data to be passed around among such computing agents are partial parse trees.

Suppose that the computing agent which acts for the V symbol in Rule (1) has received a (token that represents a) partial parse tree t1. Also suppose that the computing agent which acts for the NP symbol in Rule (1) has received a partial parse tree t2. If the terminal symbol which is the right boundary of t1 is, in the original input string, adjacent to the terminal symbol which is the left boundary of t2, then t1 and t2 can be put together and they can form a larger partial parse tree which corresponds to the VP symbol in Rule (1).

For example, let us consider an input string:

I saw a girl with a telescope.

If t1 is a parse tree constructed from 'saw' and t2 is a parse tree constructed from'a girl', then the right boundary of t1 is adjacent to the left boundary of t2. But if t2 is a parse tree constructed from 'a telescope', then t1 and t2 are not adjacent and a larger parse tree cannot be constructed from them.

Now, which computing agent should check the boundary adjacency, and which one should perform the tree-constructing task? In our scheme, it is natural that the computing agent acting for the NP symbol does the boundary checking because, in many simple cases, the NP agent often receives t2 after the V agent receives t1 (due to the left-to-right nature of on-line processing). In order for the NP agent to be able to perform this task, the V agent must send t1 to the NP agent. Upon receiving t1 from the V agent, the NP agent checks the boundary adjacency between t1 and t2 if it has already received t2. If t2 has not arrived yet, the NP agent has to postpone the boundary checking until t2 arrives. If the two boundaries are not adjacent, the NP agent stores t1 in its local memory for future references. Later on when the NP agent receives subsequently arriving partial parse trees, their left boundary will be checked againt the right boundary of t1.

When the adjacency test succeeds, the NP agent concatenates t1 and t2 and sends them to the computing agent acting for the non-terminal symbol VP in Rule (1). The VP agent constructs, out of t1 and t2, a partial parse tree with the root-node tag being the non-terminal symbol 'VP.' This newly constructed partial parse tree is then distributed by the VP agent to all the computing agents each of which acts for an occurrence of symbol VP in the right-hand side of a rule. This distributed tree in turn plays a role of data (messages) to the computing agents in exactly the same way as t1 and t2 play roles of data i.e., t1 and t2 to the V and NP agents above.

This is the basic idea of our parsing scheme. It is very simple. It is the matter of course that every single computing agent acting for a non-terminal or terminal symbol can work independently, *in parallel* and *asynchronously*. Rule (1) is represented as the computing agent network illustrated in Figure 1. (This is part of a larger network.) Boxes and arrows denote computing agents and flows of trees, respectively.

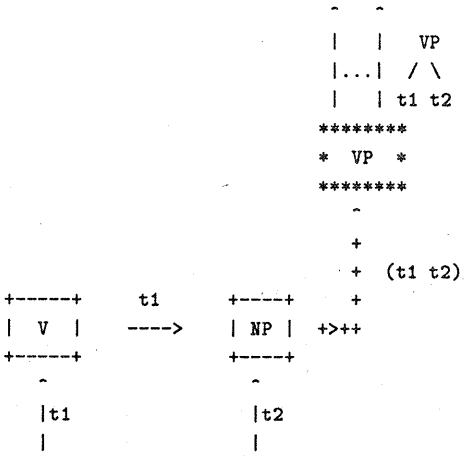

Figure 1

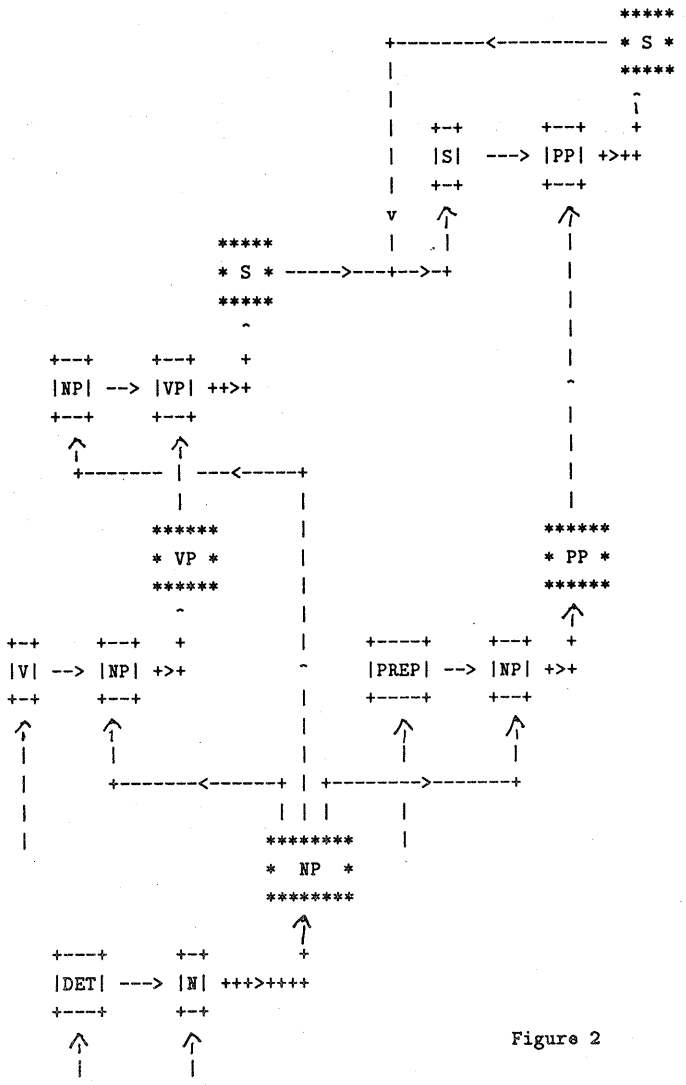

Figure 2

2.2 A Set of Rules as a Network of Computing Agents

It should be clear from the previous subsection that a set of context-free grammar rules (even a singleton grammar) is represented as a network of computing agents each of which acts for an occurrence of a non-terminal or terminal symbol in a grammar rule. More precisely, the correspondence between the set of computing agents and the set of occurrences of symbols in the set of grammar rules is one-to-one; for each occurrence of a symbol in a rule, there is one distinct computing agent. For example. the following set of rules (including Rule (1)) is represented as the network depicted in Figure 2.

```
S  --> NP VP          (2)
S  --> S PP           (3)
NP --> DET N          (4)
PP --> PREP NP        (5)
```

A box drawn with dash-lines corresponds to the computing agent acting for a symbol in the right-hand side of a grammar rule and a box drawn with star-lines corresponds to the computing agent acting for the non-terminal symbol in the left-hand side of a grammar rule. Note that the box labeled with 'NP' drawn with star-lines (at the bottom of the figure) is linked to three boxes labeled with 'NP.' This means that a partial parse tree constructed by the computing agent acting for the left-hand side symbol NP in Rule (4) is distributed to the three computing agents acting for the three occurrences of symbol NP in Rules (1), (2), and (5). Note that Rule (3) is left-recursive, which is represented as the feed-back link in Figure 2.

2.3 Three Types of Computing Agents

As the reader might have already noticed, there are three types of computing agents: Type-1 corresponds to the left-hand side symbol in a grammar rule, Type-2 corresponds to the left-corner (i.e. left-most) right-hand side symbol, and Type-3 corresponds to other right-hand side symbols. (If a grammar rule has more than two right-hand side symbols, each of the right-hand side symbols except the left-corner symbol is represented as a Type-3 agents.) For example, in Rule (1), Type-1 corresponds to VP, Type-2 to V, and Type-3 to NP.

A Type-1 computing agent A1 receives a concatenation of parse trees from the Type-3 agent acting for the right-most right-hand side symbol (NP for the case of Rule (1)) and constructs a new parse tree with its root node being the non-terminal symbol that A1 acts for and distributes it to all the Type-2 or Type-3 agents acting for the occurrences of the same non-terminal symbol (e.g., 'NP' in the above case).

A Type-2 computing agent A2 receives a partial parse tree from some computing agent that is acting for the occurrence of the same symbol as A2 acts for, and simply passes it to the computing agent acting for the symbol occurrence which is right-adjacent to the symbol occurrence that A2 is acting for. In the case of Rule (1), a Type-2 agent acting for V simply passes the received partial parse tree to the computing agent acting for NP. In the case where a grammar rule has just one right-hand side symbol as in

```
NP --> N,                    (6)
```

a Type-2 agent acting for N sends a partial parse tree to the Type-1 agent acting for NP.

A Type-3 computing agent has two kinds of sources of parse trees to receive: one from Type-1 agents and the other from the Type-2 or Type-3 agent acting for its left-adjacent symbol occurrence. In the case of Rule (1), the Type-3 agent acting for NP receives partial parse trees from Type-1 agents acting for occurrences of symbol NP in other rules and also from the Type-2 agent acting for V. Upon receiving a partial parse tree t1 from one of the sources, a Type-3 agent A3 checks to see if it has already received, from the other kind of source, a partial parse tree which clears the boundary adjacency test against t1. If such a parse tree t2 has already arrived at A3, then A3 concatenates t1 and t2 and passes them to the computing agent acting for the symbol occurrence which is right-adjacent to the symbol occurrence A3 acting for. If no such parse tree has arrived yet, A3 stores t1 in its local memory for the future use. In the case where no right-adjacent symbol exits in the grammar rule, (which means that the symbol occurrence A3 is acting for is the right-most right-hand side symbol in the grammar rule), A3 sends the concatenated trees to the Type-1 computing agent acting for the left-hand side symbol of the grammar rule.

The keen reader might have already noticed that Figures 1 and 2 follow the convention in which boxes drawn by star-lines are Type-1 agents and ones drawn by dashed-lines are Type-2 or Type-3 agents.

2.4 Terminal Symbols as Computing Agents

It should be noted that we do not make any distinction between non-terminal symbols and terminal symbols. In fact, this uniform treatment contributes to the conceptual simplicity of our parsing scheme. We do not have to make a special treatment for grammar rules such as:

```
NP --> NP and NP             (7)
```

where a lower case symbol 'and' is a terminal symbol. The uniformity implies that a word of a natural language, say 'fly' in English, which has more than one grammatical category should be described as follow:

```
V --> fly          (8)
N --> fly          (9)
```

where Rules (8) and (9) indicate that a word 'fly' can be a *verb* or *noun*. The two rules are represented by two Type-1 agents acting for V and N, and two Type-2 agents acting for the two occurrences of 'fly' in Rules (8) and (9). Thus, in our parsing scheme, the grammatical categories of each word in the whole vocabulary in use are described by grammar rules with a single right-hand side symbol. This means that conceptually, one or more computing agents exist for each word. (Those who might worry about the number of computing agents acting for words should read Subsection 4.2.)

2.5 Input to the Network

In our parsing scheme, a given set of grammar rules is compiled as a network of computing agents in the manner described above. Then how an input string is fed into the network of computing agents? We assume that an input string is a sequence of words, namely, terminal symbols.

In feeding an input string into the network, two things has to be taken into account. One is: for each word in an input string, appropriate computing agents, to which the word should be sent, must be found. Of course, such computing agents are ones that act for the occurrences of the word in the grammar rules. Notice that there can be more than one such computing agent for each word, due to multiplicity of grammatical category and the multiple occurrences of the same symbol in the right-hand side of grammar rules. Since the set of appropriate computing agents can be known in compiling a given set of grammar rules, such information should be kept in a special computing agent which does the managerial work for the network. Let us call it the *manager agent*. The manager agent receives an input string and sends (or distributes) each word in the input string in the on-line manner.

The other thing needed to be considered in the input feeding is: the information about the order of words appearing in an input string must be provided to computing agents in the network in an appropriate way. This is because Type-3 computing agents need such information to perform the boundary adjacency test. For this, each word to be sent (or distributed) to computing agents in the network should be accompanied with its positional information in the input string. Suppose an input string is `I saw a girl with a telescope`. Then a word `girl` should be sent with the pair of its starting position number and its ending position number. The actual form of data (message) for the word `girl` may look like `(3 4 girl)`. See Figure 3. This data form convention can be adopted in dealing with more general parse trees. (In fact, a single word (terminal symbol) is also the simplest case of parse trees.)

2.6 How Symbols Flow

To get a more concrete feeling of how symbols are processed in the network, let us look at the flows of words `a` and `girl` in the initial phase. Assuming that the following rules are compiled into the network,

```
DET --> a          (10)
N   --> girl       (11)
```

the manager agent sends `(2 3 a)` and `(3 4 girl)` to the Type-2 computing agent acting for `a` in Rule (10) and the Type-2 computing agent acting for `girl` in Rule (11), respectively. They are in turn sent to a Type-1 agent Det1 acting for `DET` in Rule (10) and a Type-1 agent N1 acting for N in Rule (11), respectively. These Type-1 agents construct a parse tree with its root node label being `DET` or `N`. Then the parse tree constructed by Det1 is sent to a Type-2 agent Det2 acting for `DET` in Rule (4). Similarly, the parse tree constructed by N1 is sent to a Type-*3* agent N2 acting for `N` in Rule (4). In both cases, the positional information is accompanied. That is, the data forms actually to be sent are `(2 3 (DET a))` and `(3 4 (N girl))`.

Agent Det2 simply passes the parse tree to agent N2. N2 performs the boundary adjacency test between `(2 3 (DET a))` and `(3 4 (N girl))` and finds the test to be ok. Since the test is ok, N2 concatenates the two data forms, constructing a new single data form:

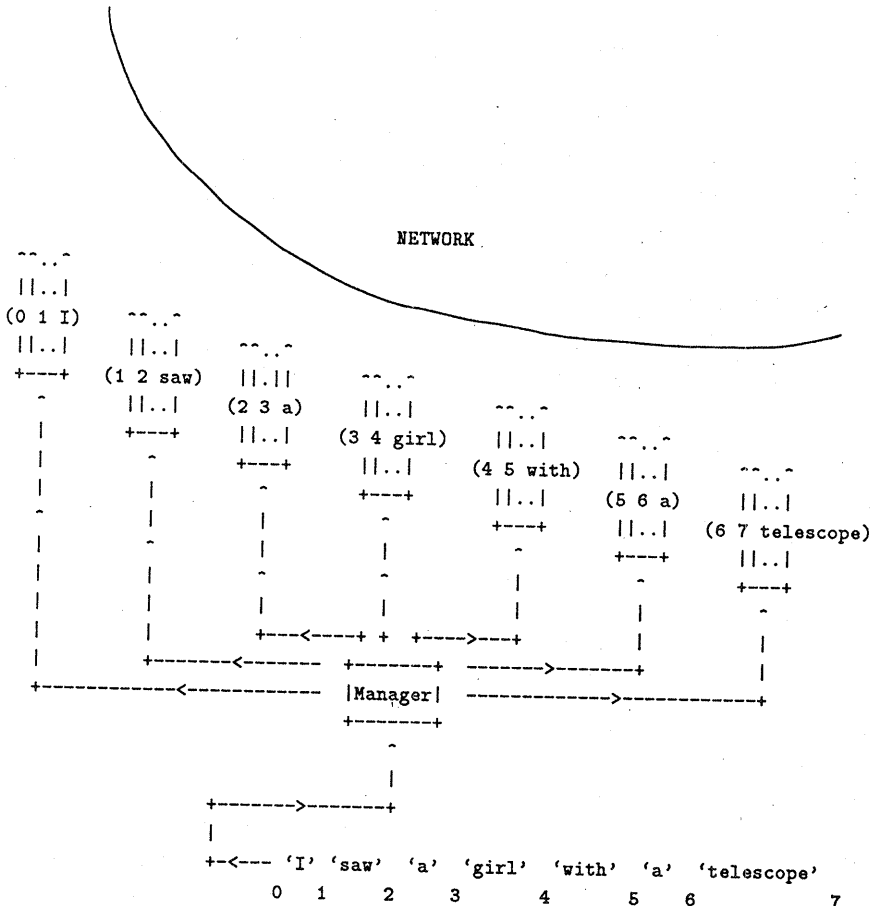
NETWORK
(0 1 I)
(1 2 saw)
(2 3 a)
(3 4 girl)
(4 5 with)
(5 6 a)
(6 7 telescope)
Manager
'I' 'saw' 'a' 'girl' 'with' 'a' 'telescope'
0 1 2 3 4 5 6 7

Figure 3

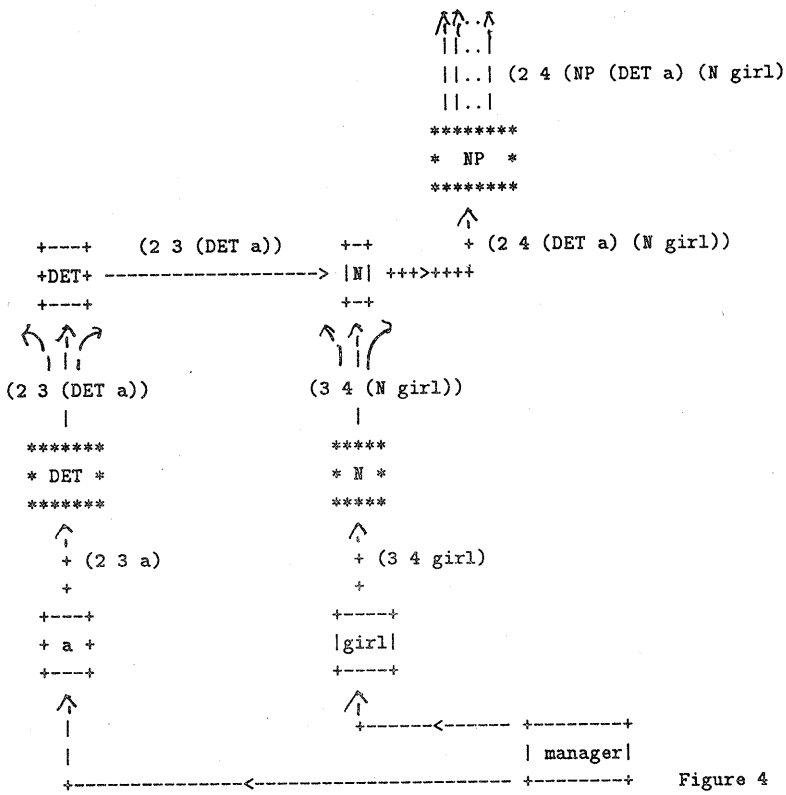

Figure 4

(2 4 (DET a) (N girl))

This new data form is then sent to the Type-1 agent acting for NP in Rule (4). This agent constructs a data form of the parse tree for NP, which looks like:

(2 4 (NP (DET a) (N girl)))

This data form will be distributed among Type-2 and Type-3 computing agents acting for symbol NP in the network. (See Figure 4.)

At the end of this section, it should be reminded that actions taken by computing agents such as Det1, Det2, N1, and N2 are performed all in parallel. Also note that such computing agents keep being activated as long as data continue to arrive.

3 Applications

3.1 On-Line Parsing and Overlap Parsing

In starting parsing process, our scheme does not require the network of computing agents to be fed any token that indicates the end of an input string. That is, an input string can be processed one by one from the beginning in the on-line fashion. Even if feeding an input string to the network is suspended in the middle of the string, partial parse trees can be constructed based on the part of the input string that has

been fed so far, and the feeding of the rest of the input string can be resumed at any moment. Thus, our parsing scheme is quite useful in real-time applications such as interpreting telephony (simultaneous interpretation). (Notice that our scheme does not require that an input string is fed in the left-to-right manner; words in the input string can be fed in any order as long as the positional information of each word in the input string is accompanied. See Subsection 2.5.)

Our parsing scheme has no difficulty even when more than one input string is fed to the network simultaneously as long as different input strings are fed separately. The separation can be easily made by attaching the same tag (or token) to each word in the same input string. Such a tag is copied and inherited to partial parse trees which are constructed from the same input string. When a Type-3 computing agent tests the boundary adjacency between two partial parse trees, the sameness of the tags of the two partial parse trees are checked additionally. The capability of handling the multiple input strings is useful in processing the overlapping utterances by more than two persons engaging in conversation.

This way of handling the multiplicity of input strings is similar to the idea of *color tokens* used in data-flow computer architectures.

3.2 Unparsing

Suppose the user is typing an input string on a keyboard and he hits the 'backspace' key to correct previously typed words. In the case where these incorrect words have already been fed to the network, our parsing scheme is able to unparse the incorrect portion of the input string and allows the user to retype it. Furthermore, the user can continue to type the rest of the originally intended input string.

This unparsing capability is realized by the use of *anti-messages*. The anti-message [Jefferson85] of a message M sent to a computing agent A is a message that will be sent to A in order to cancel the effects caused by M. The actual task of cancelling the effects is carried out by A. (Thus A has been programmed beforehand so that it can accept cancelling messages and perform the task.) If necessary, A must in turn send anti-messages to cancel the effects caused by the messages A itself has already sent. In implementing the unparsing capability, the express-mode message passing in ABCL/1 [Yonezawa86b] is useful, which is a kind of *interrupt*-like high priority message passing.

3.3 Pipe-Lining to Semantic Processing Agents

Our parsing scheme produces all the possible (partial) parse trees for a given input string. In fact, if each Type-1 computing agent in the network stores, in its local memory, all the parse trees it constructs, all the components of the triangle matrix used in CKY parsing method (i.e., all the possible parse trees) are in fact stored among the Type-1 agents in the network in a distributed manner. If the semantic processing is required, these partial or complete parse trees can be sent to some computing agents which do semantics processing.

Actually, parse trees can be sent to semantic processing agents in a *pipe-lining* manner. Suppose a Type-1 computing agent Np1 is acting for an occurrence of a non-terminal symbol NP. Instead of letting Np1 distribute the parse trees it constructs to Type-2 or Type-3 agents acting for occurrences of the symbol NP, we can let Np1 to send the parse trees to the semantics processing agent which check the semantic validity of the parse trees in the pipe-lining manner. After filtered by the semantic processing agents, only semantically valid parse trees (possibly with semantics information being attached) are passed to Type-2 or Type-3 computing agents acting for NP. See Figure 5.

These semantic filtering agents can be inserted at any links between Type-1 agents and Type-2 or Type-3 agents. The complete separation of the semantic processing phase from the syntactic processing phase in usual natural language processing systems corresponds to the placing semantic processing agents only after the Type-1 computing agents that act for the non-terminal symbol S that stands for *correct* sentences.

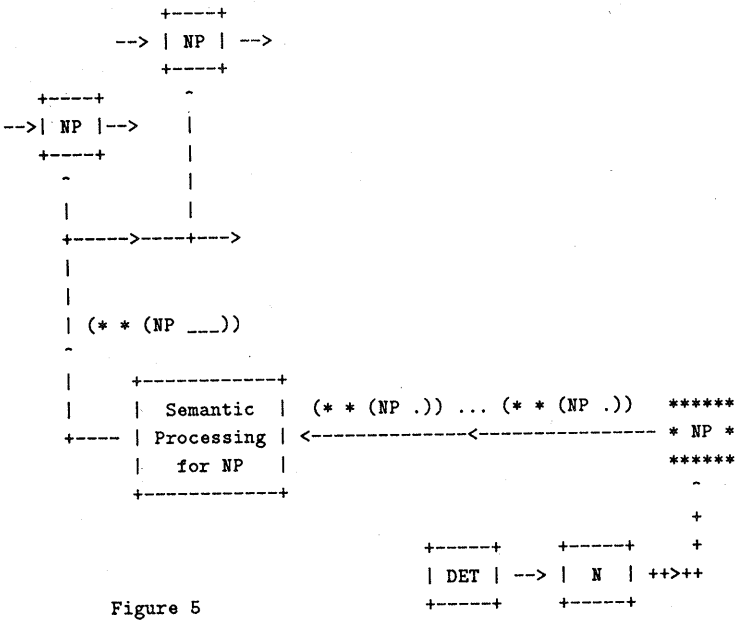

Figure 5

4 Analysis

4.1 Complexity Analysis and Degree of Parallelism

It is hard to analyze the complexity of our parsing scheme in an exact manner because much parallelism is involved in various ways. It can, however, be shown that the first complete parse tree is produced from the network in O(n+h) time, measuring from the beginning of feeding an input string, where n is the length of the input string and h is the height of the parse tree (*not the height of the network*)[2]. In this analysis we assumed that both processing of a partial parse tree by a single computing agent and a message transmission between two computing agents take a single unit time.

Since all the possible complete parse trees for a given input string are produced from the network in the pipe-lining manner, the second and subsequent complete trees are expected to be produced in a short interval, possibly a single unit time interval. But more precise estimation requires more careful analysis including simulation, which is found in [Yonezawa87a].

Also it should be pointed out that the degree of parallelism in processing an input string is O(n*h).

4.2 The Number of Computing Agents

As is obvious from the construction of the network, the number of computing agents is exactly the same as that of the nodes of the network. Since the node set of the network has one-to-one correspondence to the set of symbol occurrences in a given set of grammar rules, the number of computing agents can be very large if the grammar is complex. But not all these agents can be active simultaneously. The number of all the agents that can be active in processing an input string is small compared to that of the computing agents consisting of the network.

Since the main task of the Type-1 agent acting for the left-hand side symbol of a grammar is just to distribute a constructed parse tree, this task can be performed by the Type-3 agent which acts for the

[2]Usually, the order of the height of a parse tree is the same as the left boundary of the length of an input string.

rightmost right-hand side symbol of the grammar rule. Thus the Type-1 agent can be eliminated. This reduces the number of computing agents.

Of course, there are number of other ways to reduce the number of computing agents at the sacrifice of processing speed and the conceptual clarity of the parsing scheme. We, however, believe that maturity of the technology for exploitation of parallelism will dispel the apprehensions regarding the number of computing agents.

4.3 Generality of the Parsing Scheme

Our parsing scheme can handle the most general class of context free grammars except cyclic grammars. If a set of grammar rules has circularity[3], infinite message passing may take place in the network. To detect or avoid such infinite message passing, a special provision must be made. But fortunately such a provision can be done at the time of compiling the set of grammar rules into the corresponding network of computing agents.

As suggested in Subsection 2.2 and Figure 2, left-recursive grammar rules can be handled without any modification to the grammar rules. From the nature of bottom-up parsing, our parsing scheme cannot handle an ϵ-rule (a rule that produces a null string). But as is well known[Hopcroft79], all the ϵ-rules can be eliminated from a given set of grammar rules by transforming the set of rules without changing the generative power of the original set of rules.[4]

Acknowledgments

The authors would like to thank S. Sato (Kyoto University), and E. Shibayama (Tokyo Institute of Technology) for their comments on an early version of this paper.

References

[Agha85] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, 1986.

[BBN85] Bolt Beranek and Newman Inc: Development of a Butterfly Multiprocessor Test Bed, Rep. 5872, Quarterly Tech. Report No.1, 1985.

[Brown82] J. S. Brown, R. R. Burton, and J. deKleer, *Pedagogical Natural Language and Knowledge Engineering Techniques in SOPHIE I, II, and III*, Academic Press, London, 1982.

[Gazdar85] G. Gazdar, E. Klein, G. K. Pullum and I. A. Sag, *Generalized Phrase Structure Grammar*, Basic Blackwell Publisher, 1985.

[Gelernter86] D. Gelernter, Domesticating Parallelism, *IEEE Computers*, No. 8, 1986.

[Gottlieb83] A. Gottlieb et al.: The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. Computers*, C-32, No.2, 1983.

[Hendrix77] G. G. Hendrix: The LIFER Manual: A Guide to Building Practical Natural Language Interfaces, *Technical Report TR-138*, SRI International, 1977.

[Hillis85] D. Hillis, *The Connection Machine*, The MIT Press, 1985.

[Hopcroft79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[3] A simple example of circular rules is: `A --> B, B --> A, B --> C.`

[4] The original language is assumed to contain no null symbol.

[Ikeda87] Y. Ikeda, J. Tsujii and M, Nagao: Control of Parsing in KGW + P, Reprint of EICSJ(DenshiJouhouTsushin Gakkai) Working Group Meeting on Language Processing and Communication, June 1987.

[Jefferson85] D. R. Jefferson: Virtual Time, *ACM Trans. Prog. Lang. Syst.*, Vol.7, No.3, 1985.

[Kaplan82] R. M. Kaplan and J. Bresnan: Lexical-Functional Grammar: A Formal System for Grammar Representation, in *The Mental Representation of Grammatical Relations* J. Bresnan (ed.), The MIT Press, 1982.

[Kay79] M. Kay: Functional Grammar, *Fifth Annual Meeting of the Berkeley Linguistic Society*, 1979.

[Matsumoto86] Y. Matsumoto: A Parallel Parsing System for Natural Language Analysis, *Lecture Notes in Computer Science*, No. 225, pp. 396-409.

[Ohsawa86] I. Ohsawa and A. Yonezawa: A Parallel Parser in ABCL/1, Reprint of IPSJ(Jouhoushori Gakkai) Working Group Meeting on Software Foundation 86-SF-19, 1986. (in Japanese)

[Seitz85] C. L. Seitz: The Cosmic Cube, *CACM*, Vol.28, No. 1, 1985.

[Shibayama86] E. Shibayama and A. Yonezawa: ABCL/1 User's Guide (Draft), Dept. of Information Science, Tokyo Inst. of Technology, 1986.

[Tomita86] M. Tomita, *Efficient Parsing for Natural Language*, Kluwer Academic Publisher, 1986.

[Yonezawa86b] A. Yonezawa, J.-P. Briot and E. Shibayama: Object-Oriented Concurrent Programming in ABCL/1, *Proc. 1st ACM Symposium on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 258-268, 1983.

[Yonezawa87] A. Yonezawa and M. Tokoro (Eds), *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[Yonezawa87a] A. Yonezawa: A Complexity Analysis for a Parallel Parsing Scheme, *in preparation*, 1987.