TR-I-0229

# Unification-Based Parsing on Increasing Levels of Parallelism
# 並列効果の高い単一化解析法

P. Neuhaus　O. Furuse　H. Iida
ノイハウス・ペーター　古瀬　蔵　飯田　仁

1991.9

As effectively programmable parallel architectures become available their usage in natural language processing increases. But an often disregarded problem is the discrepancy between the number of processors required by so-called *massively-parallel* algorithms and the number of processors provided by the parallel machine actually at hand.

A parallel parsing algorithm on the basis of the well known CYK algorithm has been published. We present an efficient, further parallelized version for JPSG-like unification-based grammars and show the effectiveness of restricting parallelization with regard to the size of the parallel machine used.

効果的にプログラム可能な並列アーキテクチャが利用できるようになるにつれ、自然言語処理においても並列計算機が使われるようになってきた。しかし、超並列アルゴリズムなどで仮定される理論上のプロセッサの数と、実際に並列計算機で使用可能なプロセッサの数の違いの問題は議論されないことが多かった。本論文では、ＪＰＳＧのようなユニフィケーションに基づく文法を使って、ＣＹＫの並列化アルゴリズムを改良した効率的なパーサを提案し、与えられた並列計算機の並列度に関し、効果的に動作することを示す。

ATR 自動翻訳電話研究所
ATR Interpreting Telephony Research Laboratories

# Contents

# 1 Introduction

As effectively programmable parallel architectures become available their usage in natural language processing increases. Algorithms for parallel analysis, i.e. parsing and unification, have been published. For example, [Mat89] presents a parallel parser based on logic programming, [Lan90] shows the utilization of systolic computations for parallel parsing, [Fuj90] investigates parallel unification, to name only a few.

Some of the suggested algorithms are *massively* parallel (for instance [Lan90]). But an often disregarded problem is the discrepancy between the number of processors required by these algorithms and the number of processors provided by the parallel machine actually at hand. Usually this problem is handled by virtual processors and/or a scheduling scheme. However, this obviously introduces additional overhead, that in the worst case, negates the benefits of parallelization. So-called massively-parallel parsers are an interesting research subject, but most available parallel architectures will not allow for *massive* parallelism.

In this paper a parallel parsing algorithm for a JPSG-style unification-based grammar is presented. It is based on the CYK (Cocke-Younger-Kasami, for example [You67]) parsing[1] algorithm. The algorithm is parallelized on increasing levels in terms of the number of required processors. At the first level, as proposed by [Bar90], a linear number of processors (in the length of the input string) is used. We show that on further levels more and more parallelism can be achieved and that this is necessary for an efficient parser for natural language processing.

In section 3 the original CYK parsing algorithm and its first level parallelization are explained. After a brief discussion of possible parallelizations in section 4, section 5 presents our further parallelized versions that will fit *medium*-sized parallel machines. In section 6 implementational details are explained. Finally, section 7 gives a discussion of results that show how the choice of an appropriate parallelization scheme is crucial for efficient run-time results.

# 2 Unification-based Grammar

A unification-based grammar has two components: a (usually[2]) context-free grammar backbone and some feature structure formalism attached to it. For a basic introduction to unification-based grammars the reader is referred to [Shi86]. We use a grammar following the notion of JPSG as presented in [Gun87].

JPSG uses just one context free rule, *Mother* → *Daughter Head*, to account for the structure of Japanese sentences. This assumption of binary structures in JPSG suggests the use of a CYK type of parser, because it requires the context-free grammar to be in Chomsky Normal Form[3], i.e. it must be binary (cf. section 3).

We do not restrict the grammar used (see [Kog88]) to contain only one context free rule as was suggested in [Gun87]. Actually the grammar should try to restrict the context-free syntax as far as possible because checking context free rules is less expensive than doing unifications. [Tom91] reports that in certain parsers over 90% of the time is consumed by unification.

[1] A remark concerning the term "parser": we will use it both in the sense of parsing a context free grammar and in the sense of analyzing natural language which itself consists of parsing and unification. Its use should be clear from the context.

[2] [Mat89] covers non-context-free grammars, too.

[3] Actually it is easily possible to allow unary rules which may sometimes be convenient.

There are two main strategies combining a parser and a unification algorithm. One is to produce all parse trees of the input string first, and then to do all unifications in a second phase. The other way is to do the unification after each reduction step of the parser. Since many syntactically possible structures are semantically ill-formed, the latter strategy will rule out these structures at an early point in time. This kind of parser is called an *integrated parser* and will be our choice.

Figure 1 shows all necessary parts of a complete parser. The following sections will describe the unifier and parser only.

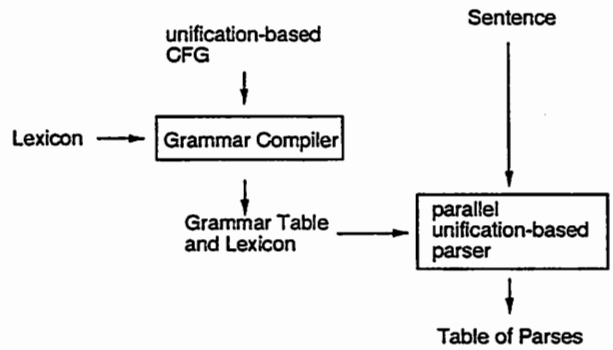


Figure 1: Unification based parser architecture

# 3 Parallel Parsing

Before introducing the first level of parallelization we will explain the original sequential CYK algorithm as presented in [You67].

## 3.1 The CYK Parser for Context Free Languages

If a grammar contains only productions of the form $A \rightarrow B\ C$ or $A \rightarrow t$ for some *non-terminals* $A$, $B$ *and* $C$ and some *terminal* $t$, it is said to be in *Chomsky normal form.* The standard CYK algorithm determines for all sub-strings of the input string their possible (sub-)tree structures. This is done by building a *table*, here $M$. The first row contains the pre-terminals (the $A$ of "$A \rightarrow t$") as *table entries* and is indexed $M_{i,i}$, where $i = 0, \ldots, n-1$ and $n$ is the length of the input string.

The next row's entries – indexed as $M_{i,i+1}$, for $i = 0, \ldots, n-2$ – are computed by *combining* two adjacent entries of the previous line, i.e. $M_{i,i}$ and $M_{i+1,i+1}$. Combining two entries means, that the non-terminals in the two sets are *joined* to pairs and for each pair the grammar is checked (by a table lookup) if it can be reduced by a production. If this is the case this production's left side is added to the set $M_{i,i+1}$. Thus this new entry describes two tokens of the input string. For example, in figure 2 the verb "ita" (past tense of "to be") and the noun "toki" ("time") form a noun phrase.

For the next rows, each entry describes substrings of the input string. For the computation of entry $M_{i,j}$ (describing all possible structures of the substring from tokens $i$ to $j$ of the input string), for all $k = i \ldots (j-1)$, entries $M_{i,k}$ and $M_{k+1,j}$ are combined. For example, figure 2 shows

the computation of $M_{2,4}$ at the third row. Entries $M_{2,2}$ and $M_{3,4}$ are combined by joining V and PP, for which no grammar rule exists. Entries $M_{2,3}$ and $M_{4,4}$ are combined by joining NP and P, yielding PP. For entries in the next row three combinations will take place, and so on.

Eventually table entry $M_{0,n-1}$ will contain the first non-terminals of all possible parses. If it contains the root symbol, obviously the string has been accepted. Actually at this point no tree is directly available. How to get the parse tree (in our implementation) will be described in section 6.

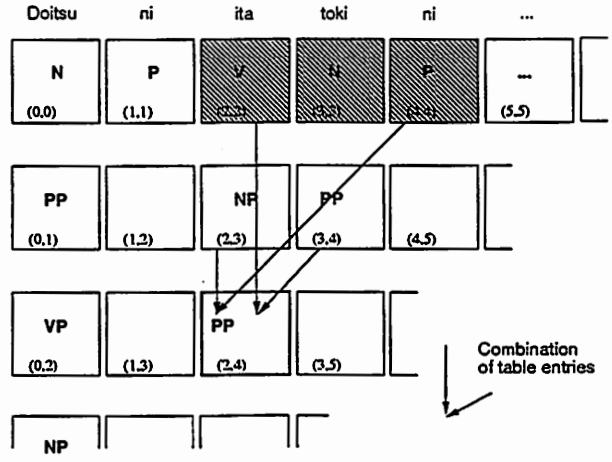


Figure 2: The combination scheme for entries in table $M$, here entry $M_{2,4}$

Since each table entry can contain at most all non-terminals of the grammar the combination of two table entries is of constant complexity in the length of the input string. That, the overall complexity is $O(n^3)$.

### 3.2 A Parallel CYK Parser

As shown by [Bar90] the above described algorithm's time complexity can be decreased to $O(n^2)$ by utilization of a number of processes linear in the number of input tokens, the time-processor product remaining the same.

As can be seen from the description of the sequential CYK algorithm the computation of a row entry depends only on previous rows. For example in figure 2, while $M_{2,4}$ is computed by one process, other processes can compute entries $M_{0,2}$, $M_{1,3}$, $M_{3,5}$, and so on. That means that all entries of a row can be computed in parallel. We need only ensure that the computation does not start before the required entries of previous rows are computed.

Let $P_i$ for $i = 0 \ldots n-1$ be the processes then $P_i$ computes entries $M_{i,i} \ldots M_{i,n-1}$. $Synchronize(P_i, P_{i+1})$ makes sure that process $P_i$ does not proceed to compute entry $M_{i,j}$ until process $P_{i+1}$ finished its computation of entry $M_{i+1,j}$. We get the following algorithm:

```
begin P{i}
  for j=i to (n-1)
  do
    synchronize(P{i},P{i+1})
```

```
            compute-entry(M{i,j})
        end P
```

This algorithm has been proposed in the field of programming language parsing. In [Bar90] it is argued that – with respect to existing parallel architectures – a parallel parsing algorithm that's processor usage is of linear complexity (in the length of the input string) is much more realistic than an algorithm with quadratic complexity is. Though input strings in natural language analysis are much shorter, a similar argument applies to natural language parsing as well.

### 3.3 Theoretical Complexity vs. Actual Usage of Processors

Because of the much shorter strings in natural language analysis the complexity in terms of the O-calculus is not so important. The O-calculus is only valid for asymtotic considerations. For us the ratio of the number of processes to the number of (hardware) processors is much more important. If it becomes extremely big or small then the parsing is far from being efficient.

The parallel CYK algorithm presented above starts out with $n$ processes doing one combination of entries. With every new row one process terminates but each remaining process has to combine one more pair of entries, thus the quadratic time complexity. What should concern us is that once there are less processes than actual processors we waste computational power. The effect of this waste can be seen in the results presented below.

## 4 Possible Parallelization of Unification-based Parsers

The basic analysis of the parallelization of unification-based parsers (cf. [Kat90]) shows that there are three sources of parallelism exploitable by a parsing algorithm:

1. context-free grammar
   (a) independent sub-trees
   (b) structural ambiguities
2. disjunctions of feature structures
3. recursive unification of complex feature structures

A parse tree usually contains *non-overlapping subtrees* that can be computed in parallel. Moreover, if there is a *structural ambiguity* in a sentence, two or more parse trees will be created. The related unifications can be done simultaneously. Figure 3 shows an example. It also illustrates item (1a) because the upper parse contains two **P** sub-trees that can be computed in parallel.

*Disjunction of feature structures* means that one terminal or non-terminal has more than one feature structure. For example, a verb may have two different feature structures, for example because of a transitive and an intransitive meaning (see figure 4). Thus there are two possibilities to form a verb-phrase by adding an inflection. The corresponding unifications can be done in parallel.

Two *complex feature structures* are said to unify if the values (possibly again complex structures) of corresponding attributes unify. Complex feature structures therefore imply recursion. Its

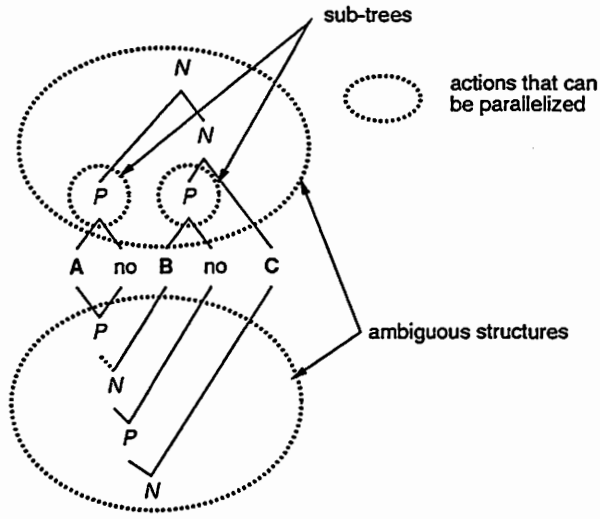


Figure 3: Parallelism in the context-free grammar

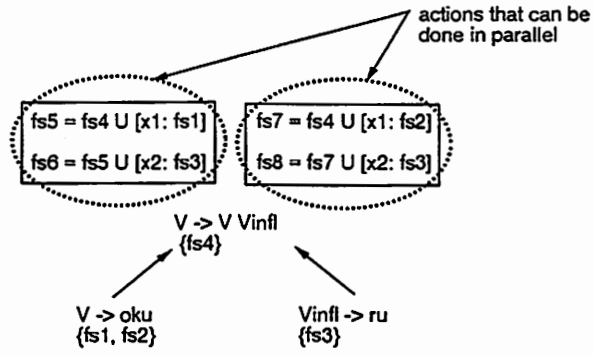


Figure 4: Parallel unification of feature structures

parallelization would lead to massive parallelism[4]. We did not exploit this source of parallelism (item (3) above) because it counteracts achieving a parsing algorithm efficient for *medium*-sized parallel architectures. For a discussion of this kind of massive parallelization and its results the reader is referred to [Fuj90].

# 5 Our Further Parallelized Parsing Algorithms

We are presenting one parsing algorithm on several levels of parallelization. Thus the following subsections show algorithmic realizations of the discussion above.

## 5.1 Combining Table Entries is Independent

Let us examine the example of computing a row entry from above. To compute entry $M_{2,4}$ it is necessary to combine entries $M_{2,2}$ with $M_{3,4}$ and $M_{2,3}$ with $M_{4,4}$. These two combinations can be done in parallel, let us say by a *worker*-process.

This will lead to the usage of $O(n^2)$ processors[5]. The synchronization overhead increases because now all *worker*-processes have to be synchronized because an entry is not entirely computed until all related "workers" are finished. To prevent them from overwriting each other's results, a *lock* must be used also (*with-lock* acquires the lock, executes the body and releases the lock). Thus we get the following algorithm:

```
begin P{i,k}  \\ worker for k-th combination
  for j=i to (n-1)
  do
     for l=i to (j-1)
     do
        synchronize(P{i,k},P{i+1,l})
     combine(M{i,k},M{(k+1),j})
     with-lock
       write(M{i,j})
end P
```

This approach is somewhat naive because in an actual parse there are often empty entries. A *worker* combining empty entries just consumes time for synchronization. After looking at another source of parallelism in the next section we will consider further the combination of table entries.

## 5.2 Combining Non-Terminals is Independent

For the combination of two entries each non-terminal in the first entry is combined with each non-terminal in the second entry. While the check of the grammar table is very fast and can be done sequentially the unification of two items is slow. So the unifications should be done in parallel after the sequential syntax check has sorted out all impossible combinations with respect

[4]To avoid massive parallelism here a *worker/agenda* mechanism could be helpful. Though the BEHOLDER package will supply such a mechanism it has not been implementation yet.

[5]More exactly the number of processors will be $\lfloor n/2 \rfloor * \lceil n/2 \rceil$

to the context free grammar. The synchronization mechanism used here is that of a *fork*, i.e. the entry-combination routine *spawns* unification processes and waits for them to be terminated.

This further parallelization covers item 2 of section 4. If a non-terminal has disjunctive feature structures there are multiple copies of it in the table entry, each with a different feature structure. Parallelizing the combination of all non-terminals of two table entries obviously leads to a parallel unification of the disjunction of feature structures. The resulting algorithm is shown in the next section after one more level of parallelization has been added.

### 5.3 Reconsider Combination of Entries

The fork mechanism described in section 5.2 can be expanded to the entire computation of an entry. That is not only the joining of non-terminals but also the combination of table entries is executed in the *fork*. Thus it covers not only all unifications of one combination of a pair of entries but all necessary combinations of previous entries. That yields the following algorithm:

```
begin P{i}
  for j=i to (n-1)
  do
     synchronize(P{i},P{i+1})
     compute-entry(M{i,j})
end P

begin compute-entry(M{i,j})
  for k=i to (j-1)
  fork
       combine(M{i,k},M{(k+1),j})
end compute-entry
```

```
begin combine(M{i,k},M{l,j})
  for all pairs in M{i,k} X M{l,j}
  do
     if reducible
     then mark pair
  for all marked pairs
  fork
       unify
end combine
```

## 6 Implementation

We implemented several parallel versions of a parser along the above described levels of parallelization on a Sequent Symmetry S81 with 12 tightly-coupled processors. The coding was done in CLiP (cf. [CLiP]), a parallel LISP based on Allegro CL. To deal with specific problems of CLiP the BEHOLDER package proved to be very useful.

Under CLiP only 11 processors are available and another processor has been spent for monitoring, such that we could test the different parsers with 1 to 10 processors. On these, so called, *Light Weight Processes* were scheduled. A sequential reference version also has been implemented.

In our algorithm on a tightly-coupled architecture the parsing table $M$ is held in common memory. In contrast to the original algorithm presented in section 3.1, we store pointers to the constituents of each non-terminal. Thus, an entry can contain multiple copies of one non-terminal – each with different constituents[6]

[6]This has a heavy impact on the time complexity of the first level parallelization. Since now an entry can contain more non-terminals than there are in the grammar the combination of two entries is not of linear complexity anymore. Theoretically the complexity became even exponential with respect to ambiguities in the grammar and the input string.

The stated synchronization condition is implemented by blocking and unblocking processes. Before a process computes an entry it reads a channel (here a mail-box) from its right neighbour (see figure 5). If it is empty (indicating that required data is still being computed by its right neighbour) the process blocks until a message arrives in its mail-box (thereby unblocking the process). If the mail-box is not empty all required data is available and the process does not have to block. Each process tells its left neighbour through a sync mail-box that it has completed the computation of the previous entry.

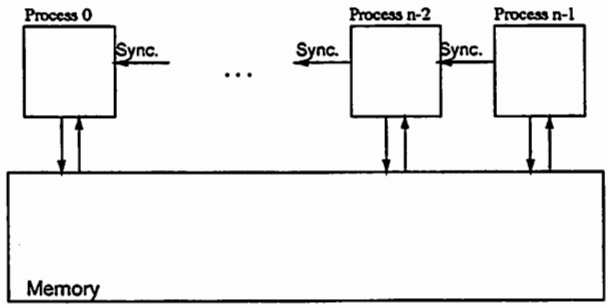


Figure 5: Synchronization scheme of parallel CYK

We are using the quasi-destructive graph unification algorithm as presented in [Tom91]. It solves the efficiency problems in case of unification failures[7] most elegantly, while it is still easy to be altered to run concurrently, see appendix A.

# 7 Evaluation

In the following the implementations (see appendix B for a more detailed description) of the presented algorithms will be referred to as:

- `:sequ` – the sequential CYK algorithm (section 3.1)
- `:lin` – the first level parallelization presented in [Bar90] (section 3.2)
- `:sqr` – the naive parallelization using $O(n^2)$ processors (section 5.1)
- `:fork` – the further parallelized parsing algorithm (sections 5.2 and 5.3)

Figure 6 shows the run-time of parsing three sentences by *:fork* compared to the sequential version of the algorithm (the dotted lines). The sentences are, A: *onamae to gozyuusyo wo onegai shi masu* ("Would you please give me your name and address?" 819 top-level unifications), B: *soredewa kochira kara sochira ni tourokuyoushi wo ookuri itashi masu* ("Then, I'll send you a registration form." 321 top-level unifications) and C: *wakari mashita* ("I see." 34 top-level unifications).

The synchronization overhead is compensated for when at least two or three processors are used. The speed up was 1:3 for the 10-processor run vs. the 1-processor run in all tests. It was at least 1:2 for the 10-processor run vs. the sequential reference version.

[7] If a unification algorithm creates copies of feature structures but eventually the unification fails, the copying is a waste of time and memory resources.

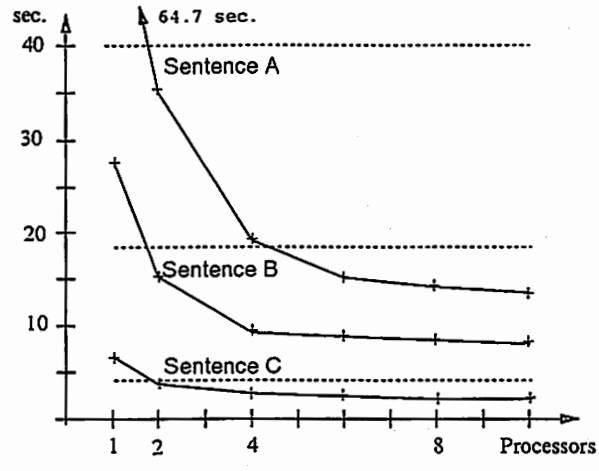


Figure 6: Parsing times for three sentences by *:fork*

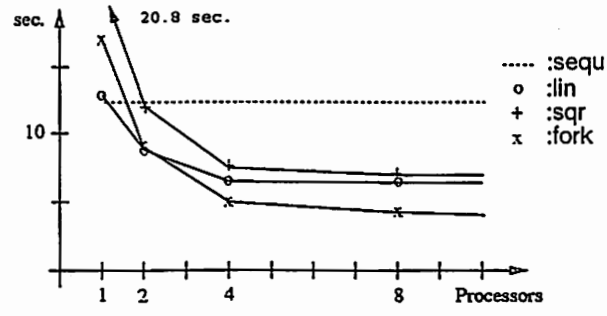


Figure 7: Comparison of all four versions

Figure 7 compares all versions of the presented algorithm for sentence D: *wakara nai ten ga gozaimashi tara watakushidomo ni itsu demo okiki kudasai.* ("Please feel free to ask if there's anything you don't understand." 141 top-level unifications).

The *:lin* version suffers from its not using all processors. This waste shows especially at its bad scaling: The run-time for more than four processors was more or less constant in all tests. The reason is, that *:lin* actually does not use most of the processors much. Still, this version is easy to implement and maybe interesting for very small parallel machines.

Finally, the *:sqr* version clearly is not usable at all. The reason is the large amount of sychronization necessary with this algorithm. Even, its implementation is more complicated than that of the other versions.

The run-time of *:fork* shows, that this is a better way to do things. Though it required more synchronization than :lin, it was faster with three or more processors. Also, it showed a better scaling.

# 8 Conclusion

We introduced the CYK parsing algorithm and its parallelization following [Bar90]. On this basis an efficient parallel unification-based parser has been presented. It used further parallelism for better utilization of the parallel machine and incorporated a unification algorithm. The strength of the presented parallel parsing algorithm lies in the combination of a fast unification algorithm with a parser that is parallelized with respect to the underlying architecture. The reduced parallelism, in contrast to massively parallel algorithms, resulted in good run-times on a medium-sized parallel machine. Even with as few as four processors, good speed-ups could be achieved.

# A The Quasi-Destructive Graph Unification

As explained in [Tom91] the quasi-destractive graph unification algorithm avoids unnecessary copying by the usage of a global clock. That is, all unification information that has to be added to a feature structure is marked by a time-stamp. Only if the whole unification suceeds the structure and its added information is copied. The time is stepped after each top-level unification, thereby devalualting the added information.

Let us now consider the example from above where a verb has different feature structures. When the unification algorithm runs simultaneously on the feature structure of a verb inflection the unification information will become defective. What is needed are independent places for these different data.

Since there is a maximal number of processors running the unification algorithms it would be sufficient to supply a special slot in the data structure for each processor. But in CLiP it is not possible to identify the actual processor, though process names can be determined. So we chose a hash-table organisation where the hash-key is the process name. This is less efficient in terms of memory usage but fast and stable.

For illustration the data structure is shown in figure 8. The first three slots are used to represent the original node information. In the hash-tables additional information and time-stamps are saved.

| type | :atomic, :complex or :leave |
|---|---|
| arc-list | list of graph node's edges |
| forward | forwarded node |
| comp-arc-list | hash-table of edges added in unification |
| comp-forward | hash-table of nodes forwarded in unification |
| comp-arc-mark | hash-table of time-stamps for comp-arc-list |
| comp-forward-mark | hash-table of time-stamps for comp-forward |

Figure 8: Adapted node structure for concurrent unification algorithm

# B Files and Usage

The system can be found in directory `/usr1/peter/IMP` on the Sequent at ATR Interpreting Telephony Research Laboratories. The parser and unifier are in files `grammar`, `pcyk`, `unify` and `interface`. In `grammar` the data structures and routines to read grammar and lexicon are defined. In `pcyk` the actual parser is contained, plus some functions for printing tree structures. Finally `unify` and `interface` provide the adapted version of the quasi-destructive graph unification algorithm.

To load the system the file `local_init.lisp` is used. There the garbage colection behaviour is changed and all necessary files are loaded[8]. Especially the example grammar and lexicon, `tom.gra` and `tom.lex` respectively, are loaded[9]. These may be changed by the user to own files. Please observe that the extensions *gra* and *lex* are obligatory.

[8] We assume that the BEHOLDER package is loaded. For details see the `/usr1/peter/.clinit.cl` file.

[9] The test grammar is essentially Kogure's as in [Kog88].

The main function is (parse sentence &optional algorithm) that parses a *sentence* using *algorithm*. It follows a list of available algorithms with a brief explanation:

- :sequ is the sequential reference version. It is started as a LWP because the Initial-LWP is slow - no parallel computing!!
- :lin is the 1st level parallelization, number of processes is linear in the length of the input
- :sqr is the *naive* further parallelization using a quadratic number of processes
- :fork1 similar to :lin version but after joining non-terminals unifications are done simitaniously, controlled by a FORK mechanism
- :fork, dito. but combining table entries AND joining non-terminals are paralleized in a FORK
- :sequfork, like :fork but on basis of the sequential algorithm
- :sqrfork, like :fork but on the basis of the quadratic algorithm
- :uni, like :fork but allowing unitary rules

The documentation string of *parse* gives further information how to control the output format of parsing results.

# C Garbage Collection

The Allegro CLiP uses a so-called "generation-scavenging garbage collection" system. for a detailed introduction see [CLiP]. We were able to adjust the garbage collection behaviour such that it could cope with the fast increasing number of graph structures, that were deleted just after the run of the parser.

A main problem was that when ever the garbage collector required more memory and the OS would not be able to supply a piece of memory subsequent to the prior memory, than an entire new work space is requested. The point is, that the now unused old work space was not freed causing the system to run in severe memory problems.

# D Example Screens

This appendix shows some examples of the system implemented on a Sequent Symmetry S81.

```
<Initial lwp> :ld local_init.lisp

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Experimental parallel parser on the basis of a CYK parser.
(parse sent[1-16] :algorithm) to use various algorithms.
Don't forget to call (memory) after first newspace expansion.

<Initial lwp> (parse sent9)
```

```
Parsing sentence: (SOREDEHA KOCHIRA KARA SOCHIRA NI TOUROKUYOUSHI WO OOKURI ITASHI MASU),
by algorithm :FORK
Using 1 processor

scavenging... expanding and moving new space-done ut: 6%, copy new: 3088048 +  old: 0 = 3088048

cpu time (non-gc) 10867 msec user, 1099 msec system
cpu time (gc)     14600 msec user, 1684 msec system
cpu time (total)  25467 msec user, 2783 msec system

NIL
-------------------------------------------
Number of parses: 5

NIL
<Initial lwp> (memory) ;; new space has been expanded
T
<Initial lwp> (set-numprocs 4)
4
<Initial lwp> (parse sent9)

Parsing sentence: (SOREDEHA KOCHIRA KARA SOCHIRA NI TOUROKUYOUSHI WO OOKURI ITASHI MASU),
by algorithm :FORK
Using 4 processors

cpu time (non-gc) 4650 msec user, 783 msec system
cpu time (gc)     0 msec user, 0 msec system
cpu time (total)  4650 msec user, 783 msec system
real time  5640 msec

NIL
-------------------------------------------
Number of parses: 5

NIL
<Initial lwp> (set-numprocs 10)
10
<Initial lwp> (parse sent9)

Parsing sentence: (SOREDEHA KOCHIRA KARA SOCHIRA NI TOUROKUYOUSHI WO OOKURI ITASHI MASU),
by algorithm :FORK
Using 10 processors

cpu time (non-gc) 3950 msec user, 867 msec system
cpu time (gc)     0 msec user, 0 msec system
cpu time (total)  3950 msec user, 867 msec system
real time  4830 msec

NIL
-------------------------------------------
Number of parses: 5

NIL
<Initial lwp> (setq *print-parse* :CFG) ;; give the context-free structure
```

```
:CFG
<Initial lwp> (parse sent11)

Parsing sentence: (OOSAKASHIKITAKUCYAYAMACHIROKUNONIZYUUSAN * SUZUKIMAYUMI DESU),
by algorithm :FORK
Using 10 processors

cpu time (non-gc) 933 msec user, 0 msec system
cpu time (gc)     0 msec user, 0 msec system
cpu time (total)  933 msec user, 0 msec system
real time  1080 msec

NIL
----------------------------------------
Number of parses: 2
Parse: ----------------------------------

 X01[[WH  X02[]]
     [SUBCAT  X03[[REST  X04 END]
                 [FIRST  X05[[SUBCAT  X06 END]
                             [SEM  X07[]]
                             [HEAD  X08[[POS  X09 P]
                                        [GRF  X10 SUBJ]
                                        [FORM  X11 GA]]]]
     [SLASH  X12[]]
     [SEM  X13[[RELN  X14 DA-IDENTICAL]
               [OBJE  X07]
               [IDEN  X15[[ARG-X1  X16[]]
                          [ARG-X2  X17[[PARM  X18[]]
                                       [RESTR  X19[[IDEN  X20 SUZUKIMAYUMI]
                                                   [OBJE  X18]
                                                   [RELN  X21 NAMED]]]]
                          [RELN  X22 *-COORDINATE]]]
     [PRAG  X23[[SPEAKER  X24[]]
                [RESTR  X25[[REST  X26[]]
                            [FIRST  X27[[REST  X28 END]
                                        [FIRST  X29[[RELN  X30 POLITE]
                                                    [RECP  X07]
                                                    [AGEN  X24]]]]
                [HEARER  X07]]
     [HEAD  X31[[CFORM  X32 SENF]
                [CTYPE  X33 DESU]
                [POS  X34 V]
                [MODL  X35[[COPL  X36 +]]]
-V
----N
-------P
----------N
----------P
-------N
----V

Parse: ----------------------------------

 X01[[WH  X02[]]
```

```
     [SUBCAT  X03 END]
     [SLASH  X04[]]
     [SEM  X05[[ARG-X1  X06[]]
               [ARG-X2  X07[[PARM  X08[]]
                            [RESTR  X09[[IDEN  X10 SUZUKIMAYUMI]
                                        [OBJE  X08]
                                        [RELN  X11 NAMED]]]
               [RELN  X12 *-COORDINATE]]
     [PRAG  X13[[SPEAKER  X14[]]
               [RESTR  X15[[REST  X16[]]
                            [FIRST  X17[[REST  X18 END]
                                        [FIRST  X19[[RELN  X20 POLITE]
                                                    [RECP  X21[]]
                                                    [AGEN  X14]]]]]
               [HEARER  X21]]
     [HEAD  X22[[CFORM  X23 SENF]
               [CTYPE  X24 DESU]
               [POS  X25 V]
               [MODL  X26[[COPL  X27 +]]]
-V
----N
-------P
----------N
----------P
-------N
----V

T
<Initial lwp> (setq *print-parse* :FS) ;; give feature structure for each node
:FS
<Initial lwp> ;; no example to save space!
```

## References


[Bar90] D. T. Barnard, D. B. Skillicorn: *Parallel Parsing: A Status Report (chp. 4)*, Queen's University, 1990

[CLiP] Franz Inc.: *Allegro CLiP Manual*, 1989

[Fuj90] T. Fujioka, H. Tomabechi, O. Furuse, H. Iida: *Parallelization Technique for Quasi-Destructive Graph Unification Algorithm*, reprint of WGNL 80-7, IPSJ, 1990

[Gun87] T. Gunji: *Japanese Phrase Structure Grammar*, D. Reidel Publishing, 1987

[Kat90] S. Kato: *Performance Evaluation of Parallel Algorithms for Unification-Based Parsers*, reprint of WGNL 77-2, IPSJ, 1990

[Kog88] K. Yoshimoto, K. Kogure: *Japanese Sentence Analysis by means of Phrase Structure Grammar*, ATR Technical Report TR-I-0049, 1988

[Lan90] L. Langlois: *Parallel Parsing via the backward trace of systolic computations*, Proceedings of the Workshop of Parallel Computation, Kingston, 1990

[Mat89] Y. Matsumoto: *Natural Language Parsing Systems based on Logic Programming*, Thesis at Kyoto University, 1989

[Shi86] S. M. Shieber: *An Introduction to Unification-based Approaches to Grammar*, CSLI Lecture Notes No. 4, 1986

[Tom91] H. Tomabechi: *Quasi-Destructive Graph Unification*, Proceedings of ACL91

[You67] D. H. Younger: *Recognition and Parsing of Context-Free Languages in Time $n^3$*, Information and Control Vol. 10, p. 189–208, 1967