Translation from Transactions of IPSJ

An Efficient Sentence Analysis Method for General Phrase Structure Grammars

TATSUYA HAYASHI*

This paper describes an efficient sentence analysis method for logic grammars that are based on Chomsky's type-0 and type-1 phrase structure grammars. It is an expansion of the YAPXR system, which accepts Restricted Context-Sensitive Grammars (RCSGs). This approach, like the YAPXR, is therefore based on a breadth-first top-down method, and is capable of effectively carrying out sentence analysis by using the extended LR method, which contains the concepts of kernel or pseudo-kernel positions. Expressions based on context-free grammars are not particularly suited to languages such as Japanese, which has the characteristics of free order, abbreviation, and non-crossing dependency relationships found among "Bunsetsu" phrases in sentences. Gapping grammars, on the other hand, are also capable of handling languages with free word order, but have rather poor performance.

The proposed method is more appropriate for such languages, since it provides the power of phrase structure grammars with improved efficiency, using the concepts of pseudo-kernel or, when possible, kernel positions.

1. Introduction

The author has introduced the YAPXR Sentence Analysis System for Restricted Context-Sensitive Grammars (RCSGs) on previous occasions [1-5]. RCSGs are based on context-free grammars, but also accept context-sensitive rules if parsing efficiency is not affected.

RCSGs follow the framework of context-free grammars.

RCSGs are not particularly suited to sentence structures above the phrase level in languages such as Japanese that have free order and abbreviation.

Gapping grammars are capable of concisely expressing a language with free word order [6-8].

However, the performance of gapping grammars may be poor because gaps match arbitrary-length, partially derived strings containing zeros, and are produced by the generation-and-testing method.

Japanese is said to have free word order, but the freedom is not unconditional. The dependency relation of a phrase applies only to the phrase to the right. It never applies to the phrase on the left, and the dependency relations do not cross each other.

Japanese language can be described plainly by using general phrase structure grammars, which are logic grammars based on Chomsky's type-0 or type-1 phrase structure grammars [9]. This description relies on the introduction of nonterminal symbols representing dependency relations.

We regard these dependency relations as syntactic events. This makes it important to develop an efficient parsing technique for phrase structure grammars.

A practical parsing system based on the breadth-first top-down method for phrase structure grammars can be created by expanding the YAPXR system [10, 11].

This paper describes the process of creating such a system and consists of:

Section 1: Introduction,
Section 2: Term definitions,
Section 3: Grammar description format,
Section 4: Implementation, and
Section 5: Operational example.

2. Term Definitions

This section defines the key terms used in this paper, apart from general terms and notations used in formal language theory.

A different ID number should precede each syntactic element on the right-hand side of the following rule:

$$\alpha \rightarrow \beta, \ (\alpha \in V^+, \ \beta \in V^*).$$

These numbers are called position IDs. The position IDs immediately before and after each element are called its left and right positions. A position ID should also be given to the ε rule. Rule 0 is an exception, however, since "b" and "e" are used there as position IDs, as in $So \rightarrow bSe \dashv$.

2.1 Leftmost Derivation Position

Assume that the right-hand element of a certain rule is $A(\in V)$ and the left position is n.

This is a translation of the paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 12 (1990), pp. 1718-1726.

*Fujitsu Laboratories Ltd.

$So \overset{*}{\Longrightarrow} \alpha A\ \delta \overset{*}{\Longrightarrow} \alpha\beta\gamma \Rightarrow \alpha\eta\gamma$, $(\alpha, \gamma, \delta, \eta \in V^*, \beta \in V^+)$

In the example above, if η starts from m, m is called the leftmost derivation position of n, represented as $m \in LD(n)$.

The ε rule, however, does not apply in $\alpha A\delta \overset{*}{\Longrightarrow} \alpha\beta\gamma$.

2.2 Position Set

Assume that element $A(\in V)$ is not at the left corner of the right-hand side of a rule and that the left position is n. The set represented as $\{n\} \cup LD(n)$ is the position set of A. The position set is only permitted as an exception for the left-corner element S of Rule 0.

2.3 Kernel Position

Assume that n and x are elements in the position set. If $x \in LD(n)$, n is treated as either the kernel position of the set or the kernel position corresponding to x.

We can easily deduce that x is to the left of the left-corner element on the right-hand side of the rule. It therefore cannot be a kernel position, except for "b" mentioned above. The kernel position can also be regarded as the position set ID.

2.4 Parsing Path

For an input sentence, $a_1, a_2 \cdots a_n$, assume a partial input string, $a_1 \cdots a_i (1 \leqq i \leqq n)$. Now assume that one of the leftmost derivations contains the partial input string:

$$So \overset{*}{\Longrightarrow} a_1 \cdots a_i \alpha, (\alpha \in V^*).$$

A parsing path consists of a kernel position string, $bn_1 n_2 \cdots n_i (1 \leqq i)$, starting with b. The path represents the rules and their application order, starting from rule 0, thus showing how the partial input string $(a_1 \cdots a_i)$ was obtained. It also represents the positional range of each rule that the partial input string seems to correspond to. In other words, the parsing path is a reduced expression of the leftmost derivation. The corresponding leftmost derivation can be obtained from each parsing path instead. Strictly speaking, several leftmost derivations may temporarily correspond to a single parsing path. However, there is no need to distinguish them.

2.5 Internal Parameter

A parameter can be given to element A on the left- or right-hand side of a rule by enclosing it in parentheses, as in A(IX). This is called an internal parameter.

A parameter is used to characterize the partial input string that corresponds to A with some sense.

A parameter value is obtained directly or indirectly from a word dictionary or set directly in the rule. In the latter case, the value is specified by adding a parenthesis after the parameter name.

2.6 Basic Constraint

The basic constraint is a logical expression or prolog predicate related to an internal parameter.

This constraint can be described at an arbitrary position on the right-hand side of the rule.

3. Grammar Description Format

The grammar used here is a phrase structure grammar that contains all the functions used in the conventional YAPXR.

The basic format is as follows:

LHS→RHS
  LHS: LHE ···
  LHE: syntctic element [(internal parameter, ···)]
    [(/external parameter, ···/)]
  RHS: ε or RHE ···
  RHE: syntactic element [(internal parameter, ···)]
    [(/external parameter, ···/)]
    [slash category]
    [{context dependent constraint}]
    [{basic constraint}]

The external parameter, context dependent constraint, and slash category are not explained here because they are not relevant to the subject of this paper. (Interested readers should consult [2, 4] and [5]).

4. Implementation

As long as the description format matches that of RCSGs, the conventional technique applies. This section explains the implementation method for general phrase structure grammar rules.

RCSGs enable all the kernel positions corresponding to the left-corner element on the right-hand side of the rule to be easily calculated when a grammar is given. However, this is not always true for phrase structure grammars. This paper at first assumes that all the kernel positions can be calculated.

First assume the following grammar rule:

$$P_1 P_2 \cdots P_m \rightarrow {}^{n1}Q_1\ {}^{n2}Q_2 \cdots {}^{nn}Q_n,$$
$$(P_i, Q_j \in V, nk\text{: positional ID})$$

For simplicity, parameters and constraints are ignored here. Assume that the predicates corresponding to Pi and Qj are pi and qj.

The parameters for YAPXR are as follows:

n: First element of the parsing stack
A_i: Remaining parsing stack (input)
A_0, A_{01}: Parsing stack (output, differential list)
O_i: Abbreviation stack (input)
O_0, O_{01}: Abbreviation stack (output, differential list)
L_i: Parameter stack (input)
L_0, L_{01}: Parameter stack (output, differential list)
T_i: Parse tree stack (input)
T_0, T_{01}: Parse tree stack (output, differential list)

The abbreviation stack is not related to this subject, so it is ignored here. Since the parsing tree for a phrase structure grammar becomes complicated, the dependency relation ($W1$, $W2$) should be placed on the parse tree stack. ($W1$, $W2$) means that phrase $W1$ depends on phrase $W2$.

4.1 Left-Corner Type

The Horn clause corresponding to syntactic element $Q1$ is as follows:

$q_1(11, A_i, [[n2, 11|A_i]|A_0], A_0, L_i,$
$[[L_i]|L_0], L_0, T_i, [[T_i]|T_0], T_0).$
$q_1(12, A_i, [[n2, 12|A_i]|A_0], A_0, L_i,$
$[[L_i]|L_0], L_0, T_i, [[T_i]|T_0], T_0).$
...
...
$q_1(1k\ A_i, [[n2, 1k|A_i]|A_0], A_0, L_i,$
$[[L_i]|L_0], L_0, T_i, [[T_i]|T_0], T_0).$

$l1$ to lk are the kernel positions corresponding to $n1$. In phrase structure grammar, elements with lj for their left position are not always nonterminal symbols.

It may be hard to calculate the kernel positions corresponding to $n1$, as mentioned before. However, this does not mean that parsing is impossible.

Parsing can be performed by either of the following two methods:

The first method is to extend the parsing path to $n2$, assuming that the first element of the parsing path is a corresponding kernel position.

This is not efficient in terms of time and space, but it does ensure correct parsing. Since parsing is executed from the bottom up in this method, the ε rule is prohibited.

The second method is more practical and permits use of the ε rule.

Consider only the first element on the left-hand side of a phrase structure rule. All the kernel positions corresponding to the left-corner element can be calculated in the same way as in restricted context-free grammars.

These are called pseudo-kernel positions.

As is easily seen, the original kernel positions are also pseudo-kernel positions.

Left-corner action can be created by using the pseudo-kernel positions.

4.2 Inside Type

The Horn clause corresponding to Q_2 is as follows:

$q_2(n2, A_i, [[n3|A_i]|A_0], A_0, L_i,$
$[[L_i]|L_0], L_0, T_i, [[T_i]|T_0], T_0).$

This is exactly the same as before

4.3 Right-Corner Type

The Horn clause corresponding to Q_n should be determined as follows:

$q_n(nn, [n|A_i], A_0, A_{01}, L_i, L_0, L_{01}, T_i, T_0, T_{01}):-$
$p_1(n, A_i, A1, A_{01}, L_i, L1, L_{01}, T_i, T1, T_{01}),$
$p_2 00(A1, A2, L1, L2, T1, T2),$
...
...
$p_m 00(A_{m-1}, A_0, L_{m-1}, L_0, T_{m-1}, T_0).$

In phrase structure grammars, left-hand side predicates are activated sequentially in the prescribed order. This assumes that a partial input string corresponding to all the left-hand side elements has been detected.

From the viewpoint of the derivation from $S0$, the right corner action is to proceed along the parsing path from the right position of the last element of to the left position of the first element of γ in the following:

$$So \stackrel{*}{\Rightarrow} \alpha P_1 P_2 \cdots P_m \gamma \Rightarrow \alpha Q_1 Q_2 \cdots Q_n \gamma$$

For $p_2 00$ to $p_m 00$, the parsing stack (input) generally consists of more than one parsing path, in the same way that predicates correspond to input sentences. The only difference is that the list ends with a variable in this case.

Therefore, the following Horn clause should be created for the interface:

$p_j 00([[N|A_h]|A_t], A_j, [L_h|L_t], L_j, [T_n|T_t], T_j):-$
$p_j(N, A_h, A_j, A_{j1}, L_h, L_j, L_{j1}, T_h, T_j, T_{j1}),$
$p_j 00(A_t, A_{j1}, L_t, L_{j1}, T_t, T_{j1}).$
$p_j 00(A_{01}, A_{01}, L_{01}, L_{01}, T_{01}, T_{01},):-!.$

For the right-corner action in the pseudo-kernel position system, the predicates corresponding to the left-hand side elements should also be activated sequentially.

The kernel position system always has the final parsing path. In the pseudo-kernel position system, however, the parsing paths may disappear during successive predicate activation.

The pseudo-kernel position system is not as efficient as the kernel position system, but is more efficient than the bottom-up system. It may be used as an intermediate system between them.

5. Operational Example

This section explains how the parsing system works, using simple instances of Japanese grammar as illustrations.

Figure 1 shows an example of grammar. This was created by slightly revising Ref. 9.

[e] is a period. [p] is a terminal symbol for a phrase and has the following parameters:

W: Independent word + adjunct
$C1$: Part of speech of independent word
$C2$: Part of speech or inflection of adjunct
rel is a nonterminal symbol representing a dependency relation.

There are no rules that have only rel on the left-hand side. Therefore, no terminal symbols are derived from them.

To make things simple, no parameters are given to

```
0. s0 → ᵇs ᵃ ⊣

1. s  → ' prseq ² [p ] ( w,c1,c2) {c2=end} ³ [e ]

2. prseq → ⁴ [p ] (w,c1,c2) ⁵rel

3. prseq → ⁶ [p ] (w,c1,c2) ⁷rel ⁸prseq

4. [p ] (w1,c11,c12) rel [p ] (w2,c21,c22) → ⁹ [p ] (w1,c11,c12) ¹⁰prseq
     '' [p ] (w2,c21,c22) {check(c11,c12,c21),put(w1,w2) }

5. [p ] (w1,c11,c12) rel [p ] (w2,c21,c22) → ¹¹ [p ] (w1,c11,c22) ¹²rel
     ¹⁴prseq ¹⁵ [p ] (w2,c21,c22) {check(c11,c12,c21),put(w1,w2) }

6. [p ] (w1,c11,c12) rel [p ] (w2,c21,c22) → ¹⁶ [p ] (w1,c11,c12) ¹⁷ [p ]
     (w2,c21,c22) {check(c11,c12,c21),put(w1,w2) }


     check(n,ga,vt).
     check(n,ni,vt).
     check(n,wo,vt).
     check(vt,cn,n).
```

Fig. 1 Example of phrase structure grammar.

"*s*" or "prseq" here. "check" in Rules 1, 4, 5, and 6 is a simple predicate that checks whether dependency is syntactically possible. The "put" instruction stores the dependency relation ($W1$, $W2$) in the parse tree stack.

The basic constraints in Rule 1 indicate that the final phrase must have an end form.

In this example, all the kernel positions can be calculated.

There are seven kernel positions (*b*, 8, 10, 14, 11, 15, and 17) corresponding to the left-corner elements $^{9}[p]$, $^{12}[p]$, and $^{16}[p]$.

Assume that the input sentence is "tarouga jirouni, renrakusuru kotowo yakusokusuru."

Assume the following input predicates to this system.

yopen ($A1$, $L1$, $T1$),
$p0$ ($A1$, $A2$, $L1$, $L2$, $T1$, $T2$, [tarouga, *n*, *ga*]),
$p0$ ($A2$, $A3$, $L2$, $L3$, $T2$, $T3$, [jirouni, *n*, *ni*]),
$p0$ ($A3$, $A4$, $L3$, $L4$, $T3$, $T4$, [renrakusru, *vt*, *cn*]),
$p0$ ($A4$, $A5$, $L4$, $L5$, $T4$, $T5$, [kotowo, *n*, *wo*]),
$p0$ ($A5$, $A6$, $L5$, $L6$, $T5$, $T6$, [yakusokusuru, *vt*, end]),
$e0$ ($A6$, $A7$, $L6$, $L7$, $T6$, $T7$, ['.', end, end]),
yclose ($A7$, $L7$, $T7$).

n: Noun
vt: Transitive verb
cn: Conjunctive
end: End form
yopen: Predicate to initialize the stack
yclose: Predicate to output the result (dependency structure)

There are 22 parsing charts. From the viewpoint of the dependency structure, however, the results can be reduced to the five shown in Fig. 2.

```
(1)
    (tarouga,yakusokusuru) (jirouni,yakusokusuru) (jirouni,renrakusuru)
    (renrakusuru,kotowo)   ( kotowo,yakusokusuru)
(2)
    (tarouga,yakusokusuru) (jirouni,yakusokusuru)
    (renrakusuru,kotowo)   ( kotowo,yakusokusuru)
(3)
    (tarouga,yakusokusuru)                        (jirouni,renrakusuru)
    (renrakusuru,kotowo)   ( kotowo,yakusokusuru)
(4)
                           ( tarouga,renrakusuru) (jirouni,renrakusuru)
    (renrakusuru,kotowo)   ( kotowo,yakusokusuru)
(5)
    (tarouga,yakusokusuru) (tarouga,renrakusuru)  (jirouni,renrakusuru)
    (renrakusuru,kotowo)   ( kotowo,yakusokusuru)
```

Fig. 2 Result of analysis of input sentence.

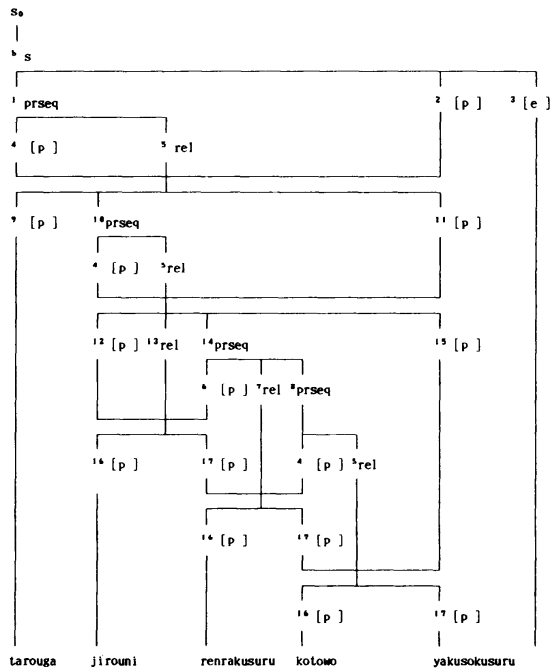

Fig. 3 Parsing chart of input sentence.

Figure 3 shows one of the parsing charts fo the above input. This example remains ambiguous until context processing is performed.

Since this system is based on the extended LR method, a parsing chart is created from the bottom up as in the conventional YAPXR system.

In Fig. 3, a partial chart is created for "kotowo, yakusokusuru" according to Rule 6. Then the "*p*, rel, *p*" predicates are activated. The first *p* generates a par-

Table 1 Performance result.

	Kernel method	Pseudo-kernel method
Parsing time	700 ms	800 ms
Memory size	44,372 bytes	261,452 bytes
Number of parsing paths to [e]	1,225	1,460

SUN 3/60
Quintus Prolog R2.4

```
0. s0 → 's' ┤

1. s  → '1' prseq '2' [p] (w,c1,c2) {c2=end} '3' [e]

2. prseq → '4' pr(w,c1,c2)

3. prseq → '5' pr(w,c1,c2) '6'prseq

4. pr(w1,c11,c12) [p] (w2,c21,c22) → '7' [p] (w1,c11,c12) '8'prseq
        '9' [p] (w2,c21,c22) {check(c11,c12,c21),put(w1,w2)}

5. pr(w1,c11,c12) [p] (w2,c21,c22) → '10' pr(w1,c11,c22) '11'prseq
                '12' [p] (w2,c21,c22) {check(c11,c12,c21),put(w1,w2)}

6. pr(w1,c11,c12) [p] (w2,c21,c22) → '13' [p] (w1,c11,c12) '14' [p]
        (w2,c21,c22) {check(c11,c12,c21),put(w1,w2)}

      check(n,ga,vt).
      check(n,ni,vt).
      check(n,wo,vt).
      check(vt,cn,n).
```

Fig. 4 Revised version of the phrase structure grammar.

tial chart for "renrakusuru, kotowo" through right-corner action based on Rule 6, and activates "p, rel, p" again. The parsing chart shown in the figure is generated through the same procedure.

The difference from conventional RCSGs is that terminal element $^{17}[p]$ is the kernel position of the left-corner element $^{16}[p]$. Table 1 shows the parsing time and working memory size for the example.

Unlike in the kernel position system, $^{2}[p]$ is added to the left-corner elements $^{9}[p]$, $^{12}[p]$, and $^{16}[p]$ as a pseudo-kernel position in the pseudo-kernel position system.

There are redundant parsing charts for the example sentence. Figure 4 shows the grammar revised to match the number of different dependency structures. Table 2 shows the result of operating this grammar.

There are no further differences between the kernel and pseudo-kernel position systems. The performance is five or six times higher than before.

The appendix shows a parsing program (Quintus Prolog version) based on the revised grammar.

The appendix also shows the Horn clause for the input predicates.

Table 2 Revised performance result.

	Kernel (pseudo-kernel) method
Parsing time	133 ms
Memory size	44,372 bytes
Number of parsing paths to [e]	205

SUN 3/60
Quintus Prolog R2.4

6. Conclusion

This paper has described an efficient parsing system developed for general phrase structure grammars by extending the YAPXR Breadth-first Top-down Parsing System.

For languages such as Japanese that have free order and abbreviation, it is not always appropriate to describe complete sentence structures within the framework of a context-free grammar. Gapping grammars are capable of handling languages with free word order, but have rather poor performance.

If the dependency relationships of the Japanese language is considered as syntactic events, a phrase structure grammar is more suitable. This system should be very effective.

Like conventional YAPXR, the new method has the following features:

(1) All the kernel or pseudo-kernel positions corresponding to the left-corner element can be obtained as soon as a grammar is given. No top-down forecasting is necessary during execution.

(2) The Prolog compiler allows double hashing with the first parameter as the atom.

(3) No back-tracking occurs during execution.

These features make the parsing system very efficient.

The major goal for the future is to develop a method of calculating kernel positions within the wide range of a phrase structure grammar.

This paper has only introduced the basic techniques of the new method shown its potential advantages.

Further studies are required in order to apply the method to the Japanese language on a practical level for machine translation or natural language interfaces.

Acknowledgment

I would like to express my deep gratitude to Professor Hozumi Tanaka (Tokyo Institute of Technology) for his continuous guidance, and also to Shunji Miya, Toshiya Sakamaki, and Kenichi Yoshida (Fujitsu SSL) for their efforts in developing this system.

References

1. HAYASHI, T. YAP: Yet Another Efficient Parsing Method for General Context-Free Grammars Based on the Logic Programming Language, *Trans. IPS Japan*, **29**, 9 (1988), 835-842.
2. HAYASHI, T. YAPX: An Extended Context-Free Grammar and Its Parsing Method Based on the Logic Programming Language, *Trans. IPS Japan*, **29**, 5 (1988), 480-487.
3. HAYASHI, T. An Efficient Implementation Method of YAPX Parsing System, *Trans. IPS Japan*, **30**, 10 (1989), 1354-1356.
4. HAYASHI, T., MIYA, S., SAKAMAKI, T. and YOSHIDA, K. Implementation and Evaluation of the Breadth-first Top-down Sentence Analysis System, *SIGNLP IPS Japan*, **74**-9 (1989), 65-72.
5. HAYASHI, T. Implementation and Evaluation of the Sentence Analysis System YAPXR, *Trans. IPS Japan*, **31**, 7 (1990), 970-978.
6. DAHL, V. and ABRAMSON, H. On Gapping Grammars, *Proc. 2nd International Conference on Logic Programming* (1984), 77-88.
7. DAHL, V. More on Gapping Grammars, *Proc. FGCS* (1984), 669-677.
8. POPOWICH, F. Unrestricted Gapping Grammars, *Proc. IJCAI 85* (1985), 756-768.
9. SUGIMURA, R. Logical Dependency Grammar and Its Constraint Analysis, ICOT Technical Memorandum TM-0679, p. 10 (1989).
10. HAYASHI, T., MIYA, S., SAKAMAKI, T. and YOSHIDA, K. An Efficient Sentence Analysis Method for General Phrase Structure Grammars, *SIGNLP IPS Japan*, 76-2, p. 8 (1990).
11. HAYASHI, T. An Efficient Sentence Analysis Method for General Phrase Structure Grammars, *Trans. IPS Japan*, **31**, 12 (1990), 1718-1726.

Appendix Parsing program for the revised grammar

```
1  yapxr_2(_Outfile):-
2      tell(_Outfile),
3      statistics,
4  %   statistics(runtime,_),
5      yopen(_Ai1,_Li1,_Ti1),!,
6      p0(_Ai1,_Ai2,_Li1,_Li2,_Ti1,_Ti2,[tarouga,n,ga]),!,
7      p0(_Ai2,_Ai3,_Li2,_Li3,_Ti2,_Ti3,[jirouni,n,ni]),!,
8      p0(_Ai3,_Ai4,_Li3,_Li4,_Ti3,_Ti4,[renrakusuru,vt,cn]),!,
9      p0(_Ai4,_Ai5,_Li4,_Li5,_Ti4,_Ti5,[kotowo,n,wo]),!,
10     p0(_Ai5,_Ai6,_Li5,_Li6,_Ti5,_Ti6,[yakusokusuru,vt,end]),!,
11 %   count_pass(_Ai6,0),
12     e0(_Ai6,_Ai7,_Li6,_Li7,_Ti6,_Ti7,['.',end,end]),!,
13 %   statistics(runtime,[_,_A]),
14 %   print('rumtime is '), print(_A),
15     statistics,
16     yclose(_Ai7,_Li7,_Ti7),
17     close(_Outfile).
18
19
20 yopen([[b]],[[]],[[]]):-!.
21
22 p0([[_N|_Ah]|_At],_Ao,[_Lh|_Lt],_Lo,[_Th|_Tt],_To,[_W,_C1,_C2]):- !,
23     p(_N,_Ah,_Ao,_Ao1,[_W,_C1,_C2|_Lh],_Lo,_Lo1,_Th,_To,_To1),
24     p0(_At,_Ao1,_Lt,_Lo1,_Tt,_To1,[_W,_C1,_C2]).
25 p0([],[],_,[],_,[],_):- !.
26 e0([[_N|_Ah]|_At],_Ao,[_Lh|_Lt],_Lo,[_Th|_Tt],_To,[_W,_C1,_C2]):- !,
27     e(_N,_Ah,_Ao,_Ao1,[_W,_C1,_C2|_Lh],_Lo,_Lo1,_Th,_To,_To1),
28     e0(_At,_Ao1,_Lt,_Lo1,_Tt,_To1,[_W,_C1,_C2]).
29 e0([],[],_,[],_,[],_):- !.
30
31 s(b,_Ai,[[e|_Ai]|_Ao],_Ao,_Li,[_Li|_Lo],_Lo,_Ti,[_Ti|_To],_To):- !.
32 s(_,_,_Ai,_Ai,_,_Li,_Li,_,_Ti,_Ti):- !.
33
34 prseq(b,_Ai,[[2,b|_Ai]|_Ao],_Ao,_Li,[_Li|_Lo],_Lo,_Ti,[_Ti|_To],_To):- !.
35 prseq(8,[_N|_Ai],_Ao,_Ao1,_Li,_Lo,_Lo1,_Ti,_To,_To1):- !,
36     prseq(_N,_Ai,_Ao,_Ao1,_Li,_Lo,_Lo1,_Ti,_To,_To1).
37 prseq(10,_Ai,[[11|_Ai]|_Ao],_Ao,_Li,[_Li|_Lo],_Lo,_Ti,[_Ti|_To],_To):- !.
38 prseq(14,_Ai,[[15|_Ai]|_Ao],_Ao,_Li,[_Li|_Lo],_Lo,_Ti,[_Ti|_To],_To):- !.
39 prseq(_,_,_Ai,_Ai,_,_Li,_Li,_,_Ti,_Ti):- !.
40
41 pr(b,_Ai,[[8,b|_Ai],[14,b|_Ai]|_Ao],_Ao1,
42     [_W,_C1,_C2|_Li],
43     [_Li,[_W,_C1,_C2|_Li]|_Lo],_Lo1,
44     _Ti,[_Ti,_Ti|_To],_To1):- !,
45     prseq(b,_Ai,_Ao,_Ao1,_Li,_Lo,_Lo1,_Ti,_To,_To1).
46 pr(8,_Ai,[[8,8|_Ai],[14,8|_Ai]|_Ao],_Ao1,
47     [_W,_C1,_C2|_Li],
48     [_Li,[_W,_C1,_C2|_Li]|_Lo],_Lo1,
49     _Ti,[_Ti,_Ti|_To],_To1):- !,
50     prseq(8,_Ai,_Ao,_Ao1,_Li,_Lo,_Lo1,_Ti,_To,_To1).
51 pr(10,_Ai,[[8,10|_Ai],[14,10|_Ai]|_Ao],_Ao1,
52     [_W,_C1,_C2|_Li],
53     [_Li,[_W,_C1,_C2|_Li]|_Lo],_Lo1,
54     _Ti,[_Ti,_Ti|_To],_To1):- !,
55     prseq(10,_Ai,_Ao,_Ao1,_Li,_Lo,_Lo1,_Ti,_To,_To1).
56 pr(14,_Ai,[[8,14|_Ai],[14,14|_Ai]|_Ao],_Ao1,
```

```
57      |_W,_C1,_C2|_Li],
58      [_Li,[_W,_C1,_C2|_Li]|_Lo],_Lo1,
59      _Ti,[_Ti,_Ti|_To],_To1):- !,
60      prseq(14,_Ai,_Ao,_Ao1,_Li,_Lo,_Lo1,_Ti,_To,_To1).
61  pr(_,_,_Ai,_Ai,_,_Li,_Li,_,_Ti,_Ti):- !.
62
63  e(3,[_N|_Ai],_Ao,_Ao1,[_W,_C1,_C2|_Li],_Lo,_Lo1,_Ti,_To,_To1):- !,
64      s(_N,_Ai,_Ao,_Ao1,_Li,_Lo,_Lo1,_Ti,_To,_To1).
65  e(_,_,_Ai,_Ai,_,_Li,_Li,_,_Ti,_Ti):- !.
66
67  p(2,_Ai,_Ao,_Ao1,[_W,_C1,_C2|_Li],_Lo,_Lo1,_Ti,_To,_To1):-
68      ((_C2 = end,!,
69        _Ao = [[3|_Ai]|_Ao1],
70        _Lo = [_Li|_Lo1],
71        _To = [_Ti|_To1]);
72       (_Ao = _Ao1, _Lo = _Lo1, _To = _To1)).
73  p(b,_Ai,[[10,b|_Ai],[17,b|_Ai]|_Ao],_Ao,
74      [_W,_C1,_C2|_Li],
75      [[_W,_C1,_C2|_Li],[_W,_C1,_C2|_Li]|_Lo],_Lo,
76      _Ti,[_Ti,_Ti|_To],_To):- !.
77  p(8,_Ai,[[10,8|_Ai],[17,8|_Ai]|_Ao],_Ao,
78      [_W,_C1,_C2|_Li],
79      [[_W,_C1,_C2|_Li],[_W,_C1,_C2|_Li]|_Lo],_Lo,
80      _Ti,[_Ti,_Ti|_To],_To):- !.
81  p(10,_Ai,[[10,10|_Ai],[17,10|_Ai]|_Ao],_Ao,
82      [_W,_C1,_C2|_Li],
83      [[_W,_C1,_C2|_Li],[_W,_C1,_C2|_Li]|_Lo],_Lo,
84      _Ti,[_Ti,_Ti|_To],_To):- !.
85  p(14,_Ai,[[10,14|_Ai],[17,14|_Ai]|_Ao],_Ao,
86      [_W,_C1,_C2|_Li],
87      [[_W,_C1,_C2|_Li],[_W,_C1,_C2|_Li]|_Lo],_Lo,
88      _Ti,[_Ti,_Ti|_To],_To):- !.
89  p(11,[_N|_Ai],[[10,11,_N|_Ai],[17,11,_N|_Ai]|_Ao2],_Ao1,
90      [_W2,_C21,_C22,_W1,_C11,_C12|_Li],
91      [[_W2,_C21,_C22,_W1,_C11,_C12|_Li],
92      [_W2,_C21,_C22,_W1,_C11,_C12|_Li]|_Lo2],_Lo1,
93      _Ti,[_Ti,_Ti|_To2],_To1):-
94        check(_C11,_C12,_C21),!,
95        pr(_N,_Ai,_Ao,_Ao1,[_W1,_C11,_C12|_Li],_Lo,_Lo1,
96          [(_W1,_W2)|_Ti],_To,_To1),
97        p00(_Ao,_Ao2,_Lo,_Lo2,_To,_To2,[_W2,_C21,_C22]).
98  p(15,[_N|_Ai],[[10,15,_N|_Ai],[17,15,_N|_Ai]|_Ao2],_Ao1,
99      [_W2,_C21,_C22,_W1,_C11,_C12|_Li],
100     [[_W2,_C21,_C22,_W1,_C11,_C12|_Li],
101     [_W2,_C21,_C22,_W1,_C11,_C12|_Li]|_Lo2],_Lo1,
102     _Ti,[_Ti,_Ti|_To2],_To1):-
103       check(_C11,_C12,_C21),!,
104       pr(_N,_Ai,_Ao,_Ao1,[_W1,_C11,_C12|_Li],_Lo,_Lo1,
105         [(_W1,_W2)|_Ti],_To,_To1),
106       p00(_Ao,_Ao2,_Lo,_Lo2,_To,_To2,[_W2,_C21,_C22]).
107 p(17,[_N|_Ai],[[10,17,_N|_Ai],[17,17,_N|_Ai]|_Ao2],_Ao1,
108     [_W2,_C21,_C22,_W1,_C11,_C12|_Li],
109     [[_W2,_C21,_C22,_W1,_C11,_C12|_Li],
110     [_W2,_C21,_C22,_W1,_C11,_C12|_Li]|_Lo2],_Lo1,
111     _Ti,[_Ti,_Ti|_To2],_To1):-
112       check(_C11,_C12,_C21),!,
113       pr(_N,_Ai,_Ao,_Ao1,[_W1,_C11,_C12|_Li],_Lo,_Lo1,
114         [(_W1,_W2)|_Ti],_To,_To1),
115       p00(_Ao,_Ao2,_Lo,_Lo2,_To,_To2,[_W2,_C21,_C22]).
116 p(_,_,_Ai,_Ai,_,_Li,_Li,_,_Ti,_Ti):- !.
117
118 p00(_Ai,_Ai,_Li,_Li,_Ti,_Ti,_):-
119     var(_Ai),!.
120 p00([[_N|_Ah]|_At],_Ao,[_Lh|_Lt],_Lo,
121       [_Th|_Tt],_To,[_W1,_C11,_C12]):- !,
122     p(_N,_Ah,_Ao,_Ao1,[_W1,_C11,_C12|_Lh],_Lo,_Lo1,_Th,_To,_To1),
123     !,
124     p00(_At,_Ao1,_Lt,_Lo1,_Tt,_To1,[_W1,_C11,_C12]).
125
126
127 check(n,ga,vt):- !.
128 check(n,ni,vt):- !.
129 check(n,wo,vt):- !.
130 check(vt,cn,n):- !.
131
132
133 yclose([],[],[]):- !.
134 yclose([_Acar|_Acdr],[_Lcar|_Lcdr],[_Tcar|_Tcdr]):- !,
135     print('Ai='),print(_Acar),nl,
136     print('Li='),print(_Lcar),nl,
137     print('Ti='),print(_Tcar),nl,
138     yclose(_Acdr,_Lcdr,_Tcdr).
139
140 count_pass([],_N):- !,
141     print('pass_num = '),print(_N),nl.
142 count_pass([_A|_B],_N):- !,
143     _M is _N + 1,
144     count_pass(_B, _M).
145
146
```