TR-548

Alexander Parser

by
T. Kanamori (Mitsubishi)

April, 1990

Alexander Parser

Tadashi KANAMORI

Central Research Laboratory
Mitsubishi Electric Corporation
8-1-1, Tsukaguchi-Honmachi
Amagasaki, Hyogo, JAPAN 661

Abstract

This paper presents a hybrid (i.e., bi-directional) parser using the bottom-up interpreter of logic programs, called *Alexander Parser*. The algorithm once translates a given context free grammar into a logic program, and then interpretes it in the bottom-up manner. Not only reflects the translation the behavior of top-down parsing but also the bottom-up interpretation avoids the infinite loops due to left-recursive rules. This algorithm is an adaptation of Seki's query evaluation method in deductive databases, called "Alexander Templates," to parsing algorithms as well as a refinement of Fuchi's description of Earley's parsing algorithm within the framework of logic programming.

Keywords: Parsing, Context Free Grammar, Logic Programming.

Contents

1. Introduction

This paper presents a hybrid (i.e., bi-directional) parser using the bottom-up interpreter of logic programs, called *Alexander Parser.* The algorithm once translates a given context free grammar into a logic program, and then interpretes it in the bottom-up manner. Not only reflects the translation the behavior of top-down parsing but also the bottom-up interpretation avoids the infinite loops due to left-recursive rules. Moreover, the similarity of the bottom-up interpreter to the database operations provides a possibility of massive parsing with a large database of words. This algorithm is an adaptation of Seki's query evaluation method in deductive databases, called "Alexander Templates," to parsing algorithms as well as a refinement of Fuchi's description of Earley's parsing algorithm within the framework of logic programming.

The rest of this paper is organized as follows: Section 2 briefly explains the context free gammars (CFGs) we consider in this paper. Section 3 presents our "Alexander Parser" starting from its naive version and then shifting to its refined version. (The correctness of the algorithm is proved in Appendix.) Section 4 discusses the issue of the degree of disagreement between the control of CFG parsers desired and the control of logic program interpreter used for implementation, and the degree of sophistication in the translation from CFGs to logic programs by examining other translators known so far.

2. Context Free Grammar (CFG)

We consider the following class of context free grammars. (Familiarity with context free grammars is assumed so that the detailed definitions are omitted.)

Grammar Rule:

A *grammar rule* is an expression of the form

$$c_0 \rightarrow c_1\ c_2\ \cdots\ c_m \qquad (m > 0),$$

where $c_0, c_1, c_2, \ldots, c_m$ are non-terminal symbols. Such a grammar rule means that the sequence of grammatical categories $c_1, c_2, \ldots, c_m$ is generated from the grammatical category c_0. Finite sets of grammar rules are denoted by P.

In the following, we assume that each grammar rule in P is assigned a unique natural number, called the *rule number.*

Example 2.1: The expression below is a grammar rule.

s → np vp .

Dictionary Rule:

A *dictionary rule* is an expression of the form

$$c \rightarrow w,$$

where c is a non-terminal symbol and w is a terminal symbol. Such a dictionary rule means that the terminal symbol w is of the grammatical category c. Finite sets of dictionary rules are denoted by D.

In the following, we assume that the size of the set of dictionary rules is much larger than the size of the set of grammar rules.

Example 2.2: The expressions below are dictionary rules.

np → "john" ,

vp $\rightarrow$ "walks" .

Input Rule:

An *input rule* is an expression either of the form

$c \rightarrow w_1\ w_2\ \cdots\ w_k \qquad (k > 0)$

or of the form

$C \rightarrow w_1\ w_2\ \cdots\ w_k \qquad (k > 0),$

where c is a non-terminal symbol, C is a variable denoting a non-terminal symbol, and $w_1, w_2, \ldots, w_k$ are terminal symbols. The first expression is a question whether the sequence of non-terminal symbols is generated from the grammatical category c, while the second expression is a question from what grammatical category the sequence of non-terminal symbols is generated. Input rules are denoted by I.

In the following, we consider only the input rules of the first form. The generalization to include the input rules of the second form is immediate.

Example 2.3: The expressions below are input rules.

s $\rightarrow$ "john" "walks" ,
C $\rightarrow$ "john" "walks" .

3. Alexander Parser

3.1 Naive Alexander Parser

The naive "Alexander Parser" receives a set of grammar rules, a set of dictionary rules and an input rule. It first translates them into a set of definite clauses, a set of unit clauses and a (singleton) set of atoms, and then interpretes them using the bottom-up intertreter.

In the "translation" phase, we introduce the following 3 predicates.

- "$parsing(C, X, Y)$" means that we are now *parsing* the sequence from X to Y to grammatical category C. (The pair of X and Y denotes a difference list.)
- "$parsed(C, X, Y)$" means that the sequence from X to Y has been *parsed* to grammatical category C.
- "$dictionary(w, c)$" is used to retrieve the dictionary rule "$c \rightarrow w$" from database. (It is always used with one-way matching.)

The "translation" phase is described as below:

Algorithm "translate0"

Input : a set of grammar rules P and a set of dictionary rules D.
Output : a pair of program and database $(\bar{P}, \bar{D})$.

Procedure :
step 0 : Initialize $\bar{P}$ to $\emptyset$.
step 1 : For each rule in P, say of the form

"$c_0 \rightarrow c_1, c_2, \ldots, c_m$" $\qquad (m > 0),$

add the $m + 1$ clauses below to $\bar{P}$:

$parsing(c_1, X_0, X_1)$:- $parsing(c_0, X_0, X_m)$.
$parsing(c_2, X_1, X_2)$:- $parsing(c_0, X_0, X_m), parsed(c_1, X_0, X_1)$.
$parsing(c_3, X_2, X_3)$:- $parsing(c_0, X_0, X_m), parsed(c_1, X_0, X_1), parsed(c_2, X_1, X_2)$.

$\vdots$

$parsing(c_m, X_{m-1}, X_m)$:- $parsing(c_0, X_0, X_m), parsed(c_1, X_0, X_1),$
$parsed(c_2, X_1, X_2), \ldots, parsed(c_{m-1}, X_{m-2}, X_{m-1}).$

$parsed(c_0, X_0, X_m)$:- $parsing(c_0, X_0, X_m), parsed(c_1, X_0, X_1), parsed(c_2, X_1, X_2), \ldots$
$parsed(c_{m-1}, X_{m-2}, X_{m-1}), parsed(c_m, X_{m-1}, X_m).$

step 2 : Add the clause below to $\tilde{P}$:
$parsed(C, [W|X], X)$:- $parsing(C, [W|X], X), dictionary(W, C).$

step 3 : Let $\tilde{D}$ be the set of unit clauses "$dictionary(w, c)$" such that dictionary rule "$c \rightarrow w$" is in D.

step 4 : Return $(\tilde{P}, \tilde{D})$.

Algorithm "translate-input"

Input : an input rule I.
Output : a set of atoms $\tilde{I}$.

Procedure : Let I be of the form "$c \rightarrow w_1\ w_2\ \cdots\ w_k$." Let $\tilde{I}$ be
$\{parsing(c, [w_1, w_2, \ldots, w_k], [\,])\}$.
Return $\tilde{I}$.

Example 3.1.1 The CFG before is translated as below by this translator:
Program and Database:

```
parsing(np,X,Y) :- parsing(s,X,Z).
parsing(vp,Y,Z) :- parsing(s,X,Z), parsed(np,X,Y).
parsed(s,X,Z) :- parsing(s,X,Z), parsed(np,X,Y), parsed(vp,Y,Z).
parsed(C,[W|X],X) :- parsing(C,[W|X],X), dictionary(W,C).
dictionary(john,np).
dictionary(walks,vp).
```

Set of Atoms:

```
parsing(s,[john,walks],[ ]).
```

Let $\tilde{P}$ be a program, Γ be a set of atoms, $\tilde{C}$ be a clause in $\tilde{P}$, and θ be a substitution for the variables appearing in the body atoms of $\tilde{C}$. Then, atom $A\theta$ is said to be *generated from* Γ *using* $\tilde{C}$ when

- atom A is the head atom of $\tilde{C}$, and
- Γ contains instances of the body atoms of $\tilde{C}$ by θ.

The "interpretation" phase is described as below:

Algorithm "interprete0"

Input : a program $\tilde{P}$, a database $\tilde{D}$, and a set of atoms $\tilde{I}$.
Output : a set of atoms.

Procedure :
step 0: Initialize Γ to Γ_0, where Γ_0 is $\tilde{I} \cup \tilde{D}$.
step 1: Update Γ to $\Gamma' \cup \Gamma_0$, where Γ' is the set of all the atoms generated from Γ using some clause in $\tilde{P}$. Repeat this step until Γ does not increase.
step 2: Return the set of all atoms of the form "$parsed(c, l_1, l_2)$" in Γ.

Example 3.1.2 For the translated program, database and set of atoms before, the naive bottom-up interpreter generates the atoms below step by step at **step 1**.

Repetition 0: $parsing(s, [john, walks], [\,])$.
Repetition 1: $parsing(np, [john, walks], Y)$.
Repetition 2: $parsed(np, [john, walks], [walks])$.
Repetition 3: $parsing(vp, [walks], [\,])$.
Repetition 4: $parsed(vp, [walks], [\,])$.
Repetition 5: $parsed(s, [john, walks], [\,])$.

3.2 Refined Alexander Parser

When atoms

$$parsing(c_0, X_0, X_m), parsed(c_1, X_0, X_1), \ldots, parsed(c_i, X_{i-1}, X_i)$$

are in the body of more than two caluses in $\tilde{P}$, the same combination of the sequences must be checked for more than two clauses repeatedly in the "interpretation" phase. To save the same combinations to a table, the refined "translation" phase utilizes a predicate "*table*" as below, where a clause with two head atoms

$$A_0, A_0' \text{ :- } A_1, A_2, \ldots, A_k$$

is just a convention of writing two clauses

$$A_0 \text{ :- } A_1, A_2, \ldots, A_k$$
$$A_0' \text{ :- } A_1, A_2, \ldots, A_k.$$

Algorithm "translate1"

Input : a set of grammar rules P and a set of dictionary rules D.
Output : a pair of program and database $(\tilde{P}, \tilde{D})$.

```
Procedure :
step 0 : Initialize $\tilde{P}$ to $\emptyset$.
step 1 : For each rule in $P$, say with rule number $n$ of the form
      "$c_0 \rightarrow c_1, c_2, \ldots, c_m$"      $(m > 0)$,
add the $m + 1$ clauses below to $\tilde{P}$:
    $parsing(c_1, X_0, X_1), table([n,1], [X_0, X_1, X_m])$ :- $parsing(c_0, X_0, X_m)$.
    $parsing(c_2, X_1, X_2), table([n,2], [X_0, X_2, X_m])$ :-
            $table([n,1], [X_0, X_1, X_m]), parsed(c_1, X_0, X_1)$.
    $parsing(c_3, X_2, X_3), table([n,3], [X_0, X_3, X_m])$ :-
            $table([n,2], [X_0, X_2, X_m]), parsed(c_2, X_1, X_2)$.
      ⋮
    $parsing(c_m, X_{m-1}, X_m), table([n,m], [X_0, X_m, X_m])$ :-
            $table([n, m-1], [X_0, X_{m-1}, X_m]), parsed(c_{m-1}, X_{m-2}, X_{m-1})$.
    $parsed(c_0, X_0, X_m)$ :- $table([n,m], [X_0, X_m, X_m]), parsed(c_m, X_{m-1}, X_m)$.
step 2 : Add the clause below to $\tilde{P}$:
    $parsed(C, [W|X], X)$ :- $parsing(C, [W|X], X), dictionary(W, C)$.
step 3 : Let $\tilde{D}$ be the set of unit clauses "$dictionary(w, c)$" such that dictionary rule "$c \rightarrow w$"
is in $D$.
step 4 : Return $(\tilde{P}, \tilde{D})$.
```

Example 3.2.1 The CFG before is translated as below by this translator:
Program and Database:

```
     parsing(np,X,Y), table([1,1],[X,Y,Z]) :- parsing(s,X,Z).
     parsing(vp,Y,Z), table([1,2],[X,Z,Z]) :-
            table([1,1],[X,Y,Z]), parsed(np,X,Y).
     parsed(s,X,Z) :- table([1,2],[X,Z,Z]), parsed(vp,Y,Z).
     parsed(C,[W|X],X) :- parsing(C,[W|X],X), dictionary(W,C).
     dictionary(john,np).
     dictionary(walks,vp).
Set of Atoms:
     parsing(s,[john,walks],[ ]).
```

According to the refinement of the "translation" phase, we need to generalize the "interpretation" phase to use the clauses with two head atoms. In addition, we adopt the "semi-naive" bottom-up interpretation.

Let $\tilde{P}$ be a program, Δ and Δ_{new} be sets of atoms, $\tilde{C}$ be a clause (possibly with two head atoms) in $\tilde{P}$, and θ be a substitution for the variables appearing in the body atoms of $\tilde{C}$. Then, atom $A\theta$ is said to be *generated from* (Δ, Δ_{new}) *using* $\tilde{C}$ when

- atom A is *in* the head of $\tilde{C}$, and
- Δ and Δ_{new} contains instances of the body atoms of $\tilde{C}$ by θ, and at least one of the instances is in Δ_{new}.

The refined "interpretation" phase is described as below:

Algorithm "interprete1"

Input : a program $\tilde{P}$, a database $\tilde{D}$, and a set of atoms $\tilde{I}$.
Output : a set of atoms.

Procedure :
step 0: Initialize Δ and Δ_{new} to $\tilde{I} \cup \tilde{D}$.
step 1: Update Δ to $\Delta' \cup \Delta$, where Δ' is the set of all the atoms generated from (Δ, Δ_{new}) using some clause in $\tilde{P}$. Update Δ_{new} to the difference between the new Δ and the previous Δ. Repeat this step until Δ does not increase.
step 2: Return the set of all atoms of the form "$parsed(c, l_1, l_2)$" in Δ.

Example 3.2.2 For the translated program, database and set of atoms before, the refined bottom-up interpreter generates the atoms below step by step at **step 1**:

Repetition 0: $parsing(s, [john, walks], [\,])$.
Repetition 1: $parsing(np, [john, walks], Y), table([1,1], [[john, walks], Y, [\,]])$.
Repetition 2: $parsed(np, [john, walks], [walks])$.
Repetition 3: $parsing(vp, [walks], [\,]), table([1,2], [[john, walks], [\,], [\,]])$.
Repetition 4: $parsed(vp, [walks], [\,])$.
Repetition 5: $parsed(s, [john, walks], [\,])$.

3.3 Correctness of Alexander Parser

The correctness of "Alexander Parser" is stated as below:

Theorem (Correctness of Alexander Parser)
Let P be a set of grammar rules, D be a set of dictionary rules, I be an input rule of the form "$c \rightarrow w_1\ w_2\ \cdots\ w_k$," $(\tilde{P}, \tilde{D})$ be the result of "*translate1*(P, D)," $\tilde{I}$ be the the result of "*translate-input*(I)." Then, the sequence of terminal symbols "$w_1, w_2, \ldots, w_k$"

is generated from the grammatical category c using P and D if and only if the result of "*interprete1*$(\bar{P}, \bar{D}, \bar{I})$" includes "*parsed*$(c, [w_1, w_2, \ldots, w_k], [\,])$."

Proof. See Apendix.

4. Discussion

"Alexander Parser" is a (breadth-first) hybrid parser using the (breadth-first) bottom-up interpreter. The translator is not very naive, because it needs to jump over the gap between the control of the desired parser and the control of the interpreter. In general, the degree of disagreement between the control of parsers and the control of interpreters coincides with the degree of complication of the translators. (More generally, the degree of disagreement between the control of the desired computation and the control of the given interpreter coincides with the degree of sophistication of program transformation.) Let us re-examine the parsers in the framework of logic programming known so far from this viewpoint.

(1) Top-down Parser Using the Top-down Interpreter

The parser by Pereira and Warren for Definite Clause Grammar (DCG) is a (depth-first) top-down parser using the (depth-first) top-down interpreter [5]. Their translator is relatively simple and direct as below, where we have slightly modified their original DCG translator for convenience of the comparison.

"translate" for the Top-down Parser Using the Top-down Interpreter

Input : a set of grammar rules P and a set of dictionary rules D.
Output : a pair of program and database $(\bar{P}, \bar{D})$.

```
Procedure :
step 0 : Initialize $\bar{P}$ to $\emptyset$.
step 1 : For each rule in $P$, say of the form
      "$c_0 \rightarrow c_1, c_2, \ldots, c_m$"      $(m > 0)$,
add the clause below to $\bar{P}$:
      $parse(c_0, X_0, X_m)$ :- $parse(c_1, X_0, X_1), parse(c_2, X_1, X_2), \ldots, parse(c_m, X_{m-1}, X_m)$.
step 2 : Add the clause below to $\bar{P}$:
      $parse(C, [W|X], X)$ :- $dictionary(W, C)$.
step 3 : Let $\bar{D}$ be the set of unit clauses "$dictionary(w, c)$" such that dictionary rule "$c \rightarrow w$"
is in $D$.
step 4 : Return $(\bar{P}, \bar{D})$.
```

"translate-input" for the Top-down Parser Using the Top-down Interpreter

Input : an input rule I.
Output : a query $\bar{I}$.

```
Procedure : Let $I$ be of the form "$c \rightarrow w_1\ w_2\ \cdots\ w_k$." Let $\bar{I}$ be
      "?- $parse(c, [w_1, w_2, \ldots, w_k], [\,])$."
Return $\bar{I}$.
```

Example 4.1 The CFG before is translated as below by this translator:
Program and Database:

```
     parse(s,X,Z) :- parse(np,X,Y), parse(vp,Y,Z).
     parse(C,[W|X],X) :- dictionary(C,[W|X],X).
     dictionary(john,np).
     dictionary(walks,vp).
```
Query:
```
     ?- parse(s,[john,walks],[ ]).
```

(2) Bottom-up Parser Using the Bottom-up Interpreter

The well-known Cocke-Younger-Kasami's parser [1] can be formalized in the framework of logic programming as a (breadth-first) bottom-up parser using the (breadth-first) bottom-up interpreter (with some systematization) as below, where l_i is the tail list $[w_{i+1}, w_{i+2}, \ldots, w_k]$, e.g., l_1 is $[w_2, \ldots, w_k]$ and l_k is [ ]. (The procedure "***translate-input***" returns a set of atoms for initializing the bottom-up interpretation when the bottom-up interpreter is employed. Notice the procedure "***translate-input***" for "Alexander Parser.")

"translate" for the Bottom-up Parser Using the Bottom-up Interpreter

Input : a set of grammar rules P and a set of dictionary rules D.
Output : a pair of program and database $(\tilde{P}, \tilde{D})$.

Procedure :
step 0 : Initialize $\tilde{P}$ to $\emptyset$.
step 1 : For each rule in P, say of the form
"$c_0 \rightarrow c_1, c_2, \ldots, c_m$" $\quad (m > 0)$,
add the clauses below to $\tilde{P}$.
$parse(c_0, X_0, X_m)$:- $parse(c_1, X_0, X_1), parse(c_2, X_1, X_2), \ldots, parse(c_m, X_{m-1}, X_m)$.
step 2 : Add the clause below to $\tilde{P}$:
$parse(C, [W|X], X)$:- $input([W|X], X), dictionary(W, C)$.
step 3 : Let $\tilde{D}$ be the set of unit clauses $dictionary(w, c)$ such that dictionary rule "$c \rightarrow w$" is in D.
step 4 : Return $(\tilde{P}, \tilde{D})$.

"translate-input" for the Bottom-up Parser Using the Bottom-up Interpreter

Input : an input rule I.
Output : a set of atoms $\tilde{I}$.

Procedure : Let I be of the form "$c \rightarrow w_1\ w_2\ \cdots\ w_k$." Let $\tilde{I}$ be
$\{input([w_1|l_1], l_1), input([w_2|l_2], l_2), \ldots, input([w_k|l_k], l_k)\}$.
Return $\tilde{I}$.

Example 4.2 The CFG before is translated as below by this translator:
Program and Database:
```
     parse(s,X,Z) :- parse(np,X,Y), parse(vp,Y,Z).
     parse(C,[W|X],X) :- input([W|X],X),dictionary(C,[W|X],X).
     dictionary(john,np).
     dictionary(walks,vp).
```
Set of Atoms:
```
     input([john,walks],[walks]),
input([walks],[ ]).
```

(3) Hybrid Parser Using the Hybrid Interpreter

An alternative approach to embody a hybrid parser is to employ a relatively simple and direct translator, but employ a more sophisticated hybrid interpreter. "Earley deduction" by Pereira and Warren [6] is the hybrid interpreter used for the hybrid parser as below, where we have again modified the original "Earley deduction" for the convenience of the comparison. (The translator is the same as that of Section 4 (1) for the top-down parser.)

"interprete" for the Hybrid Parser Using the Hybrid Interpreter

Input : a program $\tilde{P}$, a database $\tilde{D}$, and a query $\tilde{I}$.
Output : a set of atoms.

Procedure : Let $\tilde{I}$ be of the form "?- A," and $X_1, X_2, \ldots, X_n$ be the variables appearing in A ($n \geq 0$).
step 0 : Initialize Σ to { "$answer(X_1, X_2, \ldots, X_n)$:- A" }.
step 1 : Select a non-unit clause C in Σ, say with leftmost body atom B. When there exists a non-unit clause C' in $\tilde{P}$ whose head is unifiable with B, say with m.g.u. θ, add $\theta(C')$ to Σ, if it is not subsumed by any clause in Σ. When there exists a unit clause in $\tilde{D}$ or a unit clause in Σ whose head is unifiable with B, say with m.g.u. θ, add $\theta(C)$ to Σ with leftmost body atom deleted, if it is not subsumed by any clause in Σ. Repeat this step until Σ does not increase.
step 2 : Return the set of all the instances of "$answer(X_1, X_2, \ldots, X_n)$" in Σ.

Example 4.3 For the translated program, database and query before, the "Earley deduction" generates the clauses below step by step at **step 1**:

Repetetion 0: $answer$:- $parse(s, [john, walks], [\,])$.
Repetetion 1: $parse(s, [john, walks], [\,])$:-
$\quad parse(np, [john, walks], X_1), parse(vp, X_1, [\,])$.
Repetetion 2: $parse(np, [john, walks], [walks])$:- $dictionary(john, np)$.
Repetetion 3: $parse(np, [john, walks], [walks])$.
Repetetion 4: $parse(s, [john, walks], [\,])$:- $parse(vp, [walks], [\,])$.
Repetetion 5: $parse(vp, [walks], [\,])$:- $dictionary(walks, vp)$.
Repetetion 6: $parse(vp, [walks], [\,])$.
Repetetion 7: $parse(s, [john, walks], [\,])$.
Repetetion 8: $answer$.

(4) Bottom-up Parser Using the Top-down Interpreter

When the control of the desired behavior of parser and the control of the logic program interpreter for implementation differ, more complicated and indirect translation is needed as our "Alexander Parser." BUP by Matsumoto et al [4] is an example of the other combination, bottom-up (left-corner depth-first) parser using the (depth-first) top-down interpreter. The (naive version of the) translator is as below, where

- "$parsed(C, G, X, Z)$" means that the left corner of a sequence of terminal symbols has been *parsed* to grammatical category C, the rest of the sequence is from X to Z, and we are now parsing the whole sequence to G, and

- "$parsing(G,X,Y)$" means that we are now *parsing* the sequence from X to Y to grammatical category G.

"translate" for the Naive Version of BUP

Input : a set of grammar rules P and a set of dictionary rules D.
Output : a pair of program and database $(\tilde{P}, \tilde{D})$.

Procedure :
step 0 : Initialize $\tilde{P}$ to the set of the clause below:
$parsing(G,X,Z)$:- $dictionary(C,X,Y), parsed(C,G,Y,Z)$.
step 1 : For each rule in P, say of the form
"$c_0 \rightarrow c_1, c_2, \ldots, c_m$" $(m > 0)$,
add the clauses below to $\tilde{P}$:
$parsed(c_1, G, X_1, Y)$:-
$parsing(c_2, X_1, X_2), \ldots, parsing(c_m, X_{m-1}, X_m), parsed(c_0, G, X_m, Y)$.
step 2 : Add the clauses below to $\tilde{P}$:
$parsed(C,C,X,X)$.
$dictionary(C,[W|X],X)$:- $dictionary(W,C)$.
step 3 : Let $\tilde{D}$ be the set of unit clauses "$dictionary(w,c)$" such that dictionary rule "$c \rightarrow w$" is in D.
step 4 : Return $(\tilde{P}, \tilde{D})$.

"translate-input" for the Naive Version of BUP

Input : an input rule I.
Output : a query $\tilde{I}$.

Procedure : Let I be of the form "$c \rightarrow w_1\ w_2\ \cdots\ w_k$." Let $\tilde{I}$ be
"?- $parsing(c, [w_1, w_2, \ldots, w_k], [\,])$."
Return $\tilde{I}$.

Example 4.4 The CFG before is translated as below by this translator:
Program and Database:
```
    parsing(G,X,Z) :- dictionary(C,X,Y), parsed(C,G,Y,Z).
    parsed(np,G,X,Z) :- parsing(vp,X,Y), parsed(s,G,Y,Z). parsed(G,G,X,X).
    dictionary(C,[W|X],X) :- dictionary(W,C).
    dictionary(john,np).
    dictionary(walks,vp).
```
Query:
```
    ?- parsing(s,[john,walks],[ ]).
```

(5) Relations to Other Works

Our "Alexander Parser" is an adaptation of Seki's query evaluation method in deductive databases, called "Alexander Templates" [7],[8], to parsing algorithms. In fact, if we
- apply his algorithm to the logic programs generated by the translator of Section 4 (1), considering "*dictionary*" as an extensional database predicate, and
- identify the predicates "*call_parse*," "*sol_parse*" and "*cont*" predicates with the predicates "*parsing*," "*parsed*" and "*table*,"

then the result is exactly our "Alexander Parser."

Our "Alexander Parser" is also a refinement of Fuchi's description of Earley's parsing algorithm [2] within the framework of logic programming [3]. The predicate D and the "guide predicate" G in his description play the same role as the predicates "*parsed*" and "*parsing*" in our "Alexander Parser." The predicate corresponding to our "*table*" is not explicit in his description, because the class of CFGs he considered is in Chomsky normal form. (A set of all the atoms of the form "$table([n,i],[l_0,l_i,l_m])$" with the same second argument in our "Alexander Parser" corresponds to a "parse table" of the original Earley's parsing algorithm.)

5. Conclusions

This paper has presented a hybrid parser using the bottom-up interpreter of logic programs, called "Alexander Parser." The implementation of the parser on the parallel machine is an interesting research theme left for future.

Acknowledgements

This research was done as a part of the Fifth Generation Computer Systems project of Japan. We would like to thank Dr. K. Fuchi (Director of ICOT) for the opportunity of doing this research, and Dr. K. Furukawa (Deputy Director of ICOT) and Dr. R. Hasegawa (Chief of ICOT 1st Laboratory) for their advice and encouragement. We would also thank Dr. Y. Matsumoto (Kyoto University) and Mr. H. Seki (Mitsubishi Electric Corporation) for their valuable suggestions.

References

[1] Aho,A.V., and Ullman,J.D., *The Theory of Parsing, Translation and Compiling, Volume 1: Parsing*, Prentice Hall, 1972.

[2] Earley, J., An Efficient Context-Free Parsing Algorithm, *Comm. of ACM*, 13, 2: 94–102 (1970).

[3] Fuchi,K., Predicate Logic Programming — A Proposal of EPILOG – , (in Japanese) Research Report of SIG Symbolic Processing, 1, 1, Japan Information Processing Society, July 1977. Also *J. of Japan Information Processing Society*, 26, 11: 1298–1305 (1985).

[4] Matsumoto,Y., Tanaka,H., Hirakawa,H., Miyoshi,H., and Yasukawa,H., BUP: A Bottom-Up Parser Embedded in Prolog, *New Generation Computing*, 1, 2: 145–158 (1983).

[5] Pereira,F.C.N., and Warren,D.H.D., Definite Clause Grammar for Language Analysis — A Survey of the Formalisim and a Comparison with Argumented Transition Networks —, *Artificial Intelligence*, 13: 231–278 (1988).

[6] Pereira,F.C.N., and Warren,D.H.D., Parsing as Deduction, *Proc. of 21st Annual Meeting of the Association for Computational Linguistics*, Boston, June 1983.

[7] Seki,H., On the Power of Alexander Templates, *Proc. of 8th ACM Symposium on Principles of Database Systems* :150–159, Philadelphia, March 1989.

[8] Seki,H., On the Power of Alexander Templates, ICOT Technical Report TR-5??, ICOT, Tokyo, December 1989.

Appendix. Proof of the Correctness of Alexander Parser

(1) Definitions for the Standard Notions of Parsing

As for the standard notions of parsing, we translate CFGs into logic programs in the same way as the standard DCG translator in Section 4 (1), and discuss the derivations in the original CFGs using the refutations in the corresponding logic programs. (We do not explain the detail, since it is almost obvious.)

In the following, s, t are used for terms, and X, Y for variables, possibly with primes and subscripts, to denote lists of symbols. A sequence of atoms of the form

$parse(c_1, t_0, t_1), parse(c_2, t_1, t_2), \ldots, parse(c_n, t_{n-1}, t_n)$

is called a *goal*, where $n \geq 0$, and each c_i is a non-terminal symbol ($1 \leq i \leq n$). Hereafter, P and D denote a set of grammar rules and a set of dictionary rules, respectively.

Definition Search Tree

A tree is called a *search tree* when each node is labelled with a goal. A *search tree of atom A* is a search tree whose root node is labelled with a goal consisting of only one atom A. When a node in a search tree is labelled with "$A_1, A_2, \ldots, A_n$," the atom A_1 is called the *head atom of* the node.

Definition Resolution

A terminal node of search tree T labelled with

$parse(c, t_0, t_1), parse(c_2, t_1, t_2), \ldots, parse(c_n, t_{n-1}, t_n)$

is said to be *resolvable* in $P \cup D$ when it satisfies either of the following conditions:

- There is some grammar rule "$c \rightarrow d_1\ d_2\ \cdots\ d_m$" in P. The goal
  $parse(d_1, t_0, X_1), parse(d_2, X_1, X_2), \ldots, parse(d_m, X_{m-1}, t_1),$
  $parse(c_2, t_1, t_2), \ldots, parse(c_n, t_{n-1}, t_n)$
  is called the *resolvent*.
- There is some dictionary rule "$c \rightarrow w$" in D, and the head of t_0 is "w." The goal
  $(parse(c_2, t_1, t_2), \ldots, parse(c_n, t_{n-1}, t_n))\theta$
  is called the *resolvent*, where θ is an m.g.u. of t_0 and $[w|t_1]$.

Definition Initial Search Tree

The *initial search tree of* "$c \rightarrow w_1\ w_2\ \cdots\ w_k$" is a tree consisting of a single node labelled with goal "$parse(c, [w_1, w_2, \ldots, w_k], [\,])$."

Definition Extension of Search Tree

An immediate extension of search tree T in $P \cup D$ is the result of the following operations, when a terminal node of T is resolvable.

- Let $G_1, G_2, \ldots, G_k$ ($k > 0$) be all the resolvents of the node in $P \cup D$. Then, add k child nodes labelled with $G_1, G_2, \ldots, G_k$ to the node.

A search tree T_{ext} is an *extension of* search tree T in $P \cup D$ if T_{ext} is obtained from T through successive application of immediate extensions.

Definition Parsing Path and Partial Parsing Path

A path in search tree T is called a *parsing path of "$parse(c, t_0, t_1)$"* when it is a path starting with a node labelled with

$parse(c, t_0, t_1), G$

and followed by the nodes labelled with

$H_1\theta_1, G\theta_1,$

$H_2\theta_1\theta_2, G\theta_1\theta_2,$

$\vdots$

$H_{k-1}\theta_1\theta_2\cdots\theta_{k-1}, G\theta_1\theta_2\cdots\theta_{k-1},$

$G\theta_1\theta_2\cdots\theta_k,$

where $G, H_1, H_2, \ldots, H_{k-1}$ are sequences of atoms. Then, atom "$parse(c, t_0, t_1)\theta_1\theta_2\cdots\theta_k$" is called the *solution of the parsing path*. A path in a search tree T starting from a node with head atom "$parse(c, t_0, t_1)$" is called a *partial parsing path of "$parse(c, t_0, t_1)$"* when it does not contain any parsing path of "$parse(c, t_0, t_1)$" as its prefix. The *length of a (partial) parsing path* is the number of nodes contained in the path.

(2) Definitions for Alexander Parser

As for the notions of our parsing, some of the following definitions overlap with the contents of Section 3. We have repeated them for making the correspondence between the standard notions of parsing and the notions of our parsing clear. Hereafter, $\bar{P}$ and $\bar{D}$ denote the result of "$translate0(P, D)$."

Definition Atom Set

A set of atoms is called an *atom set* when it consists of atoms of the form "$parsing(c, t, t')$," "$parsed(c, t, t')$," "$dictionary(w, c)$."

Definition Generated Atom

Let $\bar{P}$ be a program, Γ be an atom set, $\bar{C}$ be a clause in $\bar{P}$, and θ be a substitution for the variables appearing in the body atoms of $\bar{C}$. Then, atom $A\theta$ is said to be *generated from* Γ *using* $\bar{C}$ when

- atom A is the head atom of $\bar{C}$, and
- Γ contains instances of the body atoms of $\bar{C}$ by θ.

Definition Initial Atom Set

The *initial atom set of* "$\{parsing(c, [w_1, w_2, \ldots, w_k], [\ ])\}$" is the set of atoms $\{parsing(c, [w_1, w_2, \ldots, w_k], [\ ])\} \cup \bar{D}$.

Definition Extension of Atom Set

An *immediate extension* of atom set Γ in $\bar{P} \cup \bar{D}$ is

$\Gamma' \cup \Gamma_0,$

where Γ' is the set of all the atoms generated from Γ using some clause in $\bar{P}$, and Γ_0 is an initial atom set. An atom set Γ_{ext} is an *extension of* atom set Γ in $\bar{P} \cup \bar{D}$ if Γ_{ext} is obtained from Γ through successive application of immediate extensions.

(3) Proof of the Correctness

The following Lemma 1 reduces the correctness of the refined "Alexander Parser" to that of the naive "Alexander Parser," which is in turn guaranteed by the following Lemma 2 and Lemma 3. Let P be a set of grammar rules, D be a set of dictionary rules, I be an input rule, $(\bar{P}, \bar{D})$ be the result of "$translate0(P, D)$," and $\bar{I}$ be the result of "$translate\text{-}input(I)$."

Lemma 1 Let Γ_∞ and Δ_∞ be the set of all the atoms of the form "$parsing(c, t, t')$," "$parsed(c, t, t')$," "$dictionary(w, c)$" generated by the naive "Alexander Parser" and the refined "Alexander Parser," respectively. Then, Γ_∞ and Δ_∞ are identical.

Proof. Obvious by induction on the number of the steps required to generate the atoms.

Lemma 2

Let T be an extension of an initial search tree, and Γ be an extension of an initial atom set.

(a) If T contains a partial parsing path of "$\boldsymbol{parse}(c,t_0,t_1)$" whose last node has head atom "$parse(d,s_0,s_1)$," and Γ contains "$\boldsymbol{parsing}(c,t_0,t_1)$," then some extension of Γ contains "$parsing(d,s_0,s_1)$."

(b) If T contains a parsing path of "$\boldsymbol{parse}(c,t_0,t_1)$" with solution "$\boldsymbol{parse}(c,t_0,t_1)\theta$," and Γ contains "$\boldsymbol{parsing}(c,t_0,t_1)$," then some extension of Γ contains "$\boldsymbol{parsed}(c,t_0,t_1)\theta$."

Proof. The proof is by simultaneous induction on the length of (partial) parsing paths.

Proof of Part (a):

Let r be a partial parsing path starting from node u and ending with node v.

Base Case: If the length of r is 1, then "$\boldsymbol{parsing}(d,s_0,s_1)$" is identical to "$\boldsymbol{parsing}(c,t_0,t_1)$," hence, from the assumption, "$\boldsymbol{parsing}(d,s_0,s_1)$" is in Γ.

Induction Step: If the length of r is greater than 1, there exists a grammar rule, say of the form

"$c_0 \rightarrow c_1\ c_2\ \cdots\ c_m$"

with which u is resolvable. Let u_0 be the immediate child node of u labelled with resolvent

$parse(c_1,t_0,Y_1), parse(c_2,Y_1,Y_2),\ldots,parse(c_m,Y_{m-1},t_1),\ldots$

Let the path from u_0 to v be divided into

r_1 : parsing path of "$parse(c_1,t_0,Y_1)$" with solution "$parse(c_1,t_0,Y_1)\theta_1$,"

r_2 : parsing path of "$parse(c_2,Y_1,Y_2)\theta_1$" with solution "$parse(c_2,Y_1,Y_2)\theta_1\theta_2$,"

$\vdots$

r_i : parsing path of "$parse(c_i,Y_{i-1},Y_i)\theta_1\theta_2\cdots\theta_{i-1}$"
with solution "$parse(c_i,Y_{i-1},Y_i)\theta_1\theta_2\cdots\theta_i$,"

r_{i+1} : partial parsing path "$parse(c_{i+1},Y_i,Y_{i+1})\theta_1\theta_2\cdots\theta_i$" with length shorter than r.

Because a clause

$parsing(c_1,X_0,X_1)$:- $parsing(c_0,X_0,X_m)$

is in $\bar{P}$, atom "$parsing(c_1,t_0,Y_1)$" is in the immediate extension of Γ. From the induction hypothesis for part (b), atom "$\boldsymbol{parsed}(c_1,t_0,Y_1)\theta_1$" is in some extension of Γ. Similarly, atoms

$parsing(c_2,Y_1,Y_2)\theta_1$,
$parsed(c_2,Y_1,Y_2)\theta_1\theta_2$,

$\vdots$

$parsing(c_i,Y_{i-1},Y_i)\theta_1\theta_2\cdots\theta_{i-1}$,
$parsed(c_i,Y_{i-1},Y_i)\theta_1\theta_2\cdots\theta_i$,
$parsing(c_{i+1},Y_i,Y_{i+1})\theta_1\theta_2\cdots\theta_i$

are in some extension of Γ. Then, from the induction hypothesis for part (a), atom "$parsing(d,s_0,s_1)$" is in some extension of Γ.

Proof of Part (b):

Let r be a parsing path starting from node u and ending with node v.

Base Case: If the length of r is 1, there exists a dictionary rule, say of the form

"$c \rightarrow w$"

with which u is resolvable. Becaues a clause

$parsed(C,[W|X],X)$:- $parsing(C,[W|X],X), \boldsymbol{dictionary}(W,C)$

is in $\bar{P}$, and atom "$\boldsymbol{dictionary}(w,c)$" is in $\bar{D}$, (hence in Γ,) atom "$\boldsymbol{parsed}(c,[w|t],t)$" is in the immediate extension of Γ.

Induction Step: If the length of r is greater than 1, there exists a grammar rule, say of the form

"$c_0 \rightarrow c_1\ c_2\ \cdots\ c_m$"

with which u is resolvable. Let u_0 be the immediate child node of u labelled with resolvent

$parse(c_1, t_0, Y_1), parse(c_2, Y_1, Y_2), \ldots, parse(c_m, Y_{m-1}, t_1), \ldots$

Let the path from u_0 to v be divided into

r_1 : parsing path of "$parse(c_1, t_0, Y_1)$" with solution "$parse(c_1, t_0, Y_1)\theta_1$,"

r_2 : parsing path of "$parse(c_2, Y_1, Y_2)\theta_1$" with solution "$parse(c_2, Y_1, Y_2)\theta_1\theta_2$,"

$\vdots$

r_m : parsing path of "$parse(c_m, Y_{m-1}, Y_m)\theta_1\theta_2\cdots\theta_{m-1}$"
with solution "$parse(c_m, Y_{m-1}, Y_m)\theta_1\theta_2\cdots\theta_m$."

Because a clause

$parsing(c_1, X_0, X_1)$:- $parsing(c_0, X_0, X_m)$

is in $\tilde{P}$, atom "$parsing(c_1, t_0, Y_1)$" is in the immediate extension of Γ. From the induction hypothesis for part (b), atom "$parsed(c_1, t_0, Y_1)\theta_1$" is in some extension of Γ. Similarly, atoms

$parsing(c_2, Y_1, Y_2)\theta_1$,
$parsed(c_2, Y_1, Y_2)\theta_1\theta_2$,

$\vdots$

$parsing(c_m, Y_{m-1}, Y_m)\theta_1\theta_2\cdots\theta_{m-1}$,
$parsed(c_m, Y_{m-1}, Y_m)\theta_1\theta_2\cdots\theta_m$

are in some extension of Γ. Then, because a clause

$parsed(c_0, X_0, X_m)$:-
$parsing(c_0, X_0, X_m), parsed(c_1, X_0, X_1), \ldots, parsed(c_m, X_{m-1}, X_m)$

is in $\tilde{P}$, atom "$parsed(c, t_0, t_1)\theta_1\theta_2\cdots\theta_m$," i.e., "$parsed(c, t_0, t_1)\theta$" is in some extension of Γ.

Lemma 3

Let Γ be an extension of the initial atom set of $\tilde{I}$, and T be an extension of the initial search tree of I.

(a) If Γ contains "$parsing(c_i, t_{i-1}, t_i)$," then some extension of T contains a node with head atom "$parse(c_i, t_{i-1}, t_i)$."

(b) If Γ contains "$parsed(c_0, t_0, t_m)$," then some extension of T contains a parsing path with solution "$parse(c_0, t_0, t_m)$."

Proof. The proof is by simultaneous induction on the number of steps required to generate the atoms in Γ from the initial atom set.

Proof of Part (a):

Base Case: If the number of required steps is 0, "$parsing(c_i, t_{i-1}, t_i)$" is in the initial atom set, hence "$parse(c_i, t_{i-1}, t_i)$" is the leftmost atom of the root node of T.

Induction Step: If the number of required steps is more than 0, there exists an instance of a clause in $\tilde{P}$ of the form

$parsing(c_i, t_{i-1}, t_i)$:- $parsing(c_0, t_0, t_m), parsed(c_1, t_0, t_1), \ldots, parsed(c_{i-1}, t_{i-2}, t_{i-1})$

translated from a grammar rule

"$c_0 \rightarrow c_1\ c_2\ \cdots\ c_m$,"

where $parsing(c_0, t_0, t_m), parsed(c_1, t_0, t_1), \ldots, parsed(c_{i-1}, t_{i-2}, t_{i-1})$ are already in Γ at the previous step ($i > 0$). From the induction hypothesis for part (a), there exists an extension of T such that it contains a node whose leftmost atom is "$parse(c_0, t_0, t_m)$," and the grammar rule above is resolvable with the node. Similarly, from the induction hypothesis for part (b), there exists a further extension of T such that it contains a path starting with the node followed by the parsing paths with solutions "$parse(c_1, t_0, t_1)$," ..., "$parse(c_{i-1}, t_{i-2}, t_{i-1})$." Then, the leftmost atom of the last node of the path is "$parse(c_i, t_{i-1}, t_i)$."

Proof of Part (b):

Base Case: If the number of required steps is 0, it must be in the initial atom set, which obviously does not hold, since any initial atom set contains atoms with predicates either "*parsing*" or "*dictionary*." Hence, the lemma is vacantly true.

Induction Step: If the number of required steps is greater than 0, there are the following two cases:

Case 1: There exists an instance of a clause in $\hat{P}$ of the form

$parsed(c_0, t_0, t_m)$:- $parsing(c_0, t_0, t_m), parsed(c_1, t_0, t_1), \ldots, parsed(c_m, t_{m-1}, t_m)$

translated from a grammar rule

"$c_0 \rightarrow c_1\ c_2\ \cdots\ c_m$,"

where $parsing(c_0, t_0, t_m), parsed(c_1, t_0, t_1), \ldots, parsed(c_m, t_{m-1}, t_m)$ are already in Γ at the previous step ($m > 0$). From the induction hypothesis for part (a), there exists an extension of T such that it contains a node whose leftmost atom is "$parse(c_0, t_0, t_m)$." and the grammar rule above is resolvable with the node. Similarly, from the induction hypothesis for part (b), there exists a further extension of T such that it contains a path starting with the immediate child node followed by the parsing paths with solutions "$parse(c_1, t_0, t_1)$," ..., "$parse(c_m, t_{m-1}, t_m)$." Then, the whole path is a parsing path with solution "$parse(c_0, t_0, t_m)$."

Case 2: There exists an instance of a clause in $\hat{P}$ of the form

$parsed(c_0, [w|t_m], t_m)$:- $parsing(c_0, [w|t_m], t_m), dictionary(w, c_0)$

translated from a dictionary rule

"$c_0 \rightarrow w$,"

where t_0 is $[w|t_m]$, and $parsing(c_0, t_0, t_m), dictionary(w, c_0)$ are already in Γ at the previous step. Hence, from the induction hypothesis for part (a), there exists an extension of T such that it contains a node whose leftmost atom is "$parse(c_0, t_0, t_m)$." Then, the immediate extension of the extension of T contains a parsing path with solution "$parse(c_0, t_0, t_m)$."