# Towards Machine Learning of Grammars and Compilers of Programming Languages

Keita Imada and Katsuhiko Nakamura

School of Science and Engineering,
Tokyo Denki University, Hatoyama-machi, Saitama-ken,
350-0394 Japan
nakamura@k.dendai.ac.jp

**Abstract.** This paper discusses machine learning of grammars and compilers of programming languages from samples of translation from source programs into object codes. This work is an application of incremental learning of definite clause grammars (DCGs) and syntax directed translation schema (SDTS), which is implemented in the Synapse system. The main experimental result is that Synapse synthesized a set of SDTS rules for translating extended arithmetic expressions with function calls and assignment operators into object codes from positive and negative samples of the translation. The object language is a simple intermediate language based on inverse Polish notation. These rules contain an unambiguous context free grammar for the extended arithmetic expressions, which specifies the precedence and associativity of the operators. This approach can be used for designing and implementing a new programming language by giving the syntax and semantics in the form of the samples of the translation.

## 1 Introduction

This paper discusses machine learning of grammars and compilers of programming languages from positive and negative samples of translation from source programs into object codes. This work is an application of incremental learning of definite clause grammars (DCGs) [16] and syntax-directed translation schema (SDTS) [1]. The grammatical induction of these extended context free grammars (CFGs) is implemented in the *Synapse* system [13,14].

The DCG can be converted to a logic program for parsing and generating strings in the language of the grammar. The DCG is more powerful than the CFG, as the DCG rules can have additional parameters for controlling derivations and for communicating parameters between separate nodes in the derivation tree.

This paper shows our experimental results of incremental learning of DCG rules representing a restricted form of SDTS, which specify the translation by compilers. We intend to apply this approach to design and implement a new programming language by giving the syntax and semantics in the form of the samples of the translation.

W. Daelemans et al. (Eds.): ECML PKDD 2008, Part II, LNAI 5212, pp. 98–112, 2008.

### 1.1 Grammatical Induction of Extended CFG

Our approach to grammatical induction is characterized by rule generation based on bottom-up parsing for positive sample strings, the search for rule sets and incremental learning. In Synapse, the process called *bridging* generates the production rules that bridge, or make up, any lacking parts of an incomplete derivation tree that is the result of bottom-up parsing of a positive string. The system searches for a set of rules that satisfies all the samples by *global search*, in which the system searches for the minimal set of rules that satisfies given sets of positive and negative samples by iterative deepening.

Incremental learning is essential to this approach. In order to learn a grammar from its sample strings, the positive samples are given to the rule generation process in the order of their lengths. This process continues until the system finds a set of rules that derives all the positive samples, but none of negative samples. By incremental learning, a grammar can be synthesized by adding rules to previously learned grammars of either similar languages or a subset of the target language. This is a method to solve the fundamental problem of computational complexity in learning CFG and more complex grammars [7,18].

An important feature of Synapse for the subject of this paper is that the system synthesizes minimal or semi-minimal grammars based on a covering-based approach. Many other grammatical inference systems for CFGs and their extensions are classified into the generalization based approach, in which the systems generate rules by analyzing the samples and generalizing and abstracting the rules. This is a reason that there have been few publications on learning small rule sets of both ambiguous and unambiguous CFGs and/or extended CFGs. Learning SDTS was chosen as the subject of the *Tenjinno* competition [20] at ICGI 2006. Clark [2] solved some of the problems of the competition. Most participants to this competition, as well as the *Omphalos* competition [19] at ICGI 2004 about learning CFG, did not intend to synthesize small grammars.

Learning DCG is closely related to inductive logic programming (ILP) [15]. Several studies dealt with learning grammars based on ILP [3,10]. Cussens and Pulman [4] describes a method of learning missing rules using bottom-up ILP. There have been, however, few publications focusing on learning DCG and its applications based on ILP. Fredouille et. al. [6] describes a method for efficient ILP-based learning of DCGs for biological pattern recognition. Ross [17] presents learning of Definite Clause Translation Grammar, a logical version of attribute grammar, by genetic programming.

### 1.2 Learning Grammars and Compilers

Processing of programming languages has been a key technology in computer science. There has been much work in compiler theory, including reducing the cost of implementing programming languages and optimizing object codes in compilers. On the other hand, there have been few works on applying machine learning of grammars and compilers. Dubey et. al. [5] describes a method of inferring grammar rules of a programming language dialect based on the grammar of the

original language for restoring the lost knowledge of the dialect. Other applications of machine learning to compilers are related to optimizing object codes. Monsifrot et. al. [11] and Stephenson et. al. [21] showed methods of improving optimization heuristics in compilers by using machine learning technologies, decision trees and genetic programming, respectively.

The grammars of programming languages need to not only be unambiguous, but also reflect its semantics, i.e. how a program is computed. For example, any grammar for arithmetic expressions should specify the precedence and left, or right, associativity of the operators. Therefore, synthesizing grammars requires some structural or semantic information other than positive and negative samples of source codes. To address this requirement, in our approach the samples for the learning are pairs of source codes and corresponding object codes in an intermediate language.

### 1.3 Organization of the Paper

Section 2 outlines CFG and SDTS, and defines Chomsky normal form of SDTS. Section 3 briefly defines some basics of logic programming and DCG, and shows a method of representing SDTS by DCG. Section 3 shows the bridging rule generation procedure for both CFGs and DCGs. Section 4 describes search strategies for finding minimal or semi-minimal rule sets, and shows recent experimental results of learning CFGs and DCGs. Section 6 describes learning DCG rules that represent SDTS rules for translating extended arithmetic expressions into object codes in a simple intermediate language, which is based on inverse Polish notation. Section 7 presents the concluding remarks.

## 2 CFGs and SDTS

A *context free grammar* (CFG) is a system $(N, T, P, s)$, where: $N$ and $T$ are finite sets of nonterminal and terminal symbols, respectively; $P$ is the set of (production) rules of the form $p \to u,\ p \in N, u \in (N \cup T)^+$; and $s \in N$ is the starting symbol. We write $w \Rightarrow_G x$ for $w, x \in (N \cup T)^+$, if there are a rule $(p \to u) \in P$ and strings $z_1, z_2 \in (N \cup T)^*$ such that $w = z_1 p\, z_2$ and $x = z_1 u\, z_2$. The *language* of $G$ is the set $L(G) = \{w \in T^+ \mid s \Rightarrow_G^* w\}$, where the relation $\Rightarrow_G^*$ is the reflexive transitive closure of $\Rightarrow_G$. Nonterminal symbols are represented by $p, q, r, s, t$, terminal symbols by $a, b, c, d, e$ and either nonterminal or terminal symbols by $\beta, \gamma$.

*Chomsky normal form* (CNF) rules are of the forms $p \to a$ and $p \to qr$. Synapse synthesizes rules of the *extended CNF*, $p \to \beta$ and $p \to \beta\gamma$. A feature of this normal form is that grammars can be made simpler than those in CNF.

A *syntax-directed translation schema* (SDTS) is a system $T = (N, \Sigma, \Delta, R, s)$, where: $\Sigma$ and $\Delta$ are sets of input terminal and output terminal symbols, respectively; and $R$ is a set of rules of the form

$$p \to u, v. \quad p \in N, u \in (N \cup \Sigma)^+, v \in (N \cup \Delta)^+,$$

such that the set of nonterminal symbols that occur in $u$ is equal to that in $v$. If a nonterminal symbol $p$ appears more than once in $u$, we use symbols with the subscripts $p_1, p_2$ to indicate the correspondence of the symbols in $u$ and $v$. We write $w_1, w_2 \Rightarrow_T x_1, x_2$ for $w_1, x_1 \in (N \cup \Sigma)^*$ and $w_2, x_2 \in (N \cup \Delta)^*$, if there is a rule $(p \rightarrow u, v) \in R$ and strings $y_1, y_2 \in (N \cup \Sigma)^*$ and $z_1, z_2 \in (N \cup \Delta)^*$ such that $w_1 = y_1\, p\, y_2, x_1 = y_1 u\, y_2, w_2 = z_1\, p\, z_2$ and $x_2 = z_1\, v\, z_2$. The SDTS $T$ *translates* a string $w \in \Sigma^+$, into $x \in \Delta^+$ and vice versa, if $(s, s) \Rightarrow_T^* (w, x)$.

The SDTS is *regular*, if and only if every rule is of the form of either $p \rightarrow a\, q, b\, r$ or $p \rightarrow a, b$. The regular SDTS is equivalent to a finite sequential transducer.

We restrict the form of SDTS to *Chomsky normal form* (CNF), in which every string in the right hand side of the rule has at most two symbols, e.g. $(p \rightarrow p\, q, p\, q)$, $(p \rightarrow p\, q, q\, p)$ or $(p \rightarrow a, b)$. Any SDTS $T$ with rule(s) having three symbols in the right hand side can be transformed into a SDTS in CNF, which is equivalent to $T$. The SDTS in this paper is extended so that elements of the strings may be not only constants but also lists of the constants. There are, however, SDTS rules with more than three symbols in the right hand side, that cannot be simply transformed to CNF.

*Example 1: reversing strings* The following SDTS in CNF translates any string into its reversal, for example, *aababb* to *bbabaa*.

$$s \rightarrow a, a \quad s \rightarrow b, b \quad s \rightarrow a\, s, s\, a \quad s \rightarrow b\, s, s\, b$$

This SDTS derives the set of pairs of strings,

$$(a, a), (b, b), (aa, aa), (ab, ba), (ba, ab), (bb, bb), (aaa, aaa), (aab, baa), \cdots .$$

## 3 Definite Clause Grammars (DCG)

We use the notations and syntax of standard Prolog for constants, variables, terms, lists and operators, except that the constants are called atoms in Prolog. A constant (an atom in Prolog) is an identifier that starts with a lower-case character, and a variable starts with an upper-case character, as in standard Prolog. A *subterm* of a term $T$ is either $T$, an argument of a complex term $T$ or recursively a subterm of an argument of a complex term $T$. As any list is a special term, subterms of $[a, b, c]$ are $[a, b, c], a, [b, c], b, [c], c, [\,]$.

A *substitution* $\theta$ is a mapping from a set of variables into a set of terms. For any term $t$, an *instance* $t\theta$ is a term in which each variable $X$ defined in $\theta$ is replaced by its value $\theta(X)$. For any terms $s$ and $t$, we write $s \succeq t$, and say that $s$ is *more general* than $t$, if and only if $t$ is an instance of $s$. A *unifier* for two terms $s$ and $t$ is a substitution $\theta$, such that $s\theta = t\theta$. The unifier $\theta$ is the *most general unifier* (mgu), if there is another unifier $\sigma$ for $s$ and $t$, then $s\theta \succeq s\sigma$ and $t\theta \succeq t\sigma$. We write $s \equiv_\theta t$, if $s$ is unifiable with $t$ by an mgu $\theta$. For any terms $s$ and $t$, a term $u$ is the *lgg: least general generalization*, if and only if $u \succeq s$, $u \succeq t$ and there is no other term $v$ such that $v \succeq s$, $v \succeq t$ and $u \succeq v$.

A *DCG rule*, also called *grammar rule* in the draft of the ISO standard [12], is of the form $P$ `-->` $Q_1, Q_2, \cdots, Q_m$, where:

- $P$ is a nonterminal term, which is either a symbol or a complex term of the form $p(T)$ with a *DCG term* $T$; and
- each of $Q_1, \cdots, Q_m$ is either a constant of the form `[`$a$`]` representing a terminal symbol, or a nonterminal term.

The DCG terms are additional arguments in the Horn clause rules, which are generally used for controlling the derivation and for returning results of the derivation. The deduction by DCG is similar to that of CFG, except that each of the DCG terms is unified with a corresponding term in the deduction. To simplify the synthesis process, we restrict every atom for the nonterminal symbol to have exactly one DCG term.

Most Prolog implementations have a functionality to transform the grammar rules into Horn clauses, such that a string $a_1 a_2 \cdots a_n$ is derived by the rule set from the starting symbol $s$, if and only if the query $s($`[`$a_1, a_2, \cdots, a_n$`]`$,[])$ succeeds for the parsing program composed of the transformed clauses. Note that the two arguments are used for representing strings by the difference lists.

*Example 2: A DCG for non-context-free language* The following set of rules is a DCG for the language $\{a^n b^n c^n \mid n \geq 1\}$.

```
p(1) --> [a].          p(t(N)) --> [a],p(N).
q(1) --> [b].          q(t(N)) --> [b],q(N).
r(1) --> [c].          r(t(N)) --> [c],r(N).
s(N) --> p(N),q(N),r(N).
```

The Horn clauses transformed from these DCG rules are:

```
p(1,[a|X],X).          p(t(N),[a|X],Y) :- p(N,X,Y).
q(1,[b|X],X).          q(t(N),[b|X],Y) :- q(N,X,Y).
r(1,[c|X],X).          r(t(N),[c|X],X) :- r(N,X,Y).
s(N,X0,X3) :- p(N,X0,X1),q(N,X1,X2),r(N,X2,X3).
```

For the query `?-s(N,[a,a,a,b,b,b,c,c,c],[])`, the Prolog system returns the computation result `N = t(t(1))`.

We can transform a SDTS in CNF into a DCG for translating strings by the relations in Table 1. Each pair of the form $X/Y$ represents an output side of a string by a difference list. By this method, the problem of learning SDTS in CNF from pairs of input and output strings can be transformed into that of learning DCG.

*Example 3: Reversal* The following DCG corresponds to the example SDTS for reversing strings in Section 2.

```
s([a|X]/X) --> [a].              s([b|X]/X) --> [b].
s([a|X]/[]) --> s(X/[]),[a].     s([b|X]/[]) --> s(X/[]),[b].
```

For a query `?-s(X,[b,b,a,b,a,a],[])`, the transformed Prolog program returns the value `X = [a,a,b,a,b,b]/[]` of the DCG term, and for a query `?-s([a,a,b,a,b,b]/[],X,[])`, the solution `X = [b,b,a,b,a,a]`.

**Table 1.** Relations between SDTS rules and DCG rules

| SDTS | DCG |
|---|---|
| $p \rightarrow q$ | $p(X/Y)$ --> $q$ |
| $p \rightarrow a, b$ | $p([b\|Y]/Y)$ --> $[a]$ |
| $p \rightarrow ar, ra$ | $p(X/Y)$ --> $[a], r(X/[a\|Y])$ |
| $p \rightarrow qr, qr$ | $p(X/Z)$ --> $q(X/Y), r(Y/Z)$ |
| $p \rightarrow qr, rq$ | $p(X/Z)$ --> $q(Y/Z), r(X/Y)$ |

## 4 Rule Generation Based on Bottom-Up Parsing

Fig. 1 shows the rule generation procedure, which receives a string $a_1 \cdots a_n$, the starting symbol with a DCG term $s(T)$, and a set of rules from the top-level search procedure from global variable $P$, and returns a set of DCG rules that derives the string from $s(T)$. This nondeterministic procedure is an extension of that for learning CFG in extended CNF [14]. The extension is related to adding DCG terms and generalization for the generated rules.

### 4.1 Rule Generation

The rule generation procedure includes a bottom-up parsing algorithm for parsing an input string $a_1 \cdots a_n$ using the rules in the set $P$. If the parsing does not succeed, the bridging process generates rules in extended CNF, which bridge any lacking parts of the incomplete derivation tree.

The input string $a_1 a_2 \cdots a_n$ is represented by a set of 3-tuples $\{(a_1, 0, 1), (a_2, 1, 2), \cdots, (a_n, n-1, n)\}$, and the resulting derivation tree by a set $D$ of 3-tuples of the form $(p(T), i, j)$, each of which represents that the set of rules derives $a_i \cdots a_j$ from $p(T)$. For the ambiguity check, each time a new term $(p(T), i, j)$ is generated, it is tested whether it has been generated before.

Subprocedure $Bridge(p(T), i, k)$ generates additional rules that bridge missing parts in the incomplete derivation tree represented by the set of terms in $D$. The process nondeterministically chooses six operations, as shown in Fig. 2. In operations 5 and 6, nonterminal symbols $q$ and $r$ are nondeterministically chosen from either previously used symbols or new symbols. The DCG term $U$ and $V$ of these symbols are also nondeterministically chosen from the subterms of the term $T$ of the parent node. Subprocedure $AddRule(R)$ first searches for a rule $R'$ in the set $P$ that has the same form as $R$ in the sense that $R$ and $R'$ differ only in the DCG terms. It then replaces $R'$ with the lgg of $R$ and $R'$, or simply adds $R$ to $P$.

Synapse has a special mode to efficiently generate DCG rules for SDTS. When this mode is selected, each DCG term is restricted to a difference list, and the rules are restricted to forms in Table 1. The system uses these restrictions for generating rules for SDTS in Operation 5 and 6 in Fig. 2 and in generalization in subprocedure *AddRule*.

**Procedure** *RuleGeneration*$(w, s(T_0), K)$ (Comment: $w$ : an input string, $s(T)$ : the starting symbol with a DCG term, $K$ : the bound for the number of rules. Global variable $P$ holds a set of rules.)
**Step 1** (Initialize variables.)
$D \leftarrow \emptyset$. ($D$ is a set of 3-tuples $(\beta(U), i, j)$.)
$k \leftarrow |P|$. ($k$ holds the initial number of rules in $P$.)
**Step 2:** (Parsing by inverse derivation)
For each $a_i, 1 \leq i \leq n = |w|$ in $w$, call $Derive(a_i, i-1, i)$ in order.
If $(s(T), 0, n) \in D$ and $T_0 \succeq T$ then terminate (Success).
**Step 3:** (Bridging rule generation)
If $|P| \geq K$ then terminate (Failure).
Call procedure $Bridge(s(T), 0, n)$.
Terminate (Success). (Return the set $P$ of rules).

**Procedure** *Derive*$(\beta, i, j)$ ($\beta$ : either a terminal a nonterminal term, $i, j$ : integers for representing the positions of a substring. )
1. Add $(\beta, j, k)$ to $D$. If $p(W) \rightarrow \beta' \in P$ such that $\beta \equiv_\theta \beta'$, then add $(p(W\theta), j, k)$ to $D$.
   To synthesize an unambiguous grammar, check ambiguity.
2. If $p(W) \rightarrow \alpha\beta \in P$ and $(\alpha', i, j) \in D$ with $\alpha \equiv_\theta \alpha'$ and $\beta \equiv_\theta \beta'$, then add $(p(W\theta), i, k)$ to $D$, and call $Derive(p(W\theta), i, j)$.

**Procedure** *Bridge*$(p(T), i, k)$ ($p(T)$ : a nonterminal term, $i, k$ : integers.)
Nondetermistically choose one of the operations in Fig. 2.

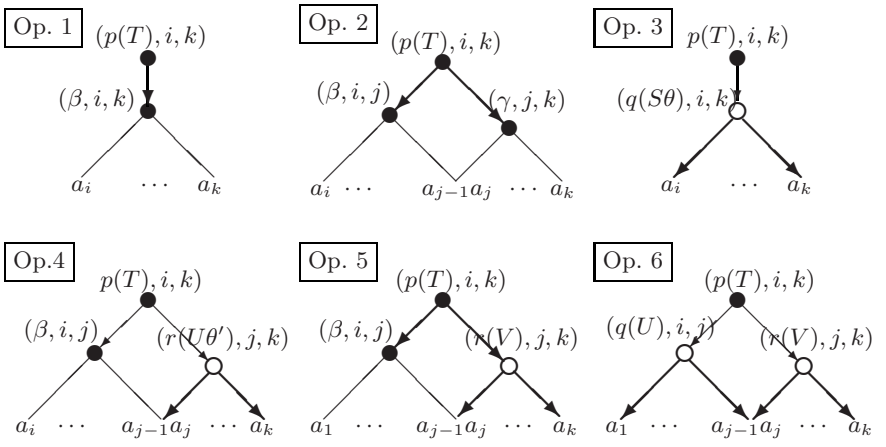
**Procedure** *AddRule*$(R)$ ($R$: a rule.)
Nondereminsitically choose one of the following process 1 or 2.
1. If $P$ contains a rule $R'$ such that $R$ differs from $R'$ only in DCG terms, delete $R'$ from $P$ and add the lgg of $R$ and $R'$ to $P$. Else add $R$ to $P$.
2. Add $R$ to $P$.

**Fig. 1.** Procedure for Rule Generation by Bridging

## 5 Search for Rule Sets

The inputs to Synapse are ordered sets $S_P$ and $S_N$ of positive and negative samples, respectively, and a set $P_0$ of initial rules for incremental learning of the grammars. Samples for learning a CFG are strings, whereas those for learning DCGs are pairs of strings and atoms of the form $s(T)$ with DCG terms $T$. The system searches for any set $P$ of rules with $P_0 \subseteq P$ such that all the strings in $S_P$ are derived from $P$ but no string in $S_N$ is derived from $P$. Synapse has two search strategies, global and serial search, for finding rule sets.

Fig. 3 shows the top-level procedure for the global search for finding minimal rule sets. The system controls the search by the iterative deepening on the number of rules to be generated. First, the number of initial rules is assigned to the bound $K$ of the number of rules. When the system fails to generate sufficient rules to parse the samples within this bound, it increases the bound by one and iterates the search. By this control, it is assured that the procedure finds a grammar with the minimal number of rules at the expense that the system repeats the same search each time the bound is increased.




**Op. 1** If $(\beta, i, k) \in D$, call $AddRule(p(T) \rightarrow \beta)$.
**Op. 2** If $(\beta, i, j) \in D$ and $(\gamma, j, k) \in D$, call $AddRule(p(T) \rightarrow \beta\gamma)$.
**Op. 3** If $p(T) \rightarrow q(S) \in P$with $T \equiv_\theta T'$, call $Bridge(q(S\theta), i, k)$.
**Op. 4** If $p(T') \rightarrow \beta\, r(U) \in P$ with $T \equiv_\theta T'$, and $(\beta\theta, i, j) \in D$ with $\beta \equiv_{\theta'} \beta'$, call $Bridge(r(U\theta'), i, k)$.
**Op. 5** If $(\beta, i, j) \in D$, call $AddRule(p(T) \rightarrow \beta\, r(U))$ and call $Bridge(r(U), j, k)$.
**Op. 6** For each $j, i + 2 \leq j \leq k - 2$, call $AddRule(p(T) \rightarrow q(U)\, r(V))$ and call $Bridge(q(U), i, j)$ and call $Bridge(r(U), j, k)$.

- In Operations 5 and 6, each of $U$ and $V$ is a nondeterminstically chosen subterm of $T$, and the nonterminal symbols $q$ and $r$ are nondeterminstically chosen among the symbols that occurs in $P$ and a newly generated symbol.
- This figure only shows one side of the symmetrical operations 4 and 5.

**Fig. 2.** Rule Generation Operations

### 5.1 Serial Search and Search for Semi-minimal Rule Sets

In the serial search, the system generates additional rules for each positive sample by iterative deepening. After the system finds a rule set satisfying a positive sample and no negative samples, the process does not backtrack to redo the search on the previous samples. By this search strategy, the system generally finds semi-minimal rule sets in shorter computation time. Other methods for finding semi-minimal rule sets include using non-minimal nonterminal symbols in rule generation and restricting the form of the generated rules. These methods generally increase the efficiency of searching for rule sets at the cost that the rule sets found may not be minimal.

### 5.2 Experimental Results on Learning CFG and DCG

The experimental results in this paper were obtained with Synapse Version 4, written in Prolog, using a Xeon processor with a 3.6 GHz clock and SWI-Prolog for Linux.

**Procedure** *GlobalSearch*$(S_P, S_N, P_0)$ ($S_P$ and $S_N$ are ordered sets of positive and negative samples, respectively, each of which is a pair of a string and the starting symbol with a DCG term. $P_0$ is a set of optional initial rules.)
**Step 1** (Initialize variables.)
$P \leftarrow P_0$ ($P$ is a global variable holding the set of rules).
$K \leftarrow |P_0|$ (the bound of the number of rules for iterative deepening).
**Step 2:** For each $(w, s(T)) \in S_P$, iterate the following operations 1 and 2.
1. Call *RuleGeneration*$(w, K)$.
2. For each $(v, p(U)) \in S_N$, test whether $P$ derives $v$ from $p(U)$ by the parsing algorithm. If there is a string $v$ derived from $P$, then terminate (Failure).

If no set of rules is obtained, then add 1 to $K$ and iterate Step 2.
**Step 3:** Output the result $P$, and terminate (Success).
For finding multiple solutions, backtrack to the previous choice point.

**Fig. 3.** Top-Level Procedure for Global Search

Synapse recently synthesized all the CFGs of the problems in Hopcroft and Ullman's textbook [8] from only the samples[1]. The problems include not only basic languages, such as parenthesis language and palindromes, but also non-trivial languages such as the set of strings containing twice as many $b$'s as $a$'s, strings not of the form $ww$, and the set $\{a^i b^j c^k \mid i = j \geq 1 \text{ or } j = k \geq 1\}$.

The experimental results show that the computation time by serial search is much faster than by global search at the expense of 1.0 to 3 times larger sizes of the rule sets in most cases. There are, however, rare cases where the learning by serial search does not converge for large volume of samples. We are currently working on solving this problem by introducing a more powerful search strategy. The restrictions to the form of the rules are also effective in speeding up the search, although the rules sets are slightly larger.

*Example 4: Reversal* The following pairs are some examples of positive samples for learning a DCG for reversal.

```
[a] - [a].      [b] - [b].          [a,b] - [b,a].
[b,a] - [a,b].  [a,a,b] - [b,a,a].  [a,a,b,a]- [a,b,a,a].
```

Given these samples and no negative samples, Synapse synthesized the following DCG rules after generating 12 rules in 0.13 sec.

```
s([a|X]/X) --> a.              s([b|X]/X) --> b.
s(X/Y) --> s(Z/Y), s(X/Z).
```

By the serial search, the same rule sets are found after generating 21 rules in less than 0.1 sec. This DCG of three rules is smaller than the DCG for reversing strings in Example 3 in Section 2.

Synapse synthesized the DCG rules for the language $\{a^n b^n c^n \mid n \geq 1\}$ in Example 2 in Section 3 after generating $6.5 \times 10^5$ rules in 3200 sec. by giving

[1] Detailed experimental results at the time 2006 are shown in [14].

three initial rules `p(1) --> [a]`, `q(1) --> [b]`, `r(1) --> [c]` in addition to the positive and negative samples.

## 6 Learning Translation in Simple Compiler

This section shows experimental results of learning translation of arithmetic expressions and assignment statements into an object language called *SIL* (Simple Intermediate Language). This language is similar to P-code and byte-code, which were originally designed for intermediate languages of Pascal and JAVA compilers, respectively. In these languages, object codes for arithmetic and logical expressions are based on the inverse Polish notation. The language SIL includes the following instructions.

- `load(T,V)` pushes the value of variable (symbolic address) `V` of type `T`, which is either `i` or `f` (float), to the top of the stack.
- `push(T,C)` pushes a constant `C` of type `T` to the top of the stack.
- `store(T,V)` stores the value at the top of the stack to variable (symbolic address) `V` of type `T`.
- Arithmetic operations: `fadd, fsubt, fmult, fdivide`. These float type operations are applied to two values on the stack and return the result at the top of the stack instead of the two values.
- `call(n,S)` calls function `S` with `n` parameters, which are placed at the top of the stack. The function returns the value at the top of the stack.

Note that although every instruction is typed, we deal with only object codes of float type in this paper.

### 6.1 Step One: Learning Translation of Arithmetic Expression

For reducing the computation time, we divide the learning of translation into two steps, and use incremental learning. In the first step, we gave Synapse positive and negative samples as shown in Fig. 4 and the initial rules in Fig. 5 (a). Each of the samples is a pair of an arithmetic expression and an object code in SIL. The negative samples containing symbol `X` in the right hand side represent restriction only on the source language. We assume that the rules for `a` and `b` are generated by processing constants, and those for `x` and `y` by declarations of the variables.

Synapse was made to search unambiguous DCGs with the starting symbol `s1`. The system synthesized eight DCG rules in Fig. 6 (a) after generating $3.4 \times 10^6$ rules in approximately 2000 sec by the global search. Among the positive samples, only the first nine samples were directly used for generating the grammar and the other positive samples are used for checking of the translation; the system parsed these samples without generating any additional rules. For the learning, 24 randomly chosen negative samples including Fig. 4 were sufficient.

**Positive samples**

```
a : push(f,a).                          ( a ) : push(f,a).
a + a : push(f,a),push(f,a),fadd.       a * a : push(f,a),push(f,a),fmult.
a / a : push(f,a),push(f,a),fdivide.    a - a : push(f,a),push(f,a),fsubt.
a + a + a : push(f,a),push(f,a),fadd,push(f,a),fadd.
a * a + a : push(f,a),push(f,a),fmult,push(f,a),fadd.
a / a + a : push(f,a),push(f,a),fdivide,push(f,a),fadd.
a - a + a : push(f,a),push(f,a),fsubt,push(f,a),push(f,a),fadd.
( a ) + a : push(f,a),push(f,a),fadd.
a + a * a : push(f,a),push(f,a),push(f,a),fmult,fadd.
a * a * a : push(f,a),push(f,a),fmult,push(f,a),fmult.
```

**Negative samples**

```
( a : X.          a ) : X.          ( + a : X.        ( * a : X.
+ a : X.          * a : X.          a + : X.          a * : X.
a + a : push(f,a),fadd,push(f,a).
a * a : push(f,a),fmult,push(f,a).
a + a + a : push(f,a),push(f,a),push(f,a),fadd,fadd.
a * a + a : push(f,a),push(f,a),push(f,a),fmult,fadd.
a + a * a : push(f,a),push(f,a),fadd,push(f,a),fmult.
```

**Fig. 4.** Samples for translating arithmetic expressions into the object codes in SIL for Step One

**(a)** Initial rules for Step One

```
n([push(f,a)|Y]/Y) --> a.        n([push(f,b)|Y]/Y) --> b.
n([load(f,x)|Y]/Y) --> x.        v([store(f,x)|Y]/Y) --> x.
n([load(f,y)|Y]/Y) --> y.        v([store(f,y)|Y]/Y) --> y.
op1([fadd|Y]/Y) --> +.           op1([fsubt|Y]/Y) --> -.
op2([fmult|Y]/Y) --> *.          op2([fdivid|Y]/Y) --> /.
lp(Y/Y) --> '('.                 rp(Y/Y) -->')'.
```

**(b)** Initial rules for Step Two

```
op3(Y/Y) --> =.                  s(X/Y) --> s1.
fn([call(1,sin)|Y]/Y) --> sin.   fn([call(1,cos)|Y]/Y) --> cos.
```

**Fig. 5.** Initial DCG rules for translating arithmetic expressions and assignment statements into object codes in SIL

### 6.2 Step Two: Learning Translation of Function Calls and Assignment Operator

In the second step, we gave Synapse the samples including those in Fig. 7, and made the system search for rules for translating function calls and assignment operator (=), based on the result of the first step. The starting symbol was set

**(a)** Rules for Arithmetic Expressions Synthesized in Step 1

```
s1(X/Y) --> e(X/Y).              s → e, e
s1(X/Z) --> s1(X/Y), f(Y/Z).     s → s f, f p
f(X/Z) --> op1(Y/Z), e(X/Y).     p → op1 e, e op1
e(X/Y) --> n(X/Y).               e → n, n
e(X/Z) --> e(X/Y), g(Y/Z).       e → e g, e g
g(X/Z) --> op2(Y/Z), n(X/Y).     r → op2 n, n op2
n(X/Z) --> lp(X/Y), p(Y/Z).      s → lp p, lp p
p(X/Z) --> s1(X/Y), rp(Y/Z).     p → s rp, s rp
```

**(b)** Rules for Function Calls and "=" Operator Synthesized in Step 2

```
n(X/Z) --> fn(Y/Z), q(X/Y).      n → fn q, q fn
q(X/Z) --> lp(X/Y), p(Y/Z).      n → lp p, lp p
s(X/Z) --> v(Y/Z), r(X/Y).       s → v r, r v
r(X/Z) --> op3(Y/Z), s(X/Y).     q → op3 s, op3 s
```

**Fig. 6.** Synthesized DCG rules and the corresponding SDTS rules for translating extended arithmetic expression into object codes in SIL

to `s`. The initial rules are composed of the synthesized rules in Step One in Fig. 6 (a) and all the initial rules in Fig. 5, which include `s(X/Y) --> s1`, (`s` → `s1` in SDTS).

By incremental learning, Synapse synthesized the four additional rules in Fig. 6 (b) after generating $1.8 \times 10^7$ rules in 4400 sec. Only the first four positive samples were directly used for generating the grammar and the other positive samples are used for checking of the translation. In Step two, 24 randomly chosen negative samples were also sufficient.

The synthesized DCG contains the CFG for the extended arithmetic expressions, which specifies that:

1. The precedence of the operators in `op2` (`*` and `/`) is higher than that of `op1` (`+` and `-`), which is higher than the assignment operator "`=`"; and
2. All arithmetic operators of `op1` and `op2` are left associative. The operator "`=`" is right associative. (Note that this associativity is coincident with that of C language. In many languages, the assignment operator is non-associative.)

The synthesized DCG rules can be converted to usual Prolog rules by adding two arguments for the difference lists representing the input strings. Since the DCG is left recursive, we needed to remove the left recursion by folding some of the clauses. The obtained program is executed as follows.

```
?- s(X/[],[a,*,b,*,b,*,'(',a,+,x,')'],[]).
X = [push(f,a),push(f,b),fmult,push(f,b),fmult,push(f,a),
    load(f,x),fadd,fmult]

?- s(X/[],[x,=,y,=,sin,'(',a,+,b,*,y,')'],[]).
```

**Positive samples**

```
x = a : push(f,a), store(f,x). sin ( a ) : push(a),call(1,sin). x
= a + b : push(f,a), push(f,b),fadd, store(f,x). x = y = a :
push(f,a), store(f,y),store(f,x). sin ( a + b ) :
push(f,a),push(f,b),fadd,call(1,sin). sin ( a ) + b :
push(f,a),call(1,sin),push(f,b),fadd. sin ( a ) * b :
push(f,a),call(1,sin),push(f,b),fmult. sin ( a ) * cos ( b ) :
push(f,a),call(1,sin), push(f,b),
                        call(1,cos),fmul.
sin ( cos ( b ) ) : push(f,b),call(1,cos),call(1,sin). sin ( a + b
) : push(f,a),push(f,b),fadd,call(1,sin).
```

**Negative samples**

```
sin a : X.        sin + a : X.      sin * a : X.     sin a ) : X.
sin ( a ) a : X.  sin ) : X.        sin sin : X.     sin + : X.
sin * : X.        = a a ) : X.      = a : X.         ( = a : X. (
a = a : X.      a = x : X.        = x ( a ) : X.   a = b b : X. x
= a + b : push(f,a),store(f,x),push(f,b),fadd.
```

**Fig. 7.** Samples for learning DCG for translating extended arithmetic expressions with function calls and assignment operators into the object codes in SIL

```
X = [push(f,a),push(f,b),load(f,y),fmult,fadd,call(1,sin),
    store(f,y),store(f,x)] ;
```

The computation is deterministic, and each query has only one solution.

## 7 Concluding Remarks

We showed an approach for machine learning of grammars and compilers of programming languages based on grammatical inference of DCG and SDTS, and showed the experimental results. The results of this paper are summarized as follows.

1. We extended the incremental learning of minimal, or semi-minimal, CFGs in the Synapse system to those of DCG and of SDTS.
2. Synapse synthesized a set of rules in SDTS for translating arithmetic expressions with function calls and assignment operators into object codes from samples of the translation. This set of SDTS rules can be used as a compiler in Prolog that outputs object codes in the intermediate language SIL.
3. The synthesized SDTS rules contain an unambiguous CFG for the extended arithmetic expressions, which specifies the precedence and associativity of the operators.

Although we showed learning of only a portion of compiling process of of the existing language, the learning system synthesized an essential part of the compiler from samples of translation. This approach can be used for produce the

grammar of a new language and at the same time implement the language from the samples of source programs and object codes.

We are currently working to improve the methods of learning DCGs and SDTSs, and extending the compiler to include type check and type conversion and to translate other statements and declarations. For type checking, the non-terminal symbols need to have additional parameters for the type. As the control instructions in the object codes generally have labels, we need a non-context-free language for the object codes. Other future subjects include:

- Theoretical analysis of learning DCG and SDTS.
- Clarifying the limitations of our methods in learning grammars and compilers of programming languages.
- Applying our approach for learning DCG to syntactic pattern recognition.
- Applying our approach for learning DCG to general ILP, and inversely applying methods in ILP to learning DCG and SDTS.

## Acknowledgements

The authors would like to thank Akira Oouchi and Tomihiro Yamada for their help in writing and testing the Synapse system. This work is partially supported by KAKENHI 19500131 and the Research Institute for Technology of Tokyo Denki University Q06J-06.

## References

1. Aho, A.V., Ullman, J.E.: The Theory of Parsing, Translation, and Compiling. Prentice-Hall, Englewood Cliffs (1972)
2. Clark, A.: Large Scale Inference of Deterministic Transductions: Tenjinno Problem 1. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 227–239. Springer, Heidelberg (2006)
3. Bratko, I.: Refining Complete Hypotheses in ILP. In: Džeroski, S., Flach, P.A. (eds.) ILP 1999. LNCS (LNAI), vol. 1634, pp. 44–59. Springer, Heidelberg (1999)
4. Cussens, J., Pulman, S.: Incorporating Linguistic Constraints into Inductive Logic Programming. In: Proc. CoNLL 2000 and LLL 2000, pp. 184–193 (2000)
5. Dubey, A., Jalote, P., Aggarwal, S.K.: Inferring Grammar Rules of Programming Language Dialects. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 201–213. Springer, Heidelberg (2006)
6. Fredouille, D.C., et al.: An ILP Refinement Operator for Biological Grammar Learning. In: Muggleton, S., Otero, R., Tamaddoni-Nezhad, A. (eds.) ILP 2006. LNCS (LNAI), vol. 4455, pp. 214–228. Springer, Heidelberg (2007)
7. de la Higuera, C., Oncina, J.: Learning Context-Free Languages, PASCAL-Pattern analysis, Static Model and Computational Learning, p. 28 (2004), `http://eprints.pascal-network.org/archive/00000061/`
8. Hopcroft, J.E., Ullman, J.E.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
9. Kowalski, R.: Logic for Problem Solving. North-Holland, Amsterdam (1979)

10. Muggleton, S.: Inverse Entailment and Progol. New generation Computing 13, 245–286 (1995)
11. Monsifrot, A., Boodin, F., Quiniou, R.: A Machine Learning Approach to Automatic Production of Compiler Heuristics. In: Scott, D. (ed.) AIMSA 2002. LNCS (LNAI), vol. 2443, pp. 41–50. Springer, Heidelberg (2002)
12. Moura, P.: Definite clause grammar rules, ISO/IEC DTR 13211-3 (2006)
13. Nakamura, K., Matsumoto, M.: Incremental Learning of Context Free Grammars Based on Bottom-up Parsing and Search. Pattern Recognition 38, 1384–1392 (2005)
14. Nakamura, K.: Incremental Learning of Context Free Grammars by Bridging Rule Generation and Semi-Optimal Rule Sets. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 72–83. Springer, Heidelberg (2006)
15. Nienhuys-Cheng, S.H., de Wolf, R.: Foundations of Inductive Logic Programming. Springer, Heidelberg (1997)
16. Pereira, F., Warren, D.H.D.: Definite Clause Grammars for Language Analysis: A Survey of the Formalism and a Comparison with Augmented Transition Networks. Jour. of Artificial Intelligence 13, 231–278 (1980)
17. Ross, B.J.: Logic-Based Genetic Programming with Definite Clause Translation Grammars. New generation Computing 19, 313–337 (2001)
18. Sakakibara, Y.: Recent advances of grammatical inference. Theoretical Computer Science 185, 15–45 (1997)
19. Starkie, B., Coste, F., van Zaanen, M.: The Omphalos Context-Free Grammar Learning Competition. In: Paliouras, G., Sakakibara, Y. (eds.) ICGI 2004. LNCS (LNAI), vol. 3264, pp. 16–27. Springer, Heidelberg (2004)
20. Starkie, B., van Zaanen, M., Estival, D.: The Tenjinno Machine Translation Competition. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 214–226. Springer, Heidelberg (2006)
21. Stephenson, M., Martin, M., O'Relly, U.M.: Meta Optimization: Improving Compiler Heuristics with Machine Learning. In: Proc. of PLDI 2003, San Diego, pp. 77–90 (2003)