Title	自己反映的構文解析系とコンパイルタイムリフレクションに関する研究
Author(s)	加藤, 大志朗
Citation	
Issue Date	2002-06
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/930
Rights	
Description	Supervisor:片山 卓也, 情報科学研究科, 博士

A Research on Reflective Parsing System and Compile-time Reflection

by

Daijiro KATO

Submitted to
Japan Advanced Institute of Science and Technology
In partial fulfillment of the requirements
For the degree of
Doctor of Philosophy

Supervisor: Professor Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology

June 2002

Abstract

The purpose of this paper is to construct a frame work of compiler-compiler for compile-time extensible language systems. This paper consists of three parts. Firstly, a self-extensible formal language system, named Reflective Context-Free Grammar (RCFG), is proposed. Secondly, an incremental construction method for LALR(1) parser is proposed. And additionally, we discuss on a frame work of descriptions of semantics for production rules which are newly introduced and embedded in program texts by users of compilers.

RCFG is one of self-extensible formal language systems, and an extension of Context-Free Grammar (CFG). The extensibility is obtained so as to embed new production rules which are desired to be used in the text following of the embedment. Typical point of RCFG is self-extensibility. To formalize the self-extensibility on the framework of CFG, we introduce a notion, named Augmented Forms (AF), and define derivation relation on AF sequences. In this paper, we establish some properties on RCFG, including the language class which is middle between Context-Free Language and Context-Sensitive Language, and the property that each word derived by given RCFG is also derived by a CFG which production rule set is identical to initial rule set of RCFG augmented with embedded production rules in the word. Especially, the latter property designates the characteristic of RCFG. Additionally, general parsing algorithm for RCFG, which is an extension of Earley's parsing algorithm for CFG, is given. Soundness and completeness of the algorithm are established, and also, the complexity of the algorithm and some restriction methods in order to accelerate parsing process are discussed.

In the second part of the paper, an incremental construction method for LALR(1) paser is discussed. In the method, both of LR(0) graphs and Look Ahead Symbol Sets are calculated in fully incremental manner, without use of any item set informations. Algorithms for the method are presented, and the worst case complexity is discussed. Adding to these, applications of the method for RCFG are discussed. To realize them, we introduce some notions. Firstly, we rearrange a method for incremental construction of LR(0) graphs so as not to use item set information. In conventional method for LR(0) parse table construction, some kind of subsets of items, called core or kernel, are used to identify states. Secondly, we introduce a notion for identification of states, named *MaxInc* which does not contain any information on item sets, instead of kernels. For incremental calculation of Look Ahead Symbol Set (LA), we introduce notions, named *Dependency Domain*, $E\Delta$ for ε-productivity judgment, and some other functions for calculation of (LA) with ε-productivity conditions. These notions essentially contribute to the incremental method proposed in this paper. We establish the equivalence between results obtained by conventional method and by our method.

In the additional part of the paper, we introduce a variation of Syntax Directed Translation (SDT) as a frame work for descriptions of semantics of new production rules which are introduced by users of compilers. As appendent arguments, we discuss on two kinds of implementation model of extended SDT from the point of view of development stages of programming language systems.

Acknowledgements

The author wishes to thank his principal advisor Professor Takuya Katayama of Japan Advanced Institute of Science and Technology, who has pointed out instructive suggestions on this paper. The author has sincere gratitude to him for granting the author this invaluable and irreplaceable oppotunity. The author wishes to thank his advisor Assosicate Professor Katsuhiko Gondow and his advisor Associate Professor Hajime Ishihara of Japan Advanced Institute of Science and Technology for their helpful discussions and suggestions. The author wishes to thank Professor Atsushi Ohori of Japan Advanced Institute of Science and Technology and Professor Takehiro Tokuda of Tokyo Institute of Technology for thier helpful discussions and accurate comments. The author wishes to thank Assistant Toshiaki Aoki of Japan Advanced Institute of Science and Technology for his helpful discussions and suggestions, and for his obliging advices. The author is also thankfull with Mr. Yasser Kotb who helped the author to detect insufficiency on the early works, with all members of Katayama-lab. in JAIST, and with all people at Kagawa University who have presented me passive but enough appreciation on my activities.

Contents

Chapter 1

Introduction and Related Works

Since Fortran, i.e., earliest high level programming language, was developed, various programming languages have been developed under various concepts. Some of them are still used for developments of actual systems, and also, some are objects of improvements. Besides them, significantly many programming languages have been proposed. However, it is obvious that there is a common problem among all of them, the problem how to give them their syntax and semantics, which is one of essences on development of programming language.

Most of syntaxes of programming languages are static. In other words, after a syntax of such a programming language is fixed, the language which is accepted by the system is also fixed. On the other hands, there are several programming languages which have dynamic syntaxes. For example, languages such as C are provided with apparent diversity of syntax, using pre-processor. Languages such as Algol68, Dec10-Prolog and SML/NJ enable for programmer to define new operators in program texts, which have some restrictions on definition. Languages such as C++ enable for programmer to define new semantics only on existing operators (over-loading), and so on. It is possible to regard all of them to have dynamic syntaxes, which means that their systems have syntaxes extensible on run-time without reconstructions of systems themselves. On languages described above, excepting C and C++, the extensibility is based on Operator Precedence, for Dec-10 Prolog, and TLG (Two Level Grammar) [22, 27], for Algol68. Implementation of extensibility of syntax based on Operator Precedence has advantage in the meaning that it has clarity and is easy to understand. However, because the language class of Operator Precedence is narrow, it has restriction when one applies it to extensibility on parts of a grammar other than concerning to operators. TLG is one of very complicated grammars. It has less readability, and moreover, it has no efficient parsing algorithm, if we make no restrictions on it [22].

In this paper, using a formal language system, RCFG (Reflective Context-Free Grammar), which is extensible on parse-time, we establish and propose a new parsing system which is able to deal language class larger than that of LALR(1). A hint for a frame work in which semantic descriptions are given for newly augmented production rules during parsing, the augmentation is enabled under the formalism of RCFG, are discussed in Appendix A. On the frame work, we consider two kinds of implementation model of extended Syntax Directed Translation (SDT) scheme, from the point of view of development stages of programming language systems.

RCFG is one of variations of CFG (Context-Free Grammar), which language class is middle between CFL (Context-Free Language) and CSL (Context-Sensitive Language), and which has an efficient parsing algorithm that is an extension of Earley's parsing algorithm for CFG [11]. Details of RCFG are provided in Chapter 3. To construct a parse-time extensible formal language system, it is an easy way to enable to embed newly defined production rules in text which is just being parsed. The effective range of newly defined production rule is after the embedded position of the rule. Deletion of production rule is not considered in the frame work of RCFG.

Some frame works for extensible grammars have been proposed([6, 7, 8, 30, 31], etc.). Many of those ([6, 7, 8], etc.) are based on Definite Clause Grammar [9], so as summarized in [8], and some others are based on Attributted Grammar [20, 21] and its resemblances [28]. There are several purpose to introduce extensibility to grammars. One of them is to describe the correspondence between declarations of variables of programming language and its uses. This problem requires a method in order to restrict the effective range of definitions of variables. So, in many approaches, parse-time augmentation and deletion of production rules have been indispensable aspects on extensible grammars. However, the function of deletion of production rules causes too much computability of the systems, which is equivalent to Turing Machine. Mostly alike system to RCFG is ECL [30, 31] on a few points. ECL is one of extensions of CFG, permits deletion of production rules, and its language class involves CSL, if no restrictions. Newly augmenting rules are extracted by a finite automaton with outputs defined with EC Grammar. We give an example language in Example 3.2.2, which is accepted by RCFG but ECL, that is of tricky one in some sense.

Our main purpose is to construct a frame work of compiler-compiler which is upper compatible to YACC [16] or Bison [10], and moreover, which can treat extensible grammar. According to this purpose, some restrictions and needs arise. 1) the base grammar processed by the system must be an extension of CFG. It must includes CFG as a special case. 2) with some restrictions on the base grammar, LALR(1) parsing scheme or some other scheme resemble to it can be processed on the system. 3) ambiguity of given grammar must be solved in YACC style. 4) about error handling. These are the reason why we propose RCFG, and 1) and 2) are solved in this paper. 3) and 4) are remained as a future works. Moreover, a frame work in which semantic descriptions are given for newly augmented production rules during parsing are discussed in Appendix A.

One might have a question on the expressive power of RCFG, because it abandoned the function of deletion of production rules. However, this simple choice leads us to harvests as a good properties of RCFG. No deletion means that all newly defined production rules have global scopes. Even if the scope is restricted to only global one, many applications are remained, e.g., operator declaration, introduction of new sentences, and so on. Additionally, it is well-known that careless deletion destroys modularity. Above all, RCFG gains efficiency on parsing. So our approach is quite a choice. The condition 1), above, is cleared with RCFG.

For a solution of the condition 2) above, we introduced a new approach of incremental construction of LALR(1) parser. A few methods had been proposed [12, 13, 15] as incremental construction system of LR parsers. All of these are not fit to our purpose, because they are for environments of development of programming languages, but for parsing system on extensible grammars. Their systems have complete item sets on each stage of computation. Calculation of Look Ahead Set is not clearly mentioned. It is conjectured to be calculated by conventional methods [1, 2, 3, 14], which is not incremental.

In contrast with these works, in our approach, each states of LR(0) parse table do not have any information of item set, and Look Ahead Symbol Set (LA) are calculated in fully incremental manner. To achieve it, several notions are introduced, and properties on them are established.

For semantic descriptions of production rules newly augmented during parsing, we start discussion with Syntax Directed Translation (SDT) scheme. SDT is a method of translating string to actions. SDT was introduced in [23] for the programming language LITHE which have flexible syntax. In Appendix A, we show a way to provide semantics for dynamically augmented rules, extending SDT so as to introduce a reflective method.

Chapter 2

Preliminaries

2.1 Basic Notations

We use the notion of *set* as a priori one, and sometime define sets denotatively such as, $\{x \mid P(x)\}$ with a predicate P. $x \in A$ denotes that a set A contains an object x, and x is called an *element* of A . An *empty set* which contains no element is denoted by ϕ. A set which contains finite elements in it is called a *finite set*, and a set which contains infinite elements in it an *infinite set*. $A \subset B$ denotes that B contains all elements of A, or simply A is called a *subset* of B. Especially, $A \subset B$ includes a case $A = B$. For a finite set $A = \{x_1, x_2, \ldots, x_n\}$, the number of elements of A is denoted by $\#(A)$ which is equal to n. Standard operations on sets, *union*, *intersection*, *difference*, *Cartesian product* and *power set* are denoted by $A \cup B$, $A \cap B$, $A \setminus B$, $A \times B$ and $\mathit{Power}(A)$, respectively. For a set of sets, say $C = \{A_1, A_2, \ldots\}$,

$$\bigcup_{A \in C} A$$

denotes

$$A_1 \cup A_2 \cup \cdots,$$

and also,

$$\bigcap_{A \in C} A$$

denotes

$$A_1 \cap A_2 \cap \cdots,$$

respectively. Especially, in this paper, we adopt the notation

$$\bigcup C$$

for the former expression.

For given $C \subset \mathit{Power}(A)$, if an element B of C satisfies the condition that $\forall B' \in C$, $B \subset B'$ implies $B = B'$, then B is called a *maximal element* of C, or simply *maximal.*

Binary Relation R on sets A and B is a subset of $A \times B$. When elements $x \in A$ and $y \in B$ have relation R, we write $x \, R \, y$ or $(x, y) \in R$. When elements $x \in A$ and $y \in B$ do not have relation R, we write $x \not R y$ or $(x, y) \notin R$. For a relation R on $A \times A$, if $\forall x \in A$, $x \, R \, x$, then R is called *reflective*, if $\forall x, y, z \in A$, $x \, R \, y$ and $y \, R \, z$ imply $x \, R \, z$, then

R is called *transitive*, and if $\forall x, y \in A$, $x\,R\,y$ implies $y\,R\,x$, then R is called *symmetric.* A relation which is reflective, transitive and symmetric is called *equivalent relation.* For given set A and its division $D = \{A_i\}$, where $\bigcup D = A$, if D satisfies the conditions,

i) $\forall i$, $A_i \neq \phi$,

ii) $\forall i, j$, if $i \neq j$, then $A_i \cap A_j = \phi$,

iii) $\forall i$, $\forall x, y \in A_i$, $x\,R\,y$,

iv) $\forall i, j$, s.t., $i \neq j$, $\forall x \in A_i$, $y \in A_j$, $x \not R\, y$,

D is called *quotient set* and denoted by A/R. Each element of D is called *equivalent class.* If an element x of A is involved in an equivalent class A_i, A_i is denoted by $R[x]$, or simply by $[x]$.

A *function* f is assumed to be a subset of $A \times B$, which satisfies the condition $\forall x \in A$, there exists an element $(x, y) \in f$ for some $y \in B$, and $\forall x \in A$, (x, y_1), $(x, y_2) \in f$ implies $y_1 = y_2$. The function type of f is denoted by $A \rightarrow B$. A is called the *domain* of the function, and B the *co-domain.* If $(x, y) \in f$, y is written by $f(x)$. A function is also called a *mapping.* If given function f satisfies the condition that $\forall x, y \in A$, $f(x) = f(y)$ implies $x = y$, then f is called a *one-to-one* mapping. If given function f satisfies the condition that $\forall y \in B$, there exists an element $x \in A$, s.t., $f(x) = y$, then f is called an *onto* mapping. A mapping which is one-to-one and onto is also called *isomorphism.* For given sets A and B, if there exists a one-to-one, onto mapping between A and B, then A and B are called *isomorphic*, sometimes with some structural conditions.

Especially in this paper, each subset R of $A \times B$ is assumed as a function $R : A \rightarrow \mathit{Power}(B)$, so as that $R(x) = \{y \in B \mid (x, y) \in R\}$. So, for given functions $f, g : A \rightarrow \mathit{Power}(B)$, treating f and g as subsets of $A \times B$, we define operations on functions $f \cap g$ and $f \cup g$, as below,

$$\begin{aligned} (f \cap g)(x) &= f(x) \cap g(x), \\ (f \cup g)(x) &= f(x) \cup g(x). \end{aligned}$$

2.2 Graph

A *graph* G is given by a 2-tuple (V, E), where V is called a *vertex set* of *verteces*, and $E \subset \mathit{Power}(V)$ called an *edge set* of *edges.* Each element of E is an unordered pair of V. The number of elements of each element of E is one or two. An *oriented graph*, or called *directed graph*, is given also by (V, E), where V is a vertex set and E is a subset of $V \times V$, which is called an *edge set* of *arcs.* An arc is also called an edge. Sometimes, V or E may be infinite sets, however, we deal with only finite sets and directed graphs, moreover, usually each vertex and each edge are labelled, in this papper. An edge $(x, y) \in E$ designates that the edge joins verteces x and y, x is called a *predecessor* and y a *successor.* A mutual sequence of verteces and edges, say $v_0, e_1, v_1, \ldots, e_n, v_n$ is called *path*, when each e_i joins v_{i-1} and v_i, where $v_i \in V$ and $e_i \in E$. In this case, v_n is called *reachable* from v_0.

A directed graph is called a *tree*, if the following conditions are satisfied,

i) there is a unique vertex, called *root*, which has no predecessor,

ii) each vertex is reachable from the root,

iii) each vertex except the root has unique predecessor,

iv) all successors of each vertex, if exist, have an order from left to right.

For given directed graph $G = (V, E)$ and $V' \subset V$, *vertex induced subgraph* G' is given so as,

$$G' = (V', E \cap (V' \times V')).$$

2.3 String and Language

An *alphabet* is a finite set of symbols, occasionally denoted by Σ with a suffix. A *string* is a finite sequence of symbols, sometimes denoted by u,v and w with a suffix. The length of a string w is written by $\mid w \mid$ which denotes the number of symbols that w consists of. A zero length string is called *empty string* or *null string*, and denoted by ε. *Concatenation* of two strings w_1 and w_2 is simply denoted by w_1w_2. On concatenations of strings, the equation $w_1(w_2w_3) = (w_1w_2)w_3$ holds for any strings w_1, w_2 and w_3.

A set of whole strings which consists of symbols in Σ is denoted by $\Sigma*$, and $\Sigma * \backslash\{\varepsilon\}$, the set which consists of whole positive length strings, is denoted by $\Sigma+$. A language is a subset of $\Sigma*$. An element of a language is also called *word*. A *product* of two languages L_1 and L_2 is denoted by L_1L_2, and defined as,

$$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

2.4 Finite Automata and Regular Expression

A *Finite Automaton* (FA) A is defined with 5-tuple as below,

$$A = (\Sigma, Q, \delta, q_0, F)$$

Σ is a finite set of symbols (*alphabet*),

Q is a finite set of *states*,

δ is a *state transition function*, which function type depends on the type of Automata,

DFA(*Deterministic Finite Automata*):

$$\delta : Q \times \Sigma \to Q$$

εNFA(*Non-deterministic Finite Automata* with *ε-transitions*)

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to Power(Q)$$

q_0 is *initial state*,

F is a subset of Q called *final states*.

For given εNFA, an ε-Closure for each state q is denoted by $\varepsilon C(q)$, and defined by,

1. $q \in \varepsilon C(q)$,
2. $q' \in \varepsilon C(q)$, then $\delta(q', \varepsilon) \subset \varepsilon C(q)$,
3. $\varepsilon C(q)$ is a minimum set which satisfies above two conditions.

εC is extended to a function $\varepsilon \hat{C} : \mathit{Power}(Q) \to \mathit{Power}(Q)$ defined as,

$$\varepsilon \hat{C}(U) = \bigcup_{q \in U} \varepsilon C(q),$$

where U is a subset of Q. We write εC also for $\varepsilon \hat{C}$, if no ambiguity. When to emphasize the ε-Closure is under the transition function δ, we write $\varepsilon C(\delta, q)$ or $\varepsilon C(\delta, U)$.

Lemma 2.4.1

$$\begin{aligned} \varepsilon C(U \cup V) &= \varepsilon C(U) \cup \varepsilon C(V) \\ \varepsilon C(\varepsilon C(U)) &= \varepsilon C(U) \end{aligned}$$

These are well-known results. Proofs are omitted.

The transition function δ is usually extended by following manner,

for DFA:

$$\begin{aligned} \delta^*(q, \varepsilon) &= q, \\ \delta^*(q, w\,a) &= \delta(\delta^*(q, w), a), \end{aligned}$$

for NFA:

$$\begin{aligned} \hat{\delta}(U, \varepsilon) &= \varepsilon C(U), \\ \hat{\delta}(U, w\,a) &= \varepsilon C(\bigcup_{q \in \hat{\delta}(U, w)} \delta(q, a)). \end{aligned}$$

We write δ for δ^* or $\hat{\delta}$, if no ambiguity.

The language $L(A)$ of given DFA A is stated as,

$$L(A) = \{ w \in \Sigma* \mid \delta(q_0, w) \in F \},$$

and also the language $L(A)$ of given εNFA A is stated as,

$$L(A) = \{ w \in \Sigma* \mid \delta(\{q_0\}, w) \cap F \neq \phi \}.$$

For given εNFA A, a DFA A' which is equivalent to A is constructed as,

$$\text{DFA } A' = (\Sigma, Q', \delta', q_0', F'),$$

where,

$$\begin{aligned} Q' &= Power(Q), \\ \delta' &= \hat{\delta}, \\ q_0' &= \varepsilon C(q_0), \\ F' &= \{U \subset Q \mid U \cap F \neq \phi\}, \end{aligned}$$

is called *Subset Construction*, and denoted by $SC(A)$ in this paper. See [14] for detail.

Regular Expression (RE) forms a language class equivalent to the class of FA. In this paper, sometimes we adopt the following notations for RE,

empty language	ϕ,	empty string	ε,
union	$R_1 + R_2$,	product	R_1R_2,
Kleene closure	$R*$.		

$RR*$ is abbreviated to $R+$.

2.5 Context-Free Grammar

Context-Free Grammar (CFG) is defined with 4-tuple $G = (V, T, P, S)$, where,

V is a finite set of *syntactic variables*, or called *non-terminals*,

T is a finite set of *terminal symbols*,

P is a finite set of *production rules*, which is a finite subset of $V \times (V \cup T)*$,

$S \in V$ is *start variable.*

Each production rule $(A, \alpha) \in P$ is denoted by $A \to \alpha$.

The derivation relation $\Rightarrow$ on $(V \cup T)*$ is defined as,

$$\gamma_1 A \gamma_2 \Rightarrow \gamma_1 \alpha \gamma_2,$$

if $A \to \alpha \in P$, for each $\gamma_1, \gamma_2 \in (V \cup T)*$. The reflective transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$.

The language stated by G is denoted by $L(G)$, so as,

$$L(G) = \{w \in T* \mid S \overset{*}{\Rightarrow} w\}.$$

2.6 Parsing Algorithms for CFG

In this section, two parsing algorithms, i.e. Earley's algorithm [11] and LALR(1) [1, 2, 3], are summarized. Earley's algorithm is the basis of general parsing algorithm for RCFG defined in Section 3.4.2. The definitions for LALR(1) stated here (Definition 2.6.3 and 2.6.7) are not standard. The definitions are constructed so as to make the structur of LALR(1) graph clear and to make incremental construction method discussed in Chapter 4 easily. For each algorithms, i.e. Earley's algorithm, LALR(1) and general parsing algorithm for RCFG, notions *item* are defined. The term 'item' is used for the distinct notions of each algorithms, according to conventional manner.

Earley's Algorithm

Algorithm 2.6.1 (Earley's Algorithm)
Suppose a CFG $G = (V, T, P, S)$ is given. An item is denoted by $[A \to \alpha \bullet \beta, i]$, where $A \to \alpha\beta \in P$ and $i \geq 0$ is an integer. For an input string $w = a_1 a_2 \cdots a_n$, parse lists $I_0, I_1, \ldots, I_n$ are being calculated during parsing in Ealey's algorithm. Each parse list I_j consists of items.

Initial Phase:

1) *initialize $I_0, I_1, \ldots, I_n$ to ϕ,*

2) *add item $[S \to \bullet\alpha, 0]$ to I_0, for each $S \to \alpha \in P$,*

repeat 3), 4) until no new item is added to I_0,

3) *if $[A \to \alpha \bullet B\beta, 0]$ and $[B \to \gamma\bullet, 0]$ are in I_0, then add $[A \to \alpha B \bullet \beta, 0]$ to I_0,*

4) *if $[A \to \alpha \bullet B\beta, 0]$ is in I_0, then add $[B \to \bullet\gamma, 0]$ to I_0 for each $B \to \gamma \in P$,*

Main Loop:
repeat 5), 6) until no new item is added to $I_0, \ldots, I_n$,

5) *if $[A \to \alpha \bullet a\,\beta, i]$ is in I_j, and $a_{j+1} = a$, then add $[A \to \alpha\, a \bullet \beta, i]$ to I_{j+1},*

6) *if $[A \to \alpha \bullet B\beta, i]$ is in I_j and $[B \to \gamma\bullet, j]$ is in I_k, then add $[A \to \alpha\, B \bullet \beta, i]$ to I_k,*

Judgement:

7) *if $[S \to \alpha\bullet, 0]$ is in I_n, then the input is accepted, otherwise rejected.*

LALR(1)

In following descriptions, each given CFGs are assumed to be *Extended Grammars.* For given CFG $G = (V, T, P, S)$, an extended grammar G' is given by,

$$G' = (V', T, P', S'),$$

where,

$$\begin{array}{rcl} V' & = & V \cup \{S'\}, \\ P' & = & P \cup \{S' \to S\}, \end{array}$$

for newly added syntactic variable S'.

Definition 2.6.2 (*Item*, Item Set)
Item Set is defined as a subset of $V \times (V \cup T)* \times (V \cup T)*$. Each item $(A, \alpha, \beta) \in V \times (V \cup T)* \times (V \cup T)*$ is denoted by $A \to \alpha \bullet\beta$. For given CFG $G = (V, T, P, S)$, item set Item derived from G is defined as

$$Item = \{A \to \alpha \bullet \beta \mid A \to \alpha\beta \in P\}.$$

When to restrict it on items which have syntactic variable X on left-hand side, it is denoted by $Item(X)$,

$$Item(X) = \{X \to \alpha \bullet \beta \mid X \to \alpha\beta \in P\}.$$

When to emphasize that Item is derived from CFG G or its production rule set P, we write $Item_G$, $Item_P$ or $Item(P)$, and also $Item_G(X)$ or $Item_P(X)$.

Definition 2.6.3 (LR(0) graph)
For given CFG G, LR(0) graph (εNFA) $lr(G)$ derived from G is defined as

$$\begin{array}{lcll} lr(G) & = & (V \cup T, Item \cup \{q_0\}, \delta, q_0, \phi) & \\ & & & \\ \delta(q_0, \varepsilon) & = & \{S \to \bullet\alpha \mid S \to \bullet\alpha \in Item\} & \\ \delta(q_0, X) & = & \phi & (X \in V \cup T) \\ \delta(A \to \alpha \bullet X\beta, X) & = & \{A \to \alpha X \bullet \beta\} & \\ \delta(A \to \alpha \bullet X\beta, Y) & = & \phi & (X \neq Y) \\ \delta(A \to \alpha \bullet X\beta, \varepsilon) & = & \{X \to \bullet\alpha \mid X \to \bullet\alpha \in Item\} & \end{array}$$

From now on, we prepare basic notions and notations for items.

Definition 2.6.4 (*εItem*, Reduce Item)
For given CFG $G = (V, T, P, S)$, let U be a set of items, wrote $U \subset Item$. A set of all items which dot are on top of right-hand side, i.e., $X \to \bullet\alpha$, is called εItem. It is strictly defined as

$$\varepsilon Item = \{Y \to \bullet\alpha \mid Y \to \alpha \in P\}.$$

We call every elements of εItem , also, εItem. A set of εItems, concerning to given syntactic variable X, is denoted by $\varepsilon Item(X)$, and defined as,

$$\varepsilon Item(X) = \{X \to \bullet\alpha \mid X \to \alpha \in P\}.$$

A kind of items, $X \to \alpha\bullet$, is called Reduce Item.

Definition 2.6.5 (*Root*, *Kernel*)
For given εNFA $lr(G) = (V \cup T,\ Item \cup \{q_0\},\ \delta,\ q_0,\ \phi)$ of given CFG G, if state q of $SC(lr(G)) = (V \cup T,\ Power(Item \cup \{q_0\}),\ \delta',\ q_0',\ \phi)$ contains a reduce item $A \to \alpha\bullet$, $Root(q,\ A \to \alpha\bullet)$ denotes a set of states in which an item $A \to \bullet\alpha$ is produced and it is a root of $A \to \alpha\bullet$ in q.

$$Root(q, A \to \alpha\bullet) = \{q' \subset Item \cup \{q_0\} \mid \delta'(q', \alpha) = q, A \to \bullet\alpha \in q'\}.$$

Also we define Kernel Set of State $q \in Power(Item \cup \{q_0\})$, denoted by $Ker(q)$, such as,

$$\begin{array}{lcll} Ker(q_0') & = & \varepsilon C(q_0) \setminus \{q_0\}, & \\ Ker(q) & = & \delta(q', X) & \quad where\ \hat{\delta}(q', X) = q. \end{array}$$

In following, firstly, we collect notions concerning to LA, which are arranged in expected forms.

Definition 2.6.6 (*First*)
For given CFG $G = (V,\ T,\ P,\ S)$ and given $\alpha \in (V \cup T)*$, a set of first symbols derived from α, denoted by $First(\alpha)$, is defined as

$$First(\alpha) = \{a \in T \mid \alpha \overset{*}{\Rightarrow} a\alpha'\}.$$

If emphasizing the value of First is on grammar G or production rule set P, we write $First_G(\alpha)$ or $First_P(\alpha)$.

Definition 2.6.7 (Look Ahead Sets of LALR(1))
For given CFG $G = (V,\ T,\ P,\ S')$, let $SC(lr(G)) = (V \cup T,\ Q,\ \delta',\ q_0',\ *)$, where $Q = Power(Item \cup \{q_0\})$ and $q_0' = \varepsilon C(\delta, q_0)$, then a function $\lambda : Q \times Item \to Power((V \cup T)*)$ is defined as,

$$\begin{array}{lcl} \lambda(q_0', S' \to \bullet S) = \{\$\} & & \\ A \to \alpha \bullet X\beta \in q, \delta'(q, X) = q' & \Rightarrow & \lambda(q, A \to \alpha \bullet X\beta) \subset \lambda(q', A \to \alpha X \bullet \beta) \\ A \to \alpha \bullet B\beta \in q & \Rightarrow & \forall B \to \gamma \in P, \beta\lambda(q, A \to \alpha \bullet B\beta) \subset \lambda(q, B \to \bullet\gamma) \\ \forall i \in Item, s.t., i \notin q & \Rightarrow & \lambda(q, i) = \phi \end{array}$$

(the value of λ is minimum set that satisfies above condition).

LA for each reduce item $A \to \alpha\bullet$ in each state of $SC(lr(G))$ is obtained by $First(\lambda(q,\ A \to \alpha\bullet))$.

Next propositon is well-known, but the substance of the proof is very important on the discussions in Section 4.4. So, a brief proof is attached.

Proposition 2.6.8 *For any given CFG $G = (V,\ T,\ P,\ S)$ and any symbol sequences α, $\beta \in (V \cup T)*$, $First(\alpha\alpha\beta) = First(\alpha\beta)$.*

(proof) Consider two cases that i) ε is derived from α and ii) ε is not derived from α. On the case i), $First(\alpha\beta) = First(\alpha) \cup First(\beta)$ and $First(\alpha\alpha\beta) = First(\alpha) \cup First(\alpha\beta) = First(\alpha) \cup First(\alpha) \cup First(\beta) = First(\alpha) \cup First(\beta)$, and on the case of ii) $First(\alpha\beta) = First(\alpha)$ and $First(\alpha\alpha\beta) = First(\alpha)$. On both cases, the equation holds.//

Proposition 2.6.9 *For any $\varepsilon Item$ which belongs to state q of $SC(lr(G))$ and which left-hand side is common syntactic variable, say A, $A \to \bullet\alpha$, $A \to \bullet\beta \in q \cap \varepsilon Item(A)$,*

$$\lambda(q, A \to \bullet\alpha) = \lambda(q, A \to \bullet\beta).$$

We write $Ind\lambda(q, A) = \lambda(q,\, A \to \bullet\ \alpha)$, which means Induced LA Set for syntactic variable A at state q.

Chapter 3

Reflective Context-Free Grammar

In this chapter, we introduce a formal language system, named Reflective Context-Free Grammar (RCFG), which provides a basis of self-extensible language systems. RCFG is a simple extension of CFG. Its language class is middle between CFL and CSL. And, moreover, it has an efficient general parsing algorithm which is an extension of Earley's parsing algorithm for CFG. The idea of RCFG is quite simple. Production rule description of CFG forms, such as,

$$Exp \rightarrow Exp\ '+'\ Exp.$$

The "Grammar of Production Rule of CFG" is also defined in CFG,

$$\begin{array}{rcl} \textit{Production-Rule} & \rightarrow & \textit{Variable}\ '\rightarrow'\ \textit{Symbol-Sequence} \\ \textit{Symbol-Sequence} & \rightarrow & \\ \textit{Symbol-Sequence} & \rightarrow & \textit{Variable Symbol-Sequence} \\ \textit{Symbol-Sequence} & \rightarrow & \textit{Terminal Symbol-Sequence.} \end{array}$$

This observation is the start point of RCFG. To realize self-extensible language system, we adopt a way so as to enable for programmers and/or developers to embed new production rules into program texts which is just being parsed. For example, in SML/NJ , new operators will be defined by use of 'infix' of 'infixr', such as,

infix 5 *newop*;.

In this statement, a new operator '*newop*' is defined as an infix operator with precedence number '5'. Here, we ignore precedence number, because it is a parsing algorithm dependent notion. If it is permitted that a portion of syntactic rules is used at defining new operator, such that,

Exp newop Exp is also *Exp*,

we gain freedom to define new operators which have another types other than infix operators, as follow,

$$Exp\ ?\ Exp : Exp \text{ is also } Exp. \tag{3.1}$$

The main idea of RCFG is to embed portions of production rules in texts which are derived by given RCFG augmented with these embedded portions themselves. In other words, texts with which RCFG deals is 'self referential' in the sense that the text contains information or partial definitions about text itself. This feature is the most significant difference with CFG. For example, in RCFG, description (3.1) above is written as,

$$[\ \underline{Exp}\ \rhd\ \underline{Exp}\ ?\ \underline{Exp}\ :\ \underline{Exp}\ ].$$

'[' and ']' indicate the sentense enclosed by these brackets is an embedded production rule. '$\underline{Exp}$' is a terminal symbol which is introduced in order to express a syntactic variable Exp. When such a description is derived from specified syntactic variable, an augmentation of designated production rule happens.

In following of this chapter, we provide a formal definition of RCFG, definition of derivation sequence, examples of RCFG, some properties on RFCG, general parsing algorithm which is an extension of Earley's parsing algorithm for CFG, soundness and completeness of the algorithm and discussion on the efficiency of the algorithm, in this order.

3.1 Definition of Reflective Context-Free Grammar

3.1.1 Formalization of RCFG

Definition 3.1.1 (RCFG)
RCFG G is defined with 8-tuple as below,

$$G = (V, T, M, D, \mathbf{Aug}, f, P, s)$$

V *is a finite set of Syntactic Variable,*

T *is a finite set of Terminal Symbol, especially including special symbols* $[, \rhd\ ,]$, *which are used for parsing-time, augmentation (dynamic extension) of Production Rules ,*

M *is a subset of* T*, called Meta-Symbols,*

D *is a subset of* M *, called Definables,*

Aug *is a subset of* V*, each elements of which causes Augmentation of production rules,*

f *is a one-to-one, onto map, s.t.,* $M \to V$*,which is used for interpretation of augmenting production rules,*

P *is a finite set of (Initial)Production Rules, each production rule is an element of* $V \times (V \cup T)*$,

s *is Start Symbol.*

From now on, we introduce two notions, FSP (Feasible Set of Production rules) and AF (Augmented Forms) to discuss and define dynamic augmentation of production rules. In RCFG framework, it is not meant that any production rules will be augmented. Production rules which are possibly augmented are predictable. Roughly, those candidates are included in $\{X \to \alpha \mid X \in f(D),\ \alpha \in (f(M) \cup T)*\}$, which is named FSP. In following discussions, we assume the domain of newly augmented production rules is FSP.

AF is one of most important notions on RCFG. On CFG, derivation is defined on sequences on syntactic variables and terminal symbols. In contrast to CFG, it is need to express the augmentation of production rules, in RCFG. So, derivation on RCFG must contains production rule information which denotes a collection of production rules available to be used at the point of derivation. An intuitive explanation for AF is given after the definition, again.

Definition 3.1.2 (feasible set of production rules)
For given RCFG $G = (V, T, M, D, \mathbf{Aug}, f, P, s)$*, a feasible set of production rules FSP is defined as bellow,*

$$FSP = P \cup (f(D) \times (f(M) \cup T)*).$$

Definition 3.1.3 (Augmented Form)
Augmented Form of Syntactic Variable $A \in V$ *forms 4-tuple,*

$$(A, P_1, P_2, w),$$

where P_1 *and* P_2 *are finite sets of production rules, and* $w \in T*$*. Also, Augmented Form of Terminal Symbol* $a \in T$ *forms,*

$$(a, P_1, P_2, w).$$

We simply call Augmented Form (AF) for both.

To reduce descriptions in following discussions, an AF sequence (X_1, P_1, P_2, u_1) $(X_2, P_2, P_3, u_2) \cdots (X_n, P_n, P_{n+1}, u_n)$ may be abbreviated to $(\alpha, P_1, P_{n+1}, u)$, where $\alpha = X_1X_2 \cdots X_n$, $u = u_1u_2 \cdots u_n$.

Intuitively, an augmented forms (X, P_1, P_2, w) indicates that string w is derived from X by use of production rules in P_2, and, while derivation is done, some new production rules which are embedded in w are attached to P_1. P_1 is to be seen as a 'core rule set for production,' and P_2 to be seen as a 'augmented rule set by production.'

3.1.2 Notations and Terminologies for RCFG

Notation 3.1.4 (Notations on RCFG)

A capital letter, A, B, C, ..., possibly with some suffix, denotes a syntactic variable exclude special syntactic variables in **Aug**.

A lowercase letter, a, b, c, ..., possibly with some suffix, denotes a terminal symbol exclude "[", "$\triangleright$" and "]".

An X, Y, Z, possibly with some suffix, denotes a syntactic variable or a terminal symbol.

A u, v, w, possibly with some suffix, denotes finite sequences of elements of T, string.

A small Greek letter, α, β, γ, ..., possibly with some suffix, denotes a finite sequence of elements of T and V.

An element (A, α) of P and FSP is denoted using arrow as '$A \rightarrow \alpha$'.

A P, Q, possibly with some suffix, denotes a finite subset of FSP.

We extend f to $\hat{f} : T \rightarrow V \cup T$, where

$$\hat{f}(a) = \begin{cases} f(a) & \text{if } a \in M \\ a & \text{otherwise.} \end{cases}$$

Also, $\hat{f}* : T* \to (V \cup T)*$ is defined recursively, where $\hat{f}*(\varepsilon) = \varepsilon$, $\hat{f}*(aw) = \hat{f}(a)\hat{f}*(w)$, and, $\tilde{f} : T* \to V \times (V \cup T)*$, where

$$\tilde{f}(w) = \begin{cases} A \to \alpha & \text{if } \hat{f}*(w) \text{ forms } A \rhd \alpha, A \in f(D) \text{ and } \alpha \in (f(M) \cup T)* \\ \top & \text{otherwise.} \end{cases}$$

We use f in order to denote one of all these functions, if no ambiguity.

Terminologies

When A is a member of $f(M)$ then we term A a *public variable.* When A is a member of $f(D)$ then we term A a *definable variable.* When for a string $[w] \in T*$, $f(w) \neq \top$ then we term $[w]$ an *embedded portion.*

3.1.3 Derivation and Language of RCFG

Definition 3.1.5 (Derivation of RCFG)
A binary relation $\Rightarrow$ on finite sequences of augmented forms is defined as,

1) ordinal case*,*

$$\begin{aligned}&(\alpha, P, P_1, w_1)(A, P_1, P_{n+1}, w_2)(\beta, P_{n+1}, P_{n+2}, w_3)\\ &\quad\Rightarrow\quad (\alpha, P, P_1, w_1)(X_1, P_1, P_2, u_1)\cdots(X_n, P_n, P_{n+1}, u_n)(\beta, P_{n+1}, P_{n+2}, w_3)\end{aligned}$$

with constraints,

- $A \notin \mathbf{Aug}$,
- $A \to X_1 \cdots X_n \in P_{n+1}$,
- $w_2 = u_1 \cdots u_n$,
- $\forall i = 1, \ldots, n,$ if $X_i \in T$, then $u_i = X_i$ and $P_{i+1} = P_i$,

2) reflective case*,*

$$\begin{aligned}&(\alpha, P, P_0, w_1)(\mathbf{p}, P_0, P'_1, w_2)(\beta, P'_1, P'_2, w_3)\\ &\quad\Rightarrow\quad (\alpha, P, P_0, w_1)([, P_0, P_0, [)\\ &\qquad\quad (X_0, P_0, P_1, u_0)(\rhd, P_1, P_1, \rhd)(X_1, P_1, P_2, u_1)\cdots(X_n, P_n, P_{n+1}, u_n)\\ &\qquad\quad (], P_{n+1}, P'_1,])(\beta, P'_1, P'_2, w_3)\end{aligned}$$

with constraints,

- $\mathbf{p} \in \mathbf{Aug}$
- $w_2 = [u_0 \rhd u_1 \cdots u_n]$
- $\mathbf{p} \to [X_0 \rhd X_1 \cdots X_n] \in P'_1$ *and* $f(u_0 \rhd u_1 \cdots u_n)$ *has valid form of production rule,*
- $\forall i = 0, \ldots, n,$ if $X_i \in T$, then $u_i = X_i$ *and* $P_{i+1} = P_i$,
- $P'_1 = P_{n+1} \cup \{f(u_0 \rhd u_1 \cdots u_n)\}$.

We write reflective transitive closure of $\Rightarrow$ by $\overset{*}{\Rightarrow}$.

This definition of derivation is some what complicated. In this definition, production rule sets which occur in each AFs has no procedure to calculate them. Only they have constraints on them, some of them are provided in definition explicitly as 'constraints' conditions as described above, and the other is implicitly provided in the definition, e.g., such as P_1 in AF sequence (α, P_0, P_1, w_1) (β, P_1, P_2, w_2). Superficially, one might consider that if appropriate combinations of production rule sets and strings were provided for each AFs, then an unexpected derivation would happen. However, such an unexpected derivation never occurs, because of the constraints described above, and this feature is established with propositions stated in the following sections.

Here, we give an example which language is out of CFL.

Example 3.1.6 (Grammar out of CFG)
Consider RCFG $G_1 = (V, T, M, D, \mathbf{Aug}, f, P, s)$, where

$$\begin{array}{rcl} V &=& \{s, \mathbf{p}, A, B\}, \\ T &=& \{0, 1, [, \rhd,], \underline{A}\}, \\ M &=& D \ = \ \{\underline{A}\}, \\ \mathbf{Aug} &=& \{\mathbf{p}\}, \\ f(\underline{A}) &=& A, \\ P &=& \{s \to \mathbf{p}A, \mathbf{p} \to [\underline{A} \rhd B], B \to \varepsilon \mid 0\,B \mid 1\,B\}, \end{array}$$

This example specifies a language $L(G_1) = \{[\underline{A} \rhd w]w \mid w \in \{0,1\}*\}$. Since there is no production rule in P, which has A on left-hand side, for any AF (s, P, P', u), after a derivation $(s, P, P', u) \Rightarrow$ ($\mathbf{p}$, P, P', u_1) (A, P', P', u_2), only one derivation is able to be done from AF (A, P', P', u_2), using an augmented rule derived from $\mathbf{p}$. A definition of language of RCFG will be given in Definition 3.1.9 below. A derivation sequence for terminal string $[\underline{A} \rhd 01]\ 01$ can be given as below,

$$\begin{array}{ll} \multicolumn{2}{l}{(s, P, P', [\underline{A} \rhd 01]01)} \\ \Rightarrow & (\mathbf{p}, P, P', [\underline{A} \rhd 01])(A, P', P', 01) \\ \Rightarrow & ([, P, P, [)(\underline{A}, P, P, \underline{A})(\rhd, P, P, \rhd)(B, P, P, 01)(], P, P',])(A, P', P', 01) \\ \Rightarrow & ([, P, P, [)(\underline{A}, P, P, \underline{A})(\rhd, P, P, \rhd)(0, P, P, 0)(B, P, P, 1)(], P, P',])(A, P', P', 01) \\ \Rightarrow & ([, P, P, [)(\underline{A}, P, P, \underline{A})(\rhd, P, P, \rhd)(0, P, P, 0)(1, P, P, 1)(B, P, P, \varepsilon)(], P, P',]) \\ & (A, P', P', 01) \\ \Rightarrow & ([, P, P, [)(\underline{A}, P, P, \underline{A})(\rhd, P, P, \rhd)(0, P, P, 0)(1, P, P, 1)(\varepsilon, P, P, \varepsilon)(], P, P',]) \\ & (A, P', P', 01) \\ \Rightarrow & ([, P, P, [)(\underline{A}, P, P, \underline{A})(\rhd, P, P, \rhd)(0, P, P, 0)(1, P, P, 1)(\varepsilon, P, P, \varepsilon)(], P, P',]) \\ & (0, P', P', 0)(1, P', P', 1) \end{array}$$

where $P' = P \cup \{A \to 01\}$. If P' is given other than $P \cup \{A \to 01\}$, the derivation on $\mathbf{p}$ must be fault. Of course, there are infinitely many choices to give arguments of AFs. However, for successful derivation, there must be given valid combination of values, because any derivation must satisfy the constraints given in the definition of derivation. On the above derivation example, the combination given is unique. Properties on arguments of AFs on derivations are established in Section 3.3.

Similar to CFG, on RCFG, notions of *derivation tree*, *leftmost derivation* and *rightmost derivation* are likely defined. These notions make clear the effective ranges of embedded portions in the texts contain them. Especially, leftmost derivation is used in the proof of completeness of the general parsing algorithm for RCFG, stated in Section 3.4.2.

Definition 3.1.7 (Leftmost Derivation) *Leftmost derivation is, informally, stated so that the subjects of derivation on each points are 'leftmost' syntactic variables. Formally, it is stated with binary relation $\Rightarrow_L$, as follows,*

1) ordinal case,

$$\begin{aligned}&(v, P, P_1, w_1)(A, P_1, P_{n+1}, w_2)(\beta, P_{n+1}, P_{n+2}, w_3)\\ &\quad\Rightarrow_L \quad (v, P, P_1, w_1)(X_1, P_1, P_2, u_1)\cdots(X_n, P_n, P_{n+1}, u_n)(\beta, P_{n+1}, P_{n+2}, w_3)\end{aligned}$$

with constraints,

- $A \notin \mathbf{Aug}$,
- $A \rightarrow X_1 \cdots X_n \in P_{n+1}$,
- $w_2 = u_1 \cdots u_n$,
- $\forall i = 1, \ldots, n,$ if $X_i \in T$, then $u_i = X_i$ and $P_{i+1} = P_i$,

2) reflective case,

$$\begin{aligned}&(v, P, P_0, w_1)(\mathbf{p}, P_0, P'_1, w_2)(\beta, P'_1, P'_2, w_3)\\ &\quad\Rightarrow_L \quad (v, P, P_0, w_1)([, P_0, P_0, [)\\ &\qquad\quad (X_0, P_0, P_1, u_0)(\rhd, P_1, P_1, \rhd)(X_1, P_1, P_2, u_1)\cdots(X_n, P_n, P_{n+1}, u_n)\\ &\qquad\quad (], P_{n+1}, P'_1,])(\beta, P'_1, P'_2, w_3)\end{aligned}$$

with constraints,

- $\mathbf{p} \in \mathbf{Aug}$
- $w_2 = [u_0 \rhd u_1 \cdots u_n]$
- $\mathbf{p} \rightarrow [X_0 \rhd X_1 \cdots X_n] \in P'_1$ *and* $f(u_0 \rhd u_1 \cdots u_n)$ *has valid form of production rule,*
- $\forall i = 0, \ldots, n,$ if $X_i \in T$, then $u_i = X_i$ *and* $P_{i+1} = P_i$,
- $P'_1 = P_{n+1} \cup \{f(u_0 \rhd u_1 \cdots u_n)\}$,

where $v \in T*$.

Definition 3.1.8 (Rightmost Derivation) *Rightmost derivation is, informally, stated so that the subjects of derivation on each points are 'rightmost' syntactic variables. Formally, it is stated with binary relation $\Rightarrow_R$, as follows,*

1) ordinal case,

$$\begin{aligned}&(\alpha, P, P_1, w_1)(A, P_1, P_{n+1}, w_2)(v, P_{n+1}, P_{n+2}, w_3)\\ &\quad\Rightarrow_R \quad (\alpha, P, P_1, w_1)(X_1, P_1, P_2, u_1)\cdots(X_n, P_n, P_{n+1}, u_n)(v, P_{n+1}, P_{n+2}, w_3)\end{aligned}$$

with constraints,

- $A \notin \mathbf{Aug}$,
- $A \to X_1 \cdots X_n \in P_{n+1}$,
- $w_2 = u_1 \cdots u_n$,
- $\forall i = 1, \ldots, n$, if $X_i \in T$, then $u_i = X_i$ and $P_{i+1} = P_i$,

2) reflective case,

$$\begin{aligned}&(\alpha, P, P_0, w_1)(\mathbf{p}, P_0, P'_1, w_2)(v, P'_1, P'_2, w_3)\\ &\Rightarrow_R \;\; (v, P, P_0, w_1)([, P_0, P_0, [)\\ &\qquad (X_0, P_0, P_1, u_0)(\rhd, P_1, P_1, \rhd)(X_1, P_1, P_2, u_1) \cdots (X_n, P_n, P_{n+1}, u_n)\\ &\qquad (], P_{n+1}, P'_1,])(v, P'_1, P'_2, w_3)\end{aligned}$$

with constraints,

- $\mathbf{p} \in \mathbf{Aug}$
- $w_2 = [u_0 \rhd u_1 \cdots u_n]$
- $\mathbf{p} \to [X_0 \rhd X_1 \cdots X_n] \in P'_1$ *and* $f(u_0 \rhd u_1 \cdots u_n)$ *has valid form of production rule,*
- $\forall i = 0, \ldots, n$, if $X_i \in T$, then $u_i = X_i$ *and* $P_{i+1} = P_i$,
- $P'_1 = P_{n+1} \cup \{f(u_0 \rhd u_1 \cdots u_n)\}$,

where $v \in T*$.

In both of leftmost and rightmost derivations, reflective transitive closure for them are denoted by $\overset{*}{\Rightarrow}_L$ and $\overset{*}{\Rightarrow}_R$, respectively.

These definitions of leftmost derivation and rightmost derivation denote the generality of the definition of derivation for RCFG. The occurrence of terminal strings in AF sequences may make someone feel that the definition is not sophisticated. The need of terminal strings, i.e. the fourth arguments of AFs, is clear, because, while embedded portions are objects of terminal strings, intermediate AF sequences contain syntactic variables as first arguments of AFs, which concern to the embedded portions, and production rules just augmented must be determined with these intermediate AF sequences. Some kind of circularity is needed in order to define self-extensibilities. The occurences of terminal strings in AFs are of such circularity. Our approach for definition of derivation is close to the stance of [6, 7], rather than ECL [30, 31].

The relations between general definition (Definition 3.1.5) and leftmost derivation (Definition 3.1.7) and between Definition 3.1.5 and rightmost derivation (Definition 3.1.8) are stated in the following section. As a consequence, they are quite similar to those of CFG.

Derivation tree on RCFG is also defined, like CFG. Derivation tree is not indispensable notion. However, it makes some properties on RCFG clear. To describe a derivation tree for given derivation sequence, we write AFs $\varphi_{i,1}$, ..., $\varphi_{i,n}$ derived from AF $\varphi_{k,l}$, where $\varphi_{i,j}$ is the AF which is derived from $\varphi_{k,l}$ and occurs at j-th position among AFs derived at i-th derivation. Initial AF is written by $\varphi_{0,1}$. For example, if $s \to a\ \ b \in P$, where s is the start variable and a and b are terminal symbols, a derivation $(s,\ P,\ P,\ a\,b) \Rightarrow (a,\ P,\ P,\ a)\ (b,\ P,\ P,\ b)$ is written by $\varphi_{0,1} \Rightarrow \varphi_{1,1}\ \varphi_{1,2}$, where $\varphi_{0,1} = (s,\ P,\ P,\ a\,b)$, $\varphi_{1,1} = (a,$

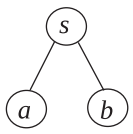


Figure 3.1: An Example of Derivation Tree

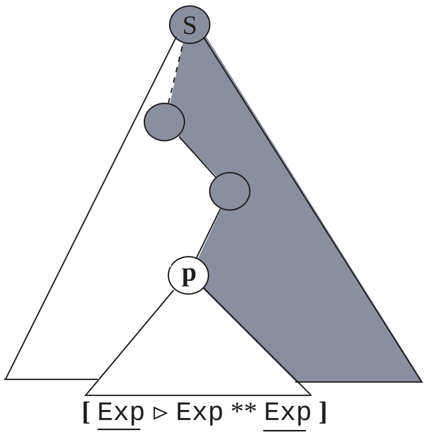


Figure 3.2: Effective Range of New Rule

P, P, a), $\varphi_{1,2} = (b,\ P,\ P,\ b)$. The set of nodes of the derivation tree is a finite subset of $\mathbf{N} \times \mathbf{N}$. (k, l) is the parent of $(i, 1)$, ..., (i, n) and (i, j) is the j-th child of (k, l). Each node of (i, j) is labelled with AF $\varphi_{i,j}$. When we illustrate a derivation tree in a graph, we may omit each arguments of AFs labelled except their first argumens. For example, a derivation tree of above example can be illustrated as Figure 3.1.

Effective range of newly added production rule is breafly illustrated in Figure 3.2. In meshed part of the derivation tree, the new production rule is available to be used for derivation.

Definition 3.1.9 (Language of RCFG)
For given RCFG $G = (V,\ T,\ M,\ D,\ \mathbf{Aug},\ f,\ P,\ s)$, the language $L(G)$ of G is defined as,

$$L(G) = \{u \in T* \mid \exists P', P_1, P_1' \in FSP,\ \exists w, w' \in T*,\ (s, P, P_1, w) \overset{*}{\Rightarrow} (u, P', P_1', w')\}.$$

Note: $u = w = w'$, $P = P'$ and $P_1 = P_1'$ are concluded with the propositions which will be shown below.

With definitions of derivation, leftmost (rightmost) derivation and language for RCFG, we can define a notion of 'ambiguity' for RCFG, similarly to CFG.

Definition 3.1.10 (Ambiguity) *For given RCFG G, if there is a word in $L(G)$, for which there are at least two distinct leftmost (rightmost) derivation sequences, G is called ambiguous grammar, or simply ambiguous.*

One may doubt the definition of ambiguity is too simple to state the notion, with supposition that there are two distinct derivation sequences in each step of which AF sequences are identical on their first arguments. It becomes clear that the supposition is not true by propositions stated in Section 3.3. The essential points are that a derivation for a variable $\mathbf{p} \in \mathbf{Aug}$ must cause an augmentation of a production rule and no augmentation occurs on ordinal cases on derivation, and, fourth arguments of each AFs are identical to strings derived from the AFs.

3.2 Examples

Here, two examples are given. One is an example for Operator Declaration Problem, in which a way how to give a grammar enables user to declare new operators is illustrated. In this example, the grammar illustrated is an ambigous in order to reduce descriptions. The other example is some what tricky one. In the later example, no production rules for start variable are given in initial production rule set. However, because a production rule for start variable can be augmented in texts, the language is not empty.

Example 3.2.1 (Operator declaration)
Consider RCFG $G_2 = (V, T, M, D, \mathbf{Aug}, f, P, s)$, where

$$\begin{aligned}
V &= \{s, \mathbf{p}, \mathit{DefList}, \mathit{Pat}, \mathit{Pat}_1, \mathit{Exp}, \mathit{Id}\},\\
T &= \{[, \rhd,], \underline{\mathit{Exp}}, a, b, \ldots, z, +, *\},\\
M &= D = \{\underline{\mathit{Exp}}\},\\
\mathbf{Aug} &= \{\mathbf{p}\},\\
f(\underline{\mathit{Exp}}) &= \mathit{Exp},\\
P &= \{s \to \mathit{DefList}\ \mathit{Exp},\\
&\quad \mathit{DefList} \to \varepsilon,\\
&\quad \mathit{DefList} \to \mathbf{p}\ \mathit{DefList},\\
&\quad \mathbf{p} \to [\underline{\mathit{Exp}} \rhd \mathit{Pat}],\\
&\quad \mathit{Pat} \to \underline{\mathit{Exp}}\,\mathit{Pat}_1 \mid \mathit{Pat}_1,\\
&\quad \mathit{Pat}_1 \to \mathit{Id}\,\underline{\mathit{Exp}} \mid \mathit{Id}\,\underline{\mathit{Exp}}\,\mathit{Pat}_1,\\
&\quad \mathit{Exp} \to \mathit{Id} \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} * \mathit{Exp},\\
&\quad \mathit{Id} \to a \mid b \mid \cdots \mid z \mid a\,\mathit{Id} \mid b\,\mathit{Id} \mid \cdots \mid z\,\mathit{Id}\}.
\end{aligned}$$

We illustrate a derivation sequence for a string "[$\underline{Exp}$ $\rhd$ exponential $\underline{Exp}$ of $\underline{Exp}$] a + exponential b of c". G_2 is an ambiguous, because of the definition of production rule concerning to variable Exp. We start with AF (s, P, P', u) for some appropriate P' and u.

$$\begin{aligned}
(s, P, P', u) &\Rightarrow (\mathit{DefList}\,\mathit{Exp}, P, P', u)\\
&\Rightarrow (\mathbf{p}, P, P_1, u_1)(\mathit{DefList}, P_1, P_2, u_2)(\mathit{Exp}, P_2, P', u_3)\\
&\Rightarrow (\mathbf{p}, P, P_1, u_1)(\varepsilon, P_1, P_2, u_2)(\mathit{Exp}, P_2, P', u_3)
\end{aligned}$$

(at this point, it becomes clear that $P_1 = P_2$, $u_2 = \varepsilon$. AFs with ε string are omitted from now on,)

$$\Rightarrow ([\underline{\mathit{Exp}} \rhd \mathit{Pat}], P, P_1, u_1)(\mathit{Exp}, P_2, P', u_3)$$

$$\overset{*}{\Rightarrow} \ ([\underline{Exp} \rhd \mathit{exponential}\, \underline{Exp}\ \mathit{of}\, \underline{Exp}], P, P_1, u_1)(Exp, P_2, P', u_3)$$

(at this point, it becomes clear that $P_1 = P_2 = P \cup \{Exp \to \mathit{exponential}\, Exp\ \mathit{of}\, Exp\}$*, and* $u_1 =$ *"*$[\ \underline{Exp} \rhd \mathit{exponential}\ \underline{Exp}\ \ \mathit{of}\ \underline{Exp}]''$*,)*

$$\begin{aligned}
&\Rightarrow \ (u_1, P, P_1, u_1)(Exp + Exp, P_2, P', u_3)\\
&\overset{*}{\Rightarrow} \ (u_1, P, P_1, u_1)(a + Exp, P_2, P', u_3)\\
&\Rightarrow \ (u_1, P, P_1, u_1)(a + \mathit{exponential}\, Exp\ \mathit{of}\, Exp, P_2, P', u_3)
\end{aligned}$$

(this derivation is enabled, because a production rule $Exp \to \mathit{exponential}\ Exp\ \ \mathit{of}\ Exp$ *is in* P_2*, which was augmented above derivation,)*

$$\begin{aligned}
\overset{*}{\Rightarrow} \ &([\underline{Exp} \rhd \mathit{exponential}\, \underline{Exp}\ \mathit{of}\, \underline{Exp}]a + \mathit{exponential}\ b\ \mathit{of}\ c, P, P',\\
&[\underline{Exp} \rhd \mathit{exponential}\, \underline{Exp}\ \mathit{of}\, \underline{Exp}]a + \mathit{exponential}\ b\ \mathit{of}\ c).
\end{aligned}$$

Example 3.2.2 (Tricky)
Here, an example which provides a reflective feature of RCFG is given. Consider RCFG $G_3 = (V, T, M, D, \mathbf{Aug}, f, P, s)$*, where*

$$\begin{aligned}
V &= \{s, \mathbf{p}, Pat\},\\
T &= \{[, \rhd,], \underline{s}, \underline{\mathbf{p}}, a, b\},\\
M &= \{\underline{\mathbf{p}}, \underline{s}\}\\
D &= \{\underline{s}\},\\
\mathbf{Aug} &= \{\mathbf{p}\},\\
f(\underline{\mathbf{p}}) &= \mathbf{p}, f(\underline{s}) = s,\\
P &= \{\mathbf{p} \to [\underline{s} \rhd Pat],\\
&\quad\ Pat \to \underline{\mathbf{p}} \mid a\, Pat \mid Pat\, a \mid b\, Pat \mid Pat\, b\}.
\end{aligned}$$

Initial production rule set P *does not contain any rule for start variable* s*. Thus, in the sense of CFG, all rules contained in initial production rule set* P *are nullable. However, this grammar states a language,*

$$L(G_3) = \{w_1[\underline{s} \rhd w_1\underline{\mathbf{p}}w_2]w_2 \mid w_1, w_2 \in \{a, b\}*\}.$$

How can we guess so? See the definition of derivation for RCFG (Definition 3.1.5) again. On case 1), i.e. the ordinal case, derivation on a syntactic variable A *is defined with a production rule* $A \to X_1 \cdots X_n$ *contained in a production rule set* P_{n+1}*, while* P_{n+1} *is fixed during derivations on* $X_1, \ldots, X_n$*. Is it circular definition, which leads to a contradiction? Actually, it leads to no contradiction. Now, we observe a derivation sequence reaches to a string '*$[\underline{s} \rhd \underline{\mathbf{p}}]$*'. From the initial AF* (s, P, P', u)*, there are possibly infinite candidates of AF sequences as a result of derivation by one step,*

$$(s, P, P', u) \Rightarrow (X_1, P, P_1, u_1) \cdots (X_n, P_{n-1}, P_n, u_n).$$

However, because $s \notin \mathbf{Aug}$*,* $(X_1, P, P_1, u_1) \cdots (X_n, P_{n-1}, P_n, u_n)$ *and* $s \to X_1 \cdots X_n$ *must satisfy the constraints given in the case 1) of Definition 3.1.5. One of the constraints is* $s \to X_1 \cdots X_n \in P_{n+1}$ *that is not in* P*. Hence, there must exist at least one AF sequence* *derived from $(X_1, P, P_1, u_1) \cdots (X_n, P_{n-1}, P_n, u_n)$, which contains an AF for $\mathbf{p}$. There is one choice '$\mathbf{p}$' for $X_1 \cdots X_n$ among infinite, possibly useless, candidates. As a hindsight, this choice leads to a success on the derivation, because from the AF for $\mathbf{p}$, an AF sequence for a string '$[\underline{s} \rhd \underline{\mathbf{p}}]$' is derived, and a new production rule $s \rightarrow \mathbf{p}$ is augmented to current production rule set P.*

Why does the example seem to be confused? One of answers might be that the notion of derivation is congenial to top down scheme, and augmentation of a new rule congenial to bottom up scheme. A parsing process of this example will be given after the definition of general parsing algorithm (Algorithm 3.4.2).

3.3 Properties on RCFG

Following arguments are basic properties on RCFG. Proposition 3.3.1 to 3.3.4 and 3.3.6 are used to establish soundness of Algorithm 3.4.2, Theorem 3.3.7 is used in the discussion on the efficiency of parsing algorithm, and Theorem 3.3.15 gives a basis for the proof of completeness of Algorithm 3.4.2. From these properties, it is able to see that RCFG is quite a simple extension of CFG, and the formalism is free from procedural arguments.

Proposition 3.3.1 *If $(\alpha, P_1, P_2, w) \overset{*}{\Rightarrow} (\beta, P'_1, P'_2, w')$ then $P_1 = P'_1$, $P_2 = P'_2$.*

(proof) By induction on length of derivations. If length is zero, trivial. For one step of derivation, there are three cases; a derivation occurs at leftmost-side, inner-point or rightmost-side of the AF sequence. In all cases, it is straightforward from the definition of derivation. //

Proposition 3.3.2 *If $(A, P_1, P_2, w) \overset{*}{\Rightarrow} (\alpha, P_1, P'_1, w')(\beta, P'_2, P_2, w'')$ then $P'_1 = P'_2$, $\forall A \in V$.*

(proof) By induction on length of derivations. If length is zero, trivial. For one step of derivation, it is straightforward from the definition on derivation. It is easy to see that the rest case holds, by use of Proposition 3.3.1. //

Proposition 3.3.3 *If $(\alpha, P_1, P_2, w) \overset{*}{\Rightarrow} (\beta,\ P'_1,\ P'_2,\ w')$ then $w = w'$.*

(proof) By induction on length of derivations. If length is zero, trivial. For one step, it is straightforward from the definition of derivation. //

Proposition 3.3.4 *If $(\alpha,\ P_1,\ P_2,\ w) \overset{*}{\Rightarrow} (u,\ P'_1,\ P'_2,\ w')$ then $u = w = w'$.*

(proof) By induction on length of derivations with Proposition 3.3.3. //

Corollary 3.3.5 *If $(A,\ P_1,\ P_2,\ w) \overset{*}{\Rightarrow} (u,\ P'_1,\ P'_2,\ w')$ for $A \in V$ and $u \in T*$ then $P_1 = P'_1$, $P_2 = P'_2$, $u = w = w'$.*

Proposition 3.3.6 *If* $(A, P_1, P_2, w) \overset{*}{\Rightarrow} (\alpha, P_1, P_1', w')\ (\beta, P_1', P_2, w'') \overset{*}{\Rightarrow} (w, P_1, P_2, w)$ *then* $P_1 \subset P_1' \subset P_2$*, for any* $A \in V$*. Moreover, if* P_1 *is properly included by* P_2*,* w *contains at least one embedded portion.*

(proof) By induction on length of derivations. If $(A, P_1, P_2, w) \Rightarrow (w, P_1, P_2, w)$ by one step, it is obvious; only on the case that $A = \mathbf{p} \in \mathbf{Aug}$ and w is an embedded portion, P_2 properly includes P_1. Under induction hypothesis, if $(A, P_1, P_2, w) \overset{*}{\Rightarrow} (\alpha, P_1, P_1', w')\ (\beta, P_1', P_2, w'')$ and α and β are derivable to w' and w'' respectively, then, from Proposition 3.3.2, also the the induction hypothesis holds. From the constraint of case 1), Definition 3.1.5, i.e., about the case $u_i = X_i$, it is obvious that if w contains no embedded portion, then $P_1 = P_2$. The contrary is also easy to show. //

Theorem 3.3.7 *If* $(A, P_1, P_2, u) \overset{*}{\Rightarrow} (u, P_1, P_2, u)$*, then* u *is also a word of the language of CFG* $G = (V, T, P_2, A)$*.*

(proof) For any step of derivations of $(A, P_1, P_2, u) \overset{*}{\Rightarrow} (u, P_1, P_2, u)$, it is easy to see that if $(\alpha, P_1, P_2, u) \Rightarrow (\beta, P_1, P_2, u)$ on RCFG holds, $\alpha \Rightarrow \beta$ on CFG G holds, from Proposition 3.3.6. //

Theorem 3.3.8 *The language class of RCFG properly includes the language class of CFG.*

(proof) If all words of given RCFG G contains no embedded portion, i.e., there is no derivation s.t. $(s, P, P', u) \overset{*}{\Rightarrow} (\alpha, P, P_1, u_1)\ (\mathbf{p}, P_1, P_2, u_2)\ (\beta, P_2, P', u_3)$, where $\mathbf{p} \in \mathbf{Aug}$, or, there is no derivation s.t. $(\mathbf{p}, P, P', u) \overset{*}{\Rightarrow} (u, P, P', u)$, then, from Proposition 3.3.6 and Theorem 3.3.7, the language of G is identical to the language of CFG (V, T, P, s). So, all languages of CFG are included in the language class of RCFG.

Now, we consider a RCFG G_1 in Example 3.1.6. $L(G_1) = \{[\underline{A} \triangleright w]w \mid w \in (T \backslash \{[,]\})*\}$. $L(G_1)$ is not CFL from the pumping theorem on CFG [14]. So, the language class of CFG is properly included by the language class of RCFG. //

Lemma 3.3.9 (Folding)
For any given RCFG G, $L(G)$ *is preserved after replacing a rule* $A \to \alpha X_1 X_2 \beta$ *with a new rules* $A \to \alpha H \beta$ *and* $H \to X_1 X_2$*, if* $X_1 \neq [$ *and* $X_2 \neq]$*, where* H *is a newly added syntactic variable* $\notin \mathbf{Aug}$.

(proof) It is almost identical to the discussion on CFG. //

Lemma 3.3.10 (Elimination of ε-rule)
For any given RCFG which language does not contain ε*(*ε*-free), there exists a RCFG* G' *which rule contains no* ε*-rule, and which language is identical to that of* G*.*

(proof) It is almost identical to the discussion on CFG. //

Lemma 3.3.11 (Normal Form of RCFG)
For any given ε-free RCFG G, there exists a RCFG G' which language is identical to that of G' , and moreover, of which each rule forms one of following three cases,

1. *$A \to a$, where $A \in V$ and $a \in T$*
2. *$A \to B\,C$, where $A, B, C \in V$*
3. *$\mathbf{p} \to [B_1 \rhd B_2]$, where $\mathbf{p} \in \mathbf{Aug}$ and B_1, $B_2 \in V$*

(proof) Same as the discussion on CFG. //

Note: ε-free RCFG means in same sense of CFG, which does not contain any production rules that are derived to ε, and moreover, does not accept production rules which have ε on right-hand side , for newly added production rules.

Theorem 3.3.12 *The language class of ε-free RCFG is properly included by that of CSG (Context-Sensitive Grammars).*

(proof) First, we sketch a construction strategy of CSG G' which language is identical to given ε-free RCFG G which has Normal Form.

1) All production rules of G with some translations are contained in G'.

2) G' has production rules for seeking an embedded portion positioning in left of focusing position, and then replace a syntactic variable due to found embedded portion.

On constructed CSG G', production sequences of RCFG G are emulated non-deterministically due to above two cases.

We finish this theorem with a tedious example which is contained in CSL, but in RCFL, i.e., $\{w[A \rhd w] \mid w \in (T \setminus \{[,]\})*\}$. //

Finally, we will guess on leftmost derivation and rightmost derivation. It is trivial from the definition of leftmost (rightmost) derivation that if there is a leftmost (rightmost) derivation sequence, we can consider the derivation sequence is merely a derivation of RCFG. Following propositions argue that if there is a derivation sequence, there exists a leftmost (rightmost) derivation sequence, and moreover, derivation trees corresponding to both derivation sequences are mutually isomorphic, where the isomorphism is identical to that of Graph Theory [5, 26] etc. The proof of completeness of general parsing algorithm (Algorithm 3.4.2 in Section 3.4.1) depends on the discussion of Theorem 3.3.15. Following arguments are not difficult at all, but important.

To establish this property, we start with a 'spliting' of derivation sequences.

Lemma 3.3.13 (Sub-sequence of Derivation) *If there is a derivation sequence*

$$(\alpha, P_0, P_1, w_1)(\beta, P_1, P_2, w_2)(\gamma, P_2, P_3, w_3) \overset{*}{\Rightarrow} (\alpha', P_0, P_1, w_1)(\beta', P_1, P_2, w_2)(\gamma', P_2, P_3, w_3),$$

then there exists a derivation

$$(\beta, P_1, P_2, w_2) \overset{*}{\Rightarrow} (\beta', P_1, P_2, w_2).$$

(proof) By induction on length of derivations.//

Lemma 3.3.14 (Concatenation of Derivation) *If there are derivation sequences,*

$$(\alpha, P_0, P_1, w_1) \overset{*}{\Rightarrow} (\alpha', P_0, P_1, w_1)$$

and

$$(\beta, P_1, P_2, w_2) \overset{*}{\Rightarrow} (\beta', P_1, P_2, w_2),$$

then there exists a derivation sequence, such as,

$$(\alpha, P_0, P_1, w_1)(\beta, P_1, P_2, w_2) \overset{*}{\Rightarrow} (\alpha', P_0, P_1, w_1)(\beta', P_1, P_2, w_2).$$

(proof) Straightforward from the definition of derivation.//

Theorem 3.3.15 *If there is a derivation sequence*

$$(\alpha, P_0, P_1, w) \overset{*}{\Rightarrow} (w, P_0, P_1, w),$$

then there exists a leftmost derivation sequence, such as,

$$(\alpha, P_0, P_1, w) \overset{*}{\Rightarrow}_L (w, P_0, P_1, w),$$

and also, there exists a rightmost derivation sequence, such as,

$$(\alpha, P_0, P_1, w) \overset{*}{\Rightarrow}_R (w, P_0, P_1, w).$$

(proof) By induction on length of AF sequences and length of derivations on them. Firstly, we devide AF sequence (α, P_0, P_1, w) into two parts (α_1, P_0, P_1', w_1) and (α_2, P_1', P_1, w_2), where $\alpha = \alpha_1\alpha_2$ and $w = w_1w_2$. Then, applying Lemma 3.3.13 and induction hypothesis on each parts, and applying Lemma 3.3.14 to the result of previous application, we obtain corresponding leftmost derivatioin sequence. On rightmost derivation, reversing the application order, we also obtain corresponding rightmost derivation sequence.//

3.4 General Parsing Algorithm for RCFG

Here, we discuss on a parsing algorithm for RCFG. The parsing algorithm is grounded on the arguments of Proposition 3.3.2 and Corollary 3.3.5. The difference between this algorithm and original Earley's parsing algorithm is mostly on items. Items are data dealt in algorithms, which intuitively denotes a point on input text where the algorithm is going on. On parsing of RCFG text, items are augmented with a finite set of production rules which might be increased during parsing.

3.4.1 Algorithm

We must firstly note that in the formalism of RCFG, "self-definitions" of any enbedded portions are not enabled. The definition of derivation requires on reflective case that a variable $\mathbf{p} \in \mathbf{Aug}$ which generates embedded portions has production rules which forms $\mathbf{p} \rightarrow [\alpha]$ for generating embedded portions. The important point is '[' and ']' present directly on the right-hand side of production rules on $\mathbf{p}$. To define self-definition on a variable $\mathbf{p}$, any embedded portion for the purpose must form "$[\underline{\mathbf{p}} \rhd \alpha]$" for some string α, and α must be equal to the whole string. Thus, to define self-definition in RCFG formalism, it is need an infinite length string. This feature enables us to define general parsing algorithm for RCFG in quite simple manner. And the feature is implicitly used in the proofs of theorems below, which shows soundness and completeness of the algorithm.

Definition 3.4.1 (item for parsing) *An item is given as 5-tuple (Rule, Scanned, Rest, Pre-Rules, Augmented-Rules), each elements are as following; Rule is a production rule which might be used to produce input text. We assume that Rule forms $A \rightarrow \alpha\beta$, where $A \in V$ and $\alpha, \beta \in (V \cup T)*$. Scanned is a left portion of right-hand side of Rule, i.e., equal to α, which denotes a portion in input text consumed during parsing so far. Rest is a right portion of right-hand side of Rule, i.e., equal to β, which would be scanned from now on. Pre-Rules denotes a finite set of production rules, which is ascertained at the time when the item arises in use on parsing. Augmented-Rules denotes a finite set of production rules, which might be augmented with rules associated with embedded portions appeared so far.*

We also adopt 'dot notation' to represent items. For example $(A \rightarrow \alpha \bullet \beta,\ R_1,\ R_2)$ is identical to $(A \rightarrow \alpha\beta,\ \alpha,\ \beta,\ R_1,\ R_2)$. From now on, items are represented with 3-tuples.

Algorithm 3.4.2 (General Parsing Algorithm for RCFG)
When an RCFG $G = (V, T, M, D, \mathbf{Aug}, f, P, s)$ and n-length input text $a_1 \cdots a_n$ are given, parse lists $I(0,0), \ldots, I(i,j), \ldots, I(n,n) (0 \leq i \leq j \leq n)$ are calculated during parsing, where each element of $I(i,j)$ consists of items. Additionally, finite sets of production rules $P_0, \ldots, P_n$ are constructed, where P_i holds a maximum set of production rules possible at the point of i-th input character (and ancestors of it in the derivation tree).

Initial phase:

1) *initialize all of $P_0, \ldots, P_n$ to P,*

2) *initialize all of $I(i,j)$ to ϕ,*

3) *add $(X \rightarrow \bullet\alpha, P, P)$ to $I(i,i)$, for each $X \rightarrow \alpha \in P$ and each $i = 0, \ldots, n$,*

Main Loop:
repeat 4), 5), 6), 7) until no new 3-tuple is added to any $I(i,j)$

4) *if $(X \rightarrow \alpha \bullet a\beta,\ R_1,\ R_2) \in I(i,j)$ and $a_{j+1} = a$, and moreover $X \notin \mathbf{Aug}$ or $\beta \neq \phi$, then add $(X \rightarrow \alpha a \bullet \beta, R_1, R_2)$ to $I(i, j+1)$,*

5) *if $(X \rightarrow [\alpha \bullet],\ R_1,\ R_2) \in I(i,j)$, $a_{j+1} =]$, $X \in \mathbf{Aug}$ and $f(a_{i+2} \ldots a_j)$ has valid form of production rule, then add $(X \rightarrow [\alpha]\bullet,\ R_1,\ R_2 \cup \{f(a_{i+2} \ldots a_j)\})$ to $I(i, j+1)$,*

6) *if* $(Y \to \gamma\bullet, R_2, R_3) \in I(j,k)$ *and* $Y \to \gamma \in R_3$ *and* $(X \to \alpha \bullet Y\beta, R_1, R_2) \in I(i,j)$ *for some* $0 \leq i \leq j \leq k \leq n$*, then add* $(X \to \alpha Y \bullet \beta, R_1, R_3)$ *to* $I(i,k)$*,*

7) *if* $(\mathbf{p} \to \alpha\bullet, R_1, R_2) \in I(i,j)$ *and* $\mathbf{p} \in \mathbf{Aug}$ *and* $\mathbf{p} \to \alpha \in R_2$ *and* $[w] = a_{i+1} \cdots a_j$ *and* $f(w)$ *has valid form of production rule, then let* $Z \to \gamma = f(w)$*, add* $Z \to \gamma$ *to* $P_j, \ldots, P_n$ *and then add* $(Y \to \bullet\beta, P', P')$ *to* $I(k,k)$ *for each* $k = 0, \ldots, n$*, each* $Y \to \beta \in P_j$ *and each* P' *s.t.* $P_0 \subset P' \subset P_k$

Judgement:

8) *if* $(s \to \alpha\bullet, P, P') \in I(0,n)$ *and* $s \to \alpha \in P'$ *for some set of production rules* P'*, then accept input else reject input.*

Note: On operation 7), only one element $(Z \to \bullet\gamma, P_k, P_k)$ is newly added to $I(k,k)$ for each $k = 0, \ldots, j-1$, where $Z \to \gamma$ is augmented production rule at that point. On operation 8), there are valid cases that $P' \neq P_n$, on those cases, some actions on 5) and 7) have occurred, but some of them would not be used in effective derivations.

Figure 3.3 and 3.4 illustrate a parsing process for input $[\ \underline{s} \rhd \underline{\mathbf{p}}\ ]$ and the grammar G_3 given in Example 3.2.2. Figure 3.3 is the snapshot at the point when initial phase is completed, and Figure 3.4 at the point when production rule $s \to \mathbf{p}$ is augmented, which is caused by the embedded portion $[\ \underline{s} \rhd \underline{\mathbf{p}}\ ]$. In both of figures, items which do not concern to the acceptance of the input are omitted. From the item ($Pat \to \bullet\underline{\mathbf{p}}$, P, P) in parse list $I(3,3)$, which is added by operation 3) in initial phase, an item ($Pat \to \underline{\mathbf{p}}\bullet$,$P$, P) is added to $I(3,4)$ by operation 4). Also, items ($\mathbf{p} \to [\bullet\, \underline{s} \rhd Pat]$, P, P), $\cdots$, ($\mathbf{p} \to [\underline{s} \rhd Pat\bullet]$, P, P) are successively added to parse list $I(0,1)$,$\cdots$,$I(0,4)$, from item ($\mathbf{p} \to \bullet[\underline{s} \rhd Pat]$, P, P) in $I(0,0)$ by operation 4), and finally an item ($\mathbf{p} \to [\underline{s} \rhd Pat]\bullet$, P, P') is added to $I(0,5)$ by operation 5), where $P' = P \cup \{s \to \mathbf{p}\}$. This operation 5) causes operation 7) and an augmentation of a production rule $s \to \mathbf{p}$ to P_5, and moreover, the addition of an item ($s \to \bullet\mathbf{p}$, P, P) to $I(0,0)$. Finally, from this item and the item included in $I(0,5)$, we obtain an item ($s \to \mathbf{p}\bullet$, P, P') in $I(0,5)$ by operation 6), and conclude the input is accepted by judgement 8).

3.4.2 Soundness and Completeness

Theorem 3.4.3 (Soundness) *If* ($A \to \alpha\bullet\beta$, R_1, R_2) $\in I(i,j)$*, then a derivation relation* (A, R_1, R', u) $\overset{*}{\Rightarrow}$ ($a_{i+1} \ldots a_j$, R_1, R_2, $a_{i+1} \ldots a_j$) (β, R_2, R', u') *holds.*

(proof) By induction on times of actions 4), 5) and 6) to obtain the element ($A \to \alpha \bullet \beta$, R_1, R_2) $\in I(i,j)$. On the case of $i = j$, i.e the case $\mid \alpha \mid = 0$, the proposition holds vacuously. We assume that $\alpha = \alpha' X$.

On the case $X \in T$ and $A \notin \mathbf{Aug}$ or $\beta \neq \phi$, from the definition of the algorithm of operation 4), ($A \to \alpha' \bullet X\beta$, R_1, R_2) must be involved in $I(i, j-1)$ and $X = a_j$. Using induction hypothesis, there is a derivation (A, R_1, R', u) $\overset{*}{\Rightarrow}$ ($a_{i+1} \ldots a_{j-1}$, R_1, R_2, $a_{i+1} \ldots a_{j-1}$) ($X\beta$, R_2, R', u'') = ($a_{i+1} \ldots a_{j-1}$, R_1, R_2, $a_{i+1} \ldots a_{j-1}$) (X, R_2, R_2, a_j) (β, R_2, R', u') = ($a_{i+1} \ldots a_j$, R_1, R_2, $a_{i+1} \ldots a_j$) (β, R_2, R', u').

On the case $X = \,]$ and $A \in \mathbf{Aug}$ and moreover $f(a_{i+2} \ldots a_{j-1})$ has valid form of production rule. This case concerns to the operation 5) of the algorithm. On this case, we must be careful on the augmentation of a new production rule. From the definition

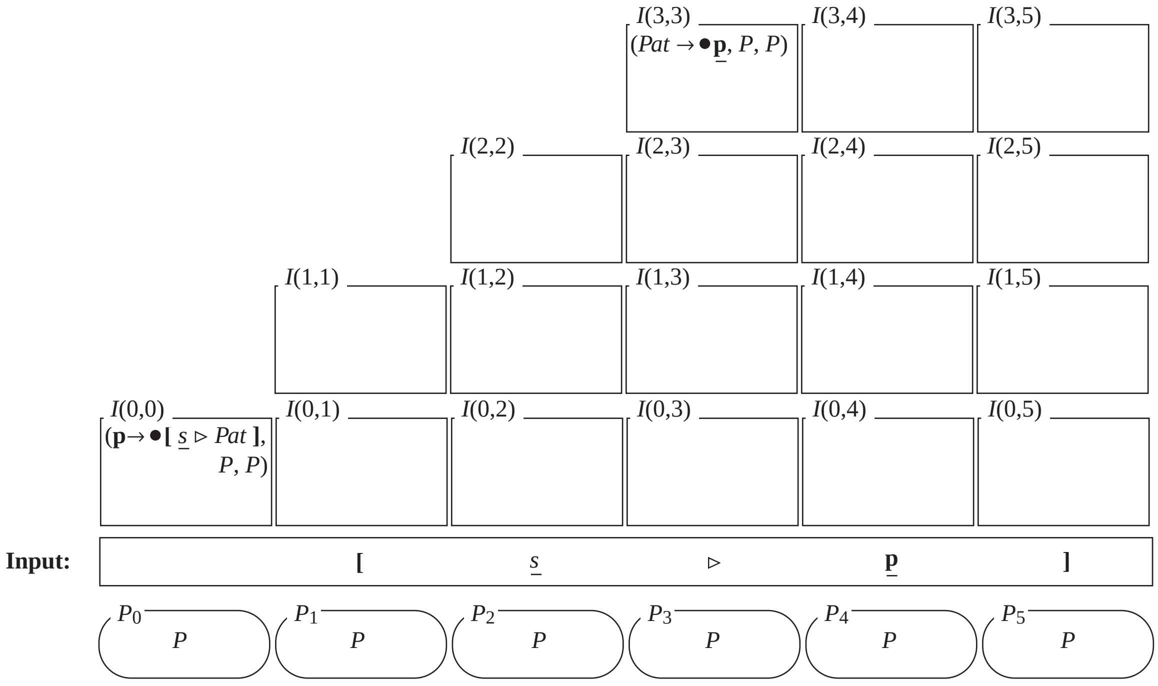


Figure 3.3: Initial Parse Lists for Input [$\underline{s} \rhd \underline{\mathbf{p}}$]

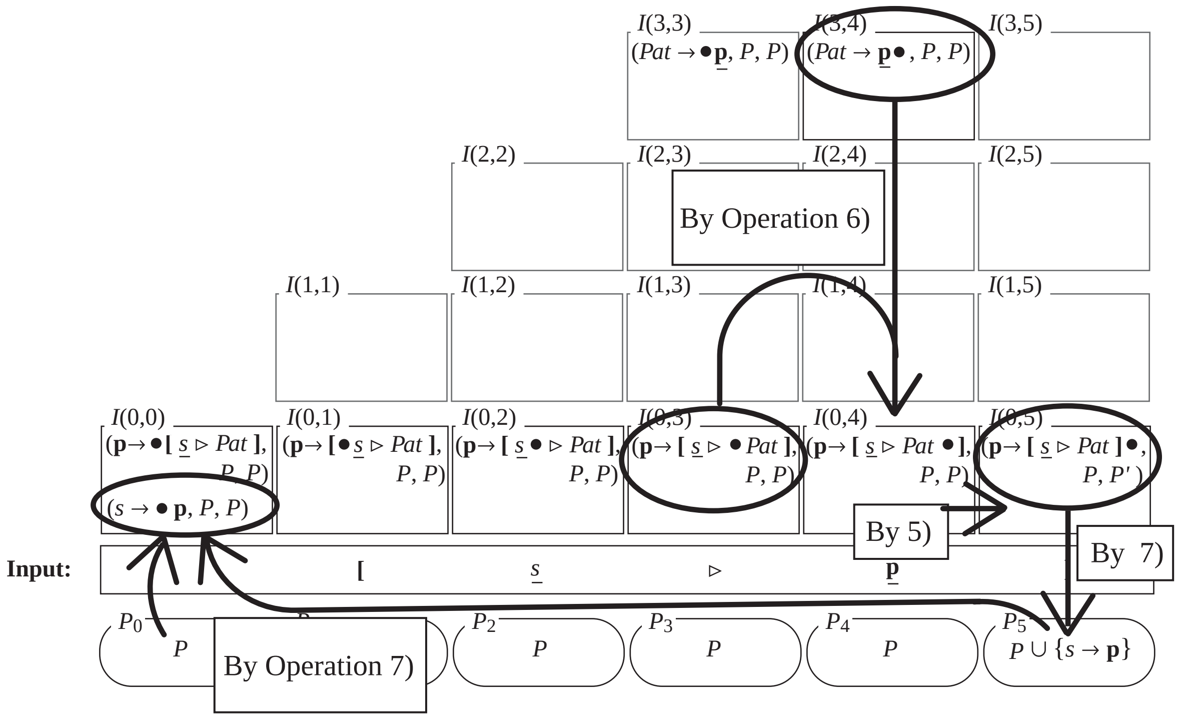


Figure 3.4: Parse Lists at Augmentation of $s \rightarrow \mathbf{p}$

of the algorithm of operation 5), $(A \to \alpha' \bullet]\beta,\ R_1,\ R_2')$ must be involved in $I(i, j-1)$, for appropriate R_2'. Using induction hypothesis, there is a derivation $(A,\ R_1,\ R',\ u) \overset{*}{\Rightarrow} (a_{i+1} \ldots a_{j-1},\ R_1,\ R_2',\ a_{i+1} \ldots a_{j-1})$ $([,\ R_2',\ R_2,\ ])$ $(\beta,\ R_2,\ R',\ u'')$ and R_2 must equal to $R_2' \cup \{f(a_{i+2} \ldots a_{j-1})\}$. This derivation is also a candidate of derivations which concerns to the element $(A \to \alpha \bullet \beta,\ R_1,\ R_2)$. So, on this case, induction hypothesis also holds.

On the case $X \in V$, from the defintion of the algorithm of operation 6), $(A \to \alpha' \bullet X\beta,\ R_1,\ R_3)$ must be involved in $I(i,k)$ and also $(X \to \gamma,\ R_3,\ R_2)$ must be involved in $I(k,j)$. Using induction hypothesis, there are derivations $(A,\ R_1,\ R',\ u) \overset{*}{\Rightarrow} (a_{i+1} \ldots a_k,\ R_1,\ R_3,\ a_{i+1} \ldots a_k)$ $(X\beta, R_3, R', u'')$ and $(X, R_3, R_2, a_{k+1} \ldots a_j) \overset{*}{\Rightarrow} (a_{k+1} \ldots a_j, R_3, R_2, a_{k+1} \ldots a_j)$. To conbine these derivations, we obtain the result.//

Corollary 3.4.4 *If w is accepted by the Algorithm, then w is a word of given RCFG G.*

To establish completeness of the algorithm, we restrict derivations to leftmost ones.

Theorem 3.4.5 (Completeness) *Suppose there exists a leftmost derivation* $(\gamma_1,\ P,\ R_1,\ u_1)$ $(A,\ R_1,\ R_3,\ w_1w_2)$ $(\gamma_2,\ R_3,\ P',\ u_2) \overset{*}{\Rightarrow} (u_1,\ P,\ R_1,\ u_1)$ $(A,\ R_1,\ R_3,\ w_1w_2)$ $(\gamma_2,\ R_3,\ P',\ u_2) \Rightarrow (u_1,\ P,\ R_1,\ u_1)$ $(\alpha,\ R_1,\ R_2,\ w_1)$ $(\beta,\ R_2,\ R_3,\ w_2)$ $(\gamma_2,\ R_3,\ P',\ u_2) \overset{*}{\Rightarrow} (u_1,\ P,\ R_1,\ u_1)(w_1,\ R_1,\ R_2,\ w_1)$ $(w_2,\ R_2,\ R_3,\ w_2)$ $(\gamma_2,\ R_3,\ P',\ u_2)$. *Then for some input $a = u_1w_1w_2u_2$, an element $(A \to \alpha \bullet \beta,\ R_1',\ R_2)$ is in $I(\mid u_1 \mid, \mid u_1 \mid + \mid w_1 \mid)$.*

(proof) By induction on pair of length of derivations and frequency of augmentation of production rules. In following, (l, m) denotes the case of l-length derivation sequence which includes or assumes m times augmentations of new rules. There is partial order between (l, m) and (l', m'), $(l, m) < (l', m')$ iff $m < m'$ or $m = m'$ and $l < l'$. In the following descriptions, we use concatenation on derivations implicitly. The supposition of leftmost derivation is needed to hold the induction hypothesis, mostly on frequencies of augmentations.

Vacuous cases are that for any production rules $X \to \alpha$ in initial production rule set P, $(X \to \bullet\alpha,\ P,\ P)$ is in $I(k,k)$, for each $k = 0, \ldots n$, each of which corresponds to 1-length derivation,

$$(X, P, R_3, w_2) \Rightarrow (\alpha, P, R_3, w_2)$$

for some appropriate R_3 and w_2. On the cases, it is trivial that the induction hypothesis holds by initialization of the algorithm. Suppose $\alpha = v\alpha'$ for some $v = b_1 \cdots b_i \in T*$. On cases if $\alpha' \neq \varepsilon$ or $v \neq v']$ or $X \notin \mathbf{Aug}$, and moreover, input string $a_1 \cdots a_n$ contains $b_1 \cdots b_i$ as a sub-string starts from $k+1$-th position, from the item $(X \to \bullet\alpha,\ P,\ P)$, we obtain items $(X \to b_1 \bullet b_2 \cdots b_i\alpha',\ P,\ P) \in I(k, k+1)$,..., $(X \to b_1 \cdots b_i \bullet \alpha',\ P,\ P) \in I(k, k+i)$ successively, using operation 4). Thus, the induction hypothesis on the induction case $(l, m) = (1, 0)$ holds.

Suppose that as an ordinal case of derivation, there is a derivation $(A,\ R_1,\ R_3,\ w_1w_2w_3)$ $\Rightarrow (w_1,\ R_1,\ R_1,\ w_1)$ $(B,\ R_1,\ R_2,\ w_2)$ $(\beta,\ R_2,\ R_3,\ w_3) \overset{*}{\Rightarrow} (w_1,\ R_1,\ R_1,\ w_1)$ $(w_2,\ R_1,\ R_2,\ w_2)$ $(w_3,\ R_2,\ R_3,\ w_3)$ on the case that $A \notin \mathbf{Aug}$. We assume that the derivation sequence from $(B,\ R_1,\ R_2,\ w_2)$ to $(w_2,\ R_1,\ R_2,\ w_2)$ is under the induction case (l_2, m_2) and derivation sequences from (β, R_2, R_3, w_3) to (w_3, R_2, R_3, w_3) is under the induction case (l_3, m_3), where $m_2 \leq m_3$. From the induction hypothesis and from the fact that $A \to w_1 B\beta \in R_3$ by the constraint of derivation, we can claim that the item ($A \to w_1 \bullet B\beta$, R_1, R_1) is in appropriate item list as the induction case $(1, m_3)$, considering operation 7). And also, we can claim that ($A \to w_1 B \bullet \beta$, R_1, R_2) is in appropriate item list as the induction case $(l_2 + 1, m_3)$, considering operation 6). By same discussion, induction proceeds so that an item ($A \to w_1 B\beta\bullet$, R_1, R_3) is in appropriate item list as the induction case ($max\{l_2, l_3\} + 1$, m_3).

Suppose that as a reflective case of derivation, there is a derivation ($\mathbf{p}$, R_1, R_3, $[w_2]$) $\Rightarrow$ ([, R_1, R_1,]) (β, R_1, R_2, w_2) (], R_2, R_3,]) $\overset{*}{\Rightarrow}$ ([, R_1, R_1,]) (w_2, R_1, R_2, w_2) (], R_2, R_3,]) on the case that $\mathbf{p} \in \mathbf{Aug}$ and $f(w_2)$ has valid form for a production rule. We assume that the derivation sequence from (β, R_1, R_2, w_2) to (w_2, R_1, R_2, w_2) is under the induction case (l_2, m_2). From the induction hypothesis and from the fact that $\mathbf{p} \to [\beta]$ $\in R_3$ by the constraint of derivation, items ($\mathbf{p} \to \bullet[\beta]$, R_1, R_1), ..., ($\mathbf{p} \to [\beta \bullet]$, R_1, R_2) are in appropriate item lists as the induction cases $(1, m_2)$ to $(l_2 + 1, m_2)$, respectively. By operation 5), an item ($\mathbf{p} \to [\beta]\bullet$, R_1, R_3) is added to appropriate item list as the induction case $(l_2 + 1, m_2 + 1)$, where $R_3 = R_2 \cup \{f(w_2)\}$, and induction proceeds. And more over, items ($Z \to \bullet\gamma$, R', R') and ($X \to \bullet\alpha$, R_3, R_3) are added to appropriate item list by operation 7) as the induction case $(1, m_2 + 1)$, where $Z \to \gamma = f(w_2)$, $P \subset R' \subset P_k$ and $X \to \alpha \in P_k$ for corresponding k.//

Corollary 3.4.6 *If w is a word of given RCFG G, then w is accepted by the algorithm.*

3.4.3 Discussion on Complexity

If we ignore costs of seeking a production rule, which are done at 6), and costs of matching between sets of production rules, which are done at 6) and 7), and also if no new rules are added, from Theorem 3.3.7, whole cost of parsing is similar to Earley's parsing algorithm [11]. Because seeking or matching of production rules is done for every items on operation 6), and number of items which would be contained in $\bigcup_{i,j} I(i,j)$ at the end of the algorithm is proportional to n^2 on worst case, we can conclude that the complexity of the algorithm is $O(n^5)$.

If embedded portions are contained in input texts, where the number of embedded portions is denoted by m, the number of items contained in $\bigcup_{i,j} I(i,j)$ is roughly 2^m times of above case, because, on operation 7), there are 2^m candidates for selecting P' which satisfies $P_0 \subset P' \subset P_k$, if P_k contains whole augmented production rules. So, roughly the complexity on worst case in which m embedded portions are contained is $O(n^3 2^m)$. On the worst cast, input texts consist of only embedded portions, but some embedded portions do not cause augmentation of production rules. Of course, a derivation from a variable $\mathbf{p} \in \mathbf{Aug}$ must cause an augmentation. The worst case is caused by the deal of embedded portions which are derived from syntactic variables other than of **Aug**. If we can suppose that given grammar is not ambiguous, and any strings derived from the grammar never contains sub-strings which form $[\alpha]$ and does not cause augmentation of production rules, we can accelerate the algorithm so as to change the condition $P_0 \subset P' \subset P_k$ in operation 7) to $P' = P_k$. On such acceleration, the worst cast is $O(n \times n^2 \times m) = O(n^3 m)$ $(\subset O(n^4))$.

Moreover, we have a restriction way to reduce complexity, i.e., so as to modify the definition of derivation. In each cases in the definition of derivation, a constraint

$$A \to X_1 \cdots X_n \in P_{n+1}$$

must be satisfied. This constraint means that the production rule used at the derivation may be a result of an augmentation of a production rule which is caused by the derivation itself. This enables circular definition and use of embedded portions, such like Example 3.2.2, and, the origin of expansion of complexity on worst cases. A choice we have here is to modify above constraint to

$$A \to X_1 \cdots X_n \in P_1.$$

To do so, we can modify the algorthm on operation 6) and 7), so as,

6) if $(Y \to \gamma\bullet, R_3, R_4) \in I(j,k)$ and $(X \to \alpha \bullet Y\beta,\ R_1,\ R_2) \in I(i,j)$ and $Y \to \gamma \in R_2$ for some $0 \leq i \leq j \leq k \leq n$, and if $R_3 \subset R_2$, then add $(X \to \alpha Y \bullet \beta,\ R_1,\ R_2 \cup R_4)$ to $I(i,k)$,

7) if $(\mathbf{p} \to \alpha\bullet, R_1, R_2) \in I(i,j)$ and $\mathbf{p} \in \mathbf{Aug}$ and $\mathbf{p} \to \alpha \in R_2$ and $[w] = a_{i+1} \cdots a_j$ and $f(w)$ has valid form of production rule, then let $Z \to \gamma = f(w)$, add $Z \to \gamma$ to $P_j, \ldots, P_n$ and then add $(Z \to \bullet\gamma,\ R_2,\ R_2)$ to $I(k,k)$ for each $k = j, \ldots, n$.

On this modification, complexity on worst case is also $O(n^5)$ if we assume that values of production rule sets in each items are hold as some ordered list, because problems $R = R'$ and $R \subset R'$ have same complexity on ordered lists R and R'.

Chapter 4

Incremental Construction of LALR Parser and its Applications for RCFG

4.1 Introduction

In this chapter, we propose an incremental construction method of LALR(1) parser for CFG, and its application to RCFG. Our main purpose is to construct a frame work of compiler-compiler which is upper compatible to YACC [16] or Bison [10], and moreover, which can treat extensible grammar. According to this purpose, some restrictions and needs arise. 1) the base grammar processed by the system must be an extension of CFG. It must includes CFG as a special case. 2) with some restrictions on the base grammar, LALR(1) parsing scheme or some other scheme resemble to it can be processed on the system. 3) ambiguity of given grammar must be solved in YACC style. 4) about error handling. These are the reason why we propose RCFG, and 1) and 2) are solved in this chapter. 3) and 4) are remained as a future works.

The typical points of the methodology is that both of LR(0) graph and Look Ahead Symbol Set (LA) for LALR(1) are constructed in fully incremental manner on our approach. To achieve it, we propose sevral notions. This chapter consists of six parts. Firstly, we discuss on an incremental method for constructing LR(0) graph (Section 4.2). The discussion in the section is a remake of [12] and [15]. We make discussion and introduce some notions from the point of view that current LR(0) graphs are some kind of division of expected new graph, rather than constructing new LR(0) graph by current graphs. From this view point, we clear that the incremental construction method for LR(0) graphs which had proposed in [12, 15] is not LR(0) graph specific notion, but is general one for directed graphs.

In Section 4.3, we introduce mainly two notions, *MonoG* and *MaxInc*. *MonoG* is an LR(0) graph which is induced from only one production rule, and we give an efficient algorithm to construct *MonoG* as Algorithm 4.5.1 in Section 4.5. *MaxInc* is one of most important notions in this chapter, which provides an idex for identification on LR(0) states. On conventional methods for constructing Parse Table for LR parsers, states of LR(0) graph are identified by an item set or a subset of items which forms the states. In the methodology we propose in this paper, we adopt a stance that no item set infomations are held in each calculation steps. To accomplish it, we introduce *MaxInc* which is based on inclusion relations between item sets of which each states consist, and, as a result, we gained efficiency on both of time and space on identifying states.

In Section 4.4, we discuss on method for calculation of LA for LALR(1) graphs. Main notions introduced in the section are *Dependency Domain*(DD), $E\Delta$, $Top\Delta$, $Dep\Delta$ and $Ind\Delta$. DD is a notion which is isomorphic to *Disjunction Normal Forms without Negative Literals* on Propositional Logic. We use DD to express conditions for ε-productivity for each syntactic symbols. $E\Delta$ is an ε-productivity judgement function, and $Top\Delta$ and $Dep\Delta$ are functions for calculating 'first' symbols and used to calculate LA sets. For three functions, we have established efficient incremental construction method, in the method no item sets have to be held during calculation on LALR(1) parse table. The notions introduced in the section are typical point of this chapter.

In Section 4.5, algorithms for calculation of incremental construction of LALR(1) graphs are provided. We discuss the efficiency of the method in Section 4.7. In Section 4.8, two ways of applications of the incremental construction method of LALR(1) graphs to RCFG are given.

4.2 Discussions on Finite Automata

In this section, we introduce a few important notions and establish results. The main results are Theorem 4.2.11 and 4.2.14 which describe the formalization of incremental construction of LR(0) graph and its soundness and completeness. To put arguments forward, we adopt the stance that current LR(0) graph and augmenting graph to it are obtained by dividing of expected result, instead of the stance that expected result is constructed by use of current LR(0) graph and augmenting graph. In Definition 4.2.13, the basis of incremental construction operation, named 'fusion', for LR(0) is formalized. However, in the definition, LR(0) specific notion, i.e. 'item', is not used. Intuitively, the fusion process is defined on given two DFAs, which are obtained from two εNFAs by subset construction method, with given *Bridge Transition* which stretches necessary transitions to construct expected result. The Bridge Transition is defined from expected result. So, the definition is not of 'construction' but of some kind of 'division' of a graph. However, it is not difficult matter on LR(0) to predict the Bridge Transition, as viewed in the next section. So, we can conclude that incremental construction method discussed in [12, 15] is not a special notion for LR(0) graph. The peculiarity on the method is the easiness to calculate Bridge Transition for it. The easiness is obtained from properties of transitions on LR(0) graphs.

4.2.1 Definition of Fusion of FAs and Some Properties

We introduce a notion *Sub-graph* of FA, which is quite similar to *node induced sub-graph* on Graph Theory, e.g. in [5, 26]. For incremental construction of LR parser, in [12, 13, 15], 'constructing methods' are given. We introduce a notion in order to give composition or fusion process on LR(0) state transition graph, which is defined precisely below, theoretical back ground from the point of view not of that new graph is created from current graph by use of incremental method, but of that the current graph is a some kind of division of expected new graph. First, we discuss on operations on εNFA, because LR(0) state transition graph described by εNFA makes us easy to understand incremental construction process of LR(0) graphs, as pointed out in [25]. Discussion on εNFA does not have direct relation to the main results of this section, but it only provides us clear view points.

Definition 4.2.1 (Sub-graph of εNFA)
For given εNFA A, a subgraph A' induced by $Q' \subset Q$ is defined as,

$$\begin{aligned} A' &= (\Sigma, Q', \delta', *, F') \\ \\ \delta'(q,a) &= \delta(q,a) \cap Q'(q \in Q', a \in \Sigma \cup \{\varepsilon\}) \\ (\text{tosay,}\delta' &= \delta \cap (Q' \times (\Sigma \cup \{\varepsilon\}) \times Q')) \\ * &= \begin{cases} q_0 & \text{if } q_0 \in Q' \\ \text{undefined} & \text{otherwise} \end{cases} \\ F' &= F \cap Q'. \end{aligned}$$

$Sub(A, Q')$ denotes induced sub-graph of A with Q'.

Definition 4.2.2 (Composition of εNFA)
For given εNFAs $A_1 = (\Sigma,\ Q_1,\ \delta_1,\ s_1,\ F_1)$, $A_2 = (\Sigma,\ Q_2,\ \delta_2,\ *_2,\ F_2)$ and given a relation $\delta' \subset Q_1 \times (\Sigma \cup \{\varepsilon\}) \times Q_2 \cup Q_2 \times (\Sigma \cup \{\varepsilon\}\ ) \times Q_1$, Composition of A_1 and A_2 with δ', $A = A_1 \ll A_2, \delta' \gg$ is defined as,

$$A = A_1 \ll A_2, \delta' \gg = (\Sigma, Q_1 \cup Q_2, \delta, s_1, F_1 \cup F_2)$$

where $\delta = \delta_1 \cup \delta_2 \cup \delta'$.

A_1 is called Subjective Subgraph of A, or simply Subjective, and, A_2 is called Dependent Subgraph of A, or simply Dependent. δ' is called Bridge Transition.

For arbitrary εNFA $A = (\Sigma,\ Q_1,\ \delta,\ q_0,\ F)$, $Sub(A, Q_1)$ and $Sub(A, Q_2)$, where $q_0 \in Q_1 \subset Q$ and $Q_2 \subset Q$, if we give Bridge Transition $\delta' = \delta \cap ((Q_1 \times (\Sigma \cup \{\varepsilon\}) \times Q_2 \cup Q_2 \times (\Sigma \cup \{\varepsilon\}) \times Q_1))$, it is obvious that $Sub(A, Q_1) \ll Sub(A, Q_2),\ \delta' \gg$ is isomorphic to A, the isomorphism is given in a manner below.

Definition 4.2.3 (Isomorphism on εNFA)
We write two εNFA $A_1 = (\Sigma,\ Q_1,\ \delta_1,\ q_1,\ F_1)$ and $A_2 = (\Sigma,\ Q_2,\ \delta_2,\ q_2,\ F_2)$ is equivalent, when there is an isomorphism $f : Q_1 \to Q_2$, s.t.,

$$\begin{aligned} f(q_1) &= f(q_2) \\ f(F_1) &= f(F_2) \\ f(\delta_1(q,a)) &= \delta_2(f(q), a)(\forall q \in Q_1, \forall a \in \Sigma \cup \{\varepsilon\}). \end{aligned}$$

Lemma 4.2.4 *For any εNFA $A = (\Sigma,\ Q,\ \delta,\ q_0,\ F)$, A is equivalent to $Sub(A, Q_1) \ll Sub(A, Q_2),\ (\delta \setminus \delta_1) \setminus \delta_2 \gg$, where $Q_1 \cup Q_2 = Q$, $q_0 \in Q_1$, $\delta_1 = \delta \cap (Q_1 \times (\Sigma \cup \{\varepsilon\}) \times Q_1)$, $\delta_2 = \delta \cap (Q_2 \times (\Sigma \cup \{\varepsilon\}) \times Q_2)$.*

(proof) Straightforward from definitions.//

Following definitions and results are important on discussion of incremental construction of LR(0) state transition graph. However all of them hold without use of the notion 'item', so we collect them in this section. Especially, relation $\mathcal{R}$ defined below will be used in order to give proof of soundness and completeness of incremental construction of LR(0) graph.

Definition 4.2.5 (Arrival Languages)
For given εNFA $A = (\Sigma, Q, \delta, q_0, F)$, Arrival Language $L(q)$, which means a set of strings that lease from initial state q_0 to q, is defined as,

$$L(q) = \{w \in \Sigma* \mid q \in \delta^*(q_0, w)\}.$$

When we emphasize that it is on A, we denote $L_A(q)$.

Definition 4.2.6 (Elimination of Unreachable States)
For given DFA $A = (\Sigma, Q, \delta, q_0, F)$, Eff$(Q)$, i.e. a set of Effective States, is defined as

$$\mathit{Eff}(Q) = \{q \mid \exists w \in \Sigma*, q = \delta(q_0, w)\}.$$

Under the condition $\delta' = \delta \cap \mathit{Eff}(Q) \times \Sigma \times \mathit{Eff}(Q)$, we can define a DFA Eff(A) which states are all effective states of A, such as,

$$\mathit{Eff}(A) = (\Sigma, \mathit{Eff}(Q), \delta', q_0, F \cap \mathit{Eff}(Q)).$$

Note: An ordinal graph of εNFA or DFA has unique state as start state. However, in this paper, we will treat a kind of *multi-entrance graphs* discussed in the next section. So, in following sections, it is expected that *Eff* is defined not only on a start state which is explicitly given in a formal statement of FA, but also on whole entrances. An FA which we treat has entrances according to each syntactic variable X, say $\mathit{Ent}\varepsilon(X)$, which means $\varepsilon C(\{X \to \bullet\alpha \mid X \to \alpha \in P\})$. These entrances are needed for fusion process in order to achieve augmenting a new production rule to current LR(0) graph. On this stance, q_0 is one of entrances, i.e. $\mathit{Ent}\varepsilon(S')$, and *Eff* must be defined as,

$$\mathit{Eff}(Q) = \{q \mid \exists X \in V, \exists w \in \Sigma*, q = \delta(\mathit{Ent}\varepsilon(X), w)\}.$$

In following sections, *Eff* will be used in this sense.

Definition 4.2.7 (Relation $\mathcal{R}$)
For given εNFA $A = (\Sigma, Q, \delta, q_0, F)$ and $Q_1, Q_2 \subset Q$ $(Q_1 \cup Q_2 = Q,\ q_0 \in Q_1)$, we state

$$\begin{aligned}\varepsilon\text{NFA}\, A_1 &= (\Sigma, Q_1, \delta_1, q_0, *) = \mathit{Sub}(A, Q_1)\\ \varepsilon\text{NFA}\, A_2 &= (\Sigma, Q_2, \delta_2, *, *) = \mathit{Sub}(A, Q_2)\end{aligned}$$

("$*$" means not used)

$$\begin{aligned}\text{DFA}\, B_1 &= SC(A)\\ &= (\Sigma, P_1, \zeta_1, s_1, H_1)\\ \varepsilon\text{NFA}\, B_2 &= SC(A_1) \ll SC(A_2), \xi \gg\\ &= (\Sigma, \mathit{Power}(Q_1) \cup \mathit{Power}(Q_2), \delta', F')\\ \text{DFA}\, B_2' &= SC(B_2)\\ &= (\Sigma, P_2, \zeta_2, s_2, H_2)\end{aligned}$$

where

$$\begin{aligned}\xi \;\subset\; &(\mathit{Power}(Q_1) \times (\Sigma \cup \{\varepsilon\}) \times \mathit{Power}(Q_2))\\ &\cup(\mathit{Power}(Q_2) \times (\Sigma \cup \{\varepsilon\}) \times \mathit{Power}(Q_1))\end{aligned}$$

$$\begin{aligned}(U,a,V)\in\xi \;\Leftrightarrow\;& U\subset Q_1, V=\varepsilon C(\delta_2,\delta(U,a)\cap Q_2)\\ \textit{or}\;& U\subset Q_2, V=\varepsilon C(\delta_1,\delta(U,a)\cap Q_1).\end{aligned}$$

Now, we define a relation $\mathcal{R}$ on $P_1\times P_2$ recursively, such as

$$\begin{aligned}&(s_1,s_2)\in\mathcal{R}\\ &(q_1,q_2)\in\mathcal{R} \;\Rightarrow\; \forall a\in\Sigma,(\zeta_1(q_1,a),\zeta_2(q_2,a))\in\mathcal{R}\end{aligned}$$

$\mathcal{R}$ is a minimum set that satisfies above two conditions.
$\mathcal{R}(q_1)$ denotes a set $\{q_2 \mid (q_1,q_2)\in\mathcal{R}\}$, and also, $\mathcal{R}^-(q_2)$ denotes a set $\{q_1 \mid (q_1,q_2)\in\mathcal{R}\}$.

Lemma 4.2.8 *For any $q_1\in P_1$, $q_2\in P_2$, if q_1 is reachable from s_1, then $\mathcal{R}(q_1)\neq\phi$, and also, if q_2 is reachable from s_2, then $\mathcal{R}^-(q_2)\neq\phi$.*

(proof) What q_1 is reachable from s_1 means that there exists a word of arrival language $w_1\in L(q_1)$, and $q_1=\zeta_1(s_1,w_1)$. Thus, $(q_1,\zeta_2(s_2,w_1))\in\mathcal{R}$. In same way, $(\zeta_1(s_1,w_2),q_2)\in\mathcal{R}$ holds.//

Lemma 4.2.9

$$\begin{aligned}\forall U\in \mathit{Power}(Q_1),\quad& \bigcup\varepsilon C(\delta',\varepsilon C(\delta_1,U))=\varepsilon C(\delta,U),\\ \forall U\in \mathit{Power}(Q_2),\quad& \bigcup\varepsilon C(\delta',\varepsilon C(\delta_2,U))=\varepsilon C(\delta,U).\end{aligned}$$

(proof) We can easily show $\bigcup\varepsilon C(\delta',\,\varepsilon C(\delta_i,U))\subset\varepsilon C(\delta,U)$ $(i=1$ or $2)$ from the facts $U\subset\varepsilon C(\delta,U)$ and $\delta'=\delta_1\cup\delta_2\cup\xi$. Conversely, for any state $q\in\varepsilon C(\delta,U)$, we show $q\in\bigcup\,\varepsilon C(\delta',\,\varepsilon C(\delta_i,U))$ by induction, considering an ε-transition sequence on A, $\rho=q_1,\dots,q_k(q_1\in\varepsilon C(\delta_i,U),q_k=q)$ $(i=1$ or $2)$. First, ρ is divided into sub-sequences from its top, $\rho=\rho_1,\dots,\rho_m$, on the condition whether each element is included in Q_1 or $Q_2\setminus Q_1$, s.t., all elements of ρ_j are included in Q_1 then all elements of ρ_{j+1} is included in $Q_2\setminus Q_1$, or conversely. On the case $m=1$, because each element q' of ρ_1 is included in $\varepsilon C(\delta_i,U)$, $q'\in\bigcup\,\varepsilon C(\delta',\,\varepsilon C(\delta_i,U))$ holds. We suppose the induction hypothesis holds on m. Let $q'=\delta(q,\varepsilon)$ and consider an ε-transition sequence $\rho q'$. If q' is contained in the same set of ρ_m, which means Q_1 or $Q_2\setminus Q_1$, then for the top state p of ρ_m, $q'\in\varepsilon C(\delta_i,p)$ holds. Thus, we can claim that $q'\in\bigcup\varepsilon C(\delta',\,\varepsilon C(\delta_i,U))$ holds from the definition of ξ. If q' is contained in the opposite set to ρ_m, because a transition from a state which consists of ρ_m to a state of $\varepsilon C(\delta_i',q')$ is given by ξ, we can claim that $\exists U'\in\varepsilon C(\delta',\,\varepsilon C(\delta_i,U))$ and $\varepsilon C(\delta_i',q')\subset U'$ hold. So, $q'\in\bigcup\varepsilon C(\delta',\,\varepsilon C(\delta_i,U))$ and the induction hypothesis holds also on $m+1$.//

These two lemmas are implicitly used in followings.

Lemma 4.2.10 *For any $a\in\Sigma$, any $U\in \mathit{Power}(Q_1)\cup \mathit{Power}(Q_2)$ and any $q\in\bigcup\delta'(\varepsilon C(\delta_i,$ $U),\,a)$, there exists $q'\in U$, s.t., $q\in\varepsilon C(\delta(\varepsilon C(\delta,\,q'),\,a))$, where $i=1$ if $U\subset Q_1$, $i=2$ if $U\subset Q_2$.*

(proof) $\varepsilon C(\delta_i, U)$ is a state of $SC(Sub(A, Q_i))$. We write the state transition function of $SC(Sub(A, Q_i))$ by δ'_i, then

$$\begin{aligned}\delta'(\varepsilon C(\delta_i, U), a) &= \{\delta'_i(\varepsilon C(\delta_i, U), a)\} \\ &\quad \cup \xi(\varepsilon C(\delta_i, U), a).\end{aligned}$$

Consider the case $q \in \delta'_i(\varepsilon C(\delta_i, U), a)$, it is obvious that

$$\begin{aligned}\delta'_i(\varepsilon C(\delta_i, U), a) &= \varepsilon C(\delta_i, \delta_i(\varepsilon C(\delta_i, U), a)) \\ &\subset \varepsilon C(\delta, \delta(\varepsilon C(\delta_i, U), a))\end{aligned}$$

holds, so the proposition holds on this case. Consider the case $q \in \bigcup \xi(\varepsilon C(\delta_i, U), a)$ remained, it is clear that $\exists U' \in \xi(\varepsilon C(\delta_i, U), a)$ s.t. $q \in U'$ holds, and from the definition of ξ,

$$\begin{aligned}U' &= \varepsilon C(\delta'_i, \delta(\varepsilon C(\delta_i, U), a) \cap Q'_i) \\ &\subset \varepsilon C(\delta, \delta(\varepsilon C(\delta, U), a))\end{aligned}$$

is trivial. So the proposition holds in any cases. //

Theorem 4.2.11 $(U_1, U_2) \in \mathcal{R} \Rightarrow U_1 = \bigcup U_2$

(proof) Proved by induction. On the case of $U_1 = s_1 = \varepsilon C(\delta, q_0)$, $s_2 = \varepsilon C(\delta_1, q_0)$ holds. Let ρ be an ε-transition sequence from q_0 to q, say $\rho = q_0, q_{i_1}, q_{i_2}, \ldots, q_{i_k} = q$. $\forall q \in U_1 \Rightarrow q \in \bigcup s_2$ vacuously holds on the case $k = 0$. On the case $k \geq 1$, we divide ρ into sub-sequences $\rho = \rho_0, \rho_1, \ldots, \rho_m$ in the same manner of Lemma 4.2.9. We write $\rho_j = q_{i_{n_j - 1}+1}, \ldots, q_{i_{n_j}}$. If j is even, all elements of ρ_j are included in Q_1, and if j is odd, all elements of ρ_j are included in Q_2. On the case $m = 0$, because whole elements of ρ are included in $\varepsilon C(\delta_1, q_0)$, $q \in \bigcup s_2$ holds. Suppose on all cases that $k = 0, \ldots, t$, the induction hypothesis is holds. Let $\rho' = \rho q'$ be an ε-transition sequence. It is obvious that $q_{i_{n_t}} \in Q_1$, and if $q' \in Q_1$, then a transition from a state of $SC(Sub(A, Q_2))$ which includes $q_{i_{n_t - 1}}$ to $\varepsilon C(\delta_1, q_{i_{n_k - 1}})$is stretched by ξ. Additionally, considering the fact $q' \in \varepsilon C(q_{i_{n_t - 1}+1})$, we can claim $q' \in \bigcup s_2$. Thus $s_1 \subset \bigcup s_2$ holds. $\bigcup s_2 \subset s_1$ is proved in the same way.

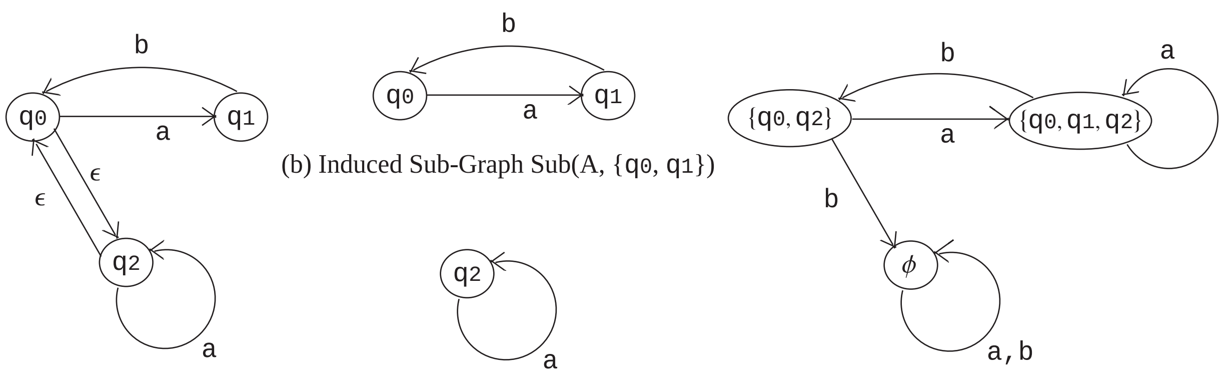
Consider states $\zeta_1(U_1, a)$ and $\zeta_2(U_2, a)$ for a pair of states (U_1, U_2), s.t., $(U_1, U_2) \in \mathcal{R}$ and $U_1 = \bigcup U_2$. From the facts that $\zeta_1(U_1, a) = \varepsilon C(\delta, \delta(U_1, a))$,

$$\begin{aligned}\zeta_2(U_2, a) &= \varepsilon C(\xi, \\ &\quad (\delta_1(U_2 \cap \mathit{Power}(Q_1), a) \cup \delta_2(U_2 \cap \mathit{Power}(Q_2), a) \\ &\quad \cup \bigcup \xi(U_2 \cap \mathit{Power}(Q_1), a) \cup \bigcup \xi(U_2 \cap \mathit{Power}(Q_2), a))\end{aligned}$$

and $U_1 = \bigcup U_2$, we can claim $\delta_1(U_2 \cap \mathit{Power}(Q_1), a) \subset \delta(U_1, a), \delta_2(U_2 \cap \mathit{Power}(Q_2), a) \subset \delta(U_1, a), \xi(U_2 \cap \mathit{Power}(Q_1), a) \subset \varepsilon C(\delta, \delta(U_1, a)), \xi(U_2 \cap \mathit{Power}(Q_2), a) \subset \varepsilon C(\delta, \delta(U_1, a))$. So, by Lemma 4.2.10, $\bigcup \zeta_2(U_2, a) \subset \varepsilon C(\delta, \delta(U_1, a))$.

Conversely, we show $\varepsilon C(\delta, \delta(U_1, a)) \subset \bigcup \zeta_2(U_2, a)$. From the construction of $\mathcal{R}$, $U_1 = \varepsilon C(\delta, U_1)$. From the definition of $SC(Sub(A, Q_1))$, $SC(Sub(A, Q_2)$ and ξ,

$$\begin{aligned}\delta(U_1, a) &\subset \delta_1(U_2 \cap \mathit{Power}(Q_1), a) \cup \delta_2(U_2 \cap \mathit{Power}(Q_2), a) \\ &\quad \cup \bigcup \xi(U_2 \cap \mathit{Power}(Q_1), a) \\ &\quad \cup \bigcup \xi(U_2 \cap \mathit{Power}(Q_2), a)\end{aligned}$$



(a) Original ϵNFA A (b) Induced Sub-Graph Sub(A, {q0, q1}) (c) Induced Sub-Graph Sub(A, {q2}) (d) DFA SC(A) by Subset Construction

Figure 4.1: Example of State Disruption (1)

holds. Using these facts and Lemma 4.2.10, we can claim

$$\begin{aligned}\varepsilon C(\delta, \delta(U_1, a)) \quad \subset \quad & \varepsilon C(\xi, \delta_1(U_2 \cap Power(Q_1), a) \\ & \cup \delta_2(U_2 \cap Power(Q_2), a) \\ & \cup \bigcup \xi(U_2 \cap Power(Q_1), a) \\ & \cup \bigcup \xi(U_2 \cap Power(Q_2), a)),\end{aligned}$$

and proof is completed.//

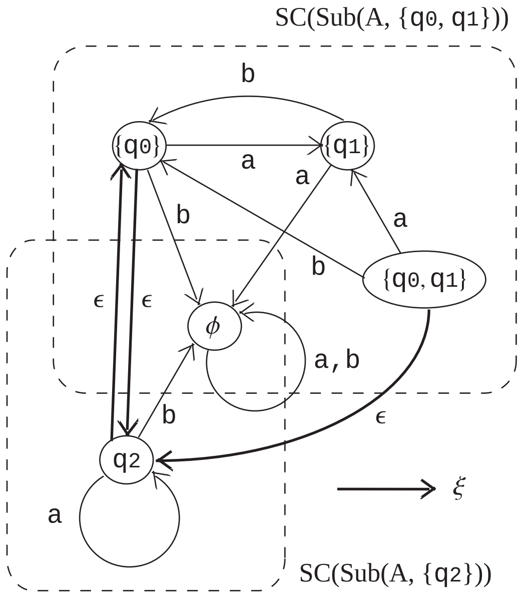
Before entering definitions which concern to incremental construction, we give rise an example which illustrates the needs of the definition. From the result of Theorem 4.2.11, one might imagine that $SC(A) \sim SC(SC(Sub(A,\, Q_1)) \ll SC(Sub(A,\, Q_2)),\, \xi \gg)$ holds with some fortunate Bridge Transition ξ. If it held, it would become quite fortunate method for us, because we would like to construct an incremental construction method for LR(0) graph without use of any item set. Unfortunately, $SC(SC(Sub(A,\, Q_1)) \ll SC(Sub(A,\, Q_2)),\, \xi \gg)$ causes state disruptions in some cases, and following example is one of them.

Example 4.2.12 *There exist εNFA $A = (\Sigma,\, Q,\, \delta,\, q_0,\, F)$ and a pair of subsets of Q, Q_1, Q_2, s.t., $Q_1 \cup Q_2 = Q$ and $q_0 \in Q_1$, which causes a result that Eff$(SC(A))$ is not equivalent to Eff$(SC(SC(Sub(A,\, Q_1)) \ll SC(Sub(A,\, Q_2)),\, \xi \gg))$.*

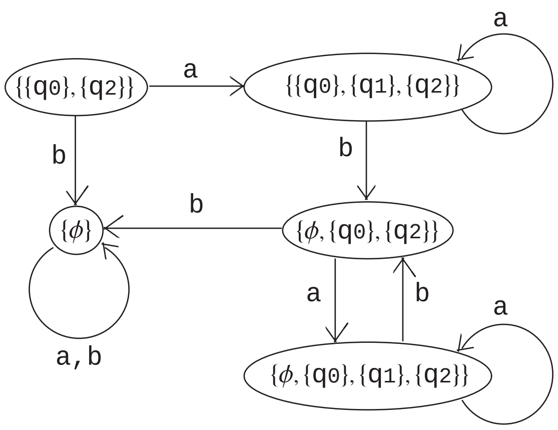
Figure 4.1 and 4.2 illustrate an example of the case above. (d) in Figure 4.1 is the result of Sub-set Construction on original εNFA (a), and (f) in Figure 4.2 is the result of Sub-set Construction on compositions of (b) $SC(Sub(A,\, \{q_0, q_1\}))$ and (c) $SC(Sub(A,\, \{q_2\}))$. (f) is not isomorphic to (d).

Definition 4.2.13 (Fusion of two DFA)
For given εNFA $A = (\Sigma,\, Q,\, \delta,\, q_0,\, F)$ and a pair of subsets of Q, Q_1, Q_2, where $Q_1 \cup Q_2 = Q$ and $q_0 \in Q_1$, we also state, same as Definition 4.2.7,

$$\varepsilon\text{NFA}\, A_1 \quad = \quad (\Sigma, Q_1, \delta_1, q_0, *) = Sub(A, Q_1)$$



(e) Composition of SC(Sub(A, {q0, q1})) and SC(Sub(A, {q2})) with ξ



(f) DFA of (e) by Subset Construction

Figure 4.2: Example of State Disruption (2)

$$\begin{aligned}
\varepsilon\text{NFA}\, A_2 &= (\Sigma, Q_2, \delta_2, *, *) = Sub(A, Q_2) \\
\\
\text{DFA}\, B_1 &= SC(A) \\
&= (\Sigma, P_1, \zeta_1, s_1, H_1) \\
\varepsilon\text{NFA}\, B_2 &= SC(A_1) \ll SC(A_2), \xi \gg \\
&= (\Sigma, Power(Q_1) \cup Power(Q_2), \delta', F') \\
\text{DFA}\, B_2' &= SC(B_2) \\
&= (\Sigma, P_2, \zeta_2, s_2, H_2).
\end{aligned}$$

DFA $Fus(SC(A_1), SC(A_2))$ under A is defined as

$$Fus(SC(A_1), SC(A_2)) = (\Sigma, Power(Q), \delta'', q_0'', F'')$$

$$q_0'' = \bigcup \varepsilon C(\xi, \varepsilon C(\delta_1, q_0))$$

$$\delta''(\bigcup V, a) \;=\; \bigcup \zeta_2(V, a) \tag{4.1}$$

$$F'' = \{U \subset Q \mid U \cap F \neq \phi\} = \{\bigcup U \mid U \in Power(F_1 \cup F_2) \setminus \phi\},$$

where ξ is defined same as Definition 4.2.7,

$$\begin{aligned}
\xi \;\subset\; & (Power(Q_1) \times (\Sigma \cup \{\varepsilon\}) \times Power(Q_2)) \\
& \cup (Power(Q_2) \times (\Sigma \cup \{\varepsilon\}) \times Power(Q_1))
\end{aligned}$$

$$\begin{array}{rcl}(U,a,V) \in \xi & \Leftrightarrow & U \subset Q_1, V = \varepsilon C(\delta_2, \delta(U,a) \cap Q_2) \\ & \mathit{or} & U \subset Q_2, V = \varepsilon C(\delta_1, \delta(U,a) \cap Q_1).\end{array}$$

We also call $SC(A_1)$ subjective and $SC(A_2)$ dependent.

The definition of Bridge Transition ξ gives us an understanding such that it is need a notion of entrances for graphs other than start states in order to fuse graphs, that are denoted by $\varepsilon C(\delta_1,\ \delta(U,a) \cap Q_1)$ for $SC(A_1)$ and $\varepsilon C(\delta_2,\ \delta(U,a) \cap Q_2)$ for $SC(A_2)$. As discussed in next section, on the case of fusing LR(0) graphs, $\varepsilon C(\delta_1,\ \delta(U,a) \cap Q_1)$ means an item set equal to $\varepsilon C(\{X \to \bullet\alpha \mid X \to \alpha \in P\})$ for some syntactic variable X concerning to the process. So, LR(0) graphs dealt in this paper are essentially some kind of multi-entrance graphs rather than conventional LR(0) graphs. We will write $\mathit{Ent\varepsilon}(X)$ for a state $\varepsilon C(\{X \to \bullet\alpha \mid X \to \alpha \in P\})$. Using $\mathit{Ent\varepsilon}$, ξ is easily defined at LR(0) graphs so as that if a state q contains an item $Y \to \alpha\ \bullet X\ \beta$, which can be determined with the existence of transition by X from q, ξ must contain a pair $(q,\ \mathit{Ent\varepsilon}(X))$. So, we can use fusion as an effective construction process of two LR(0) graphs, which does not use item set information at all.

Theorem 4.2.14 $SC(A) \sim \mathit{Fus}(SC(\mathit{Sub}(A,Q_1)), SC(\mathit{Sub}(A,Q_2)))$

(proof) From the definition of *Fus*, each state of $\mathit{Fus}(SC(\mathit{Sub}(A,\ Q_1)),\ SC(\mathit{Sub}(A,\ Q_2)))$ is merely a rename of a state of $SC(A)$. The soundness and the completeness are ensured by Theorem 4.2.11.//

Corollary 4.2.15 *For given εNFA $A = (\Sigma,\ Q,\ \delta,\ q_0,\ F)$ and given finite class of subset of Q, $Q_1,\ Q_2, \ldots, Q_n$, where $Q_1 \cup Q_2 \cup \cdots \cup Q_n = Q$, let*

$$\begin{array}{rcll} C_1 & = & SC(\mathit{Sub}(A,Q_1)) & \\ C_i & = & \mathit{Fus}(C_{i-1}, SC(\mathit{Sub}(A,Q_i)))) & (2 \le i \le n) \\ \text{where} & & & \\ \xi_j & \subset & (\mathit{Power}(Q_1 \cup \cdots \cup Q_j) \times (\Sigma \cup \{\varepsilon\}) \times \mathit{Power}(Q_{j+1})) & \\ & \cup & (\mathit{Power}(Q_{j+1}) \times (\Sigma \cup \{\varepsilon\}) \times \mathit{Power}(Q_1 \cup \cdots \cup Q_j)) & \end{array}$$

$$\begin{array}{rcl}(U,a,V) \in \xi_j & \Leftrightarrow & U \subset Q_1 \cup \cdots \cup Q_j, V = \varepsilon C(\delta'_{j+1}, \delta(U,a) \cap Q_{j+1}) \\ & \mathit{or} & U \subset Q_{j+1}, V = \varepsilon C(\delta_j, \delta(U,a) \cap (Q_1 \cup \cdots \cup Q_j))\end{array}$$

when we state $\mathit{Sub}(A, Q_1 \cup \cdots \cup Q_j) = (\Sigma, Q_1 \cup \cdots \cup Q_j, \delta_j, *_j, F_j)$, $\mathit{Sub}(A, Q_{j+1}) = (\Sigma, Q_{j+1}, \delta'_{j+1}, *'_{j+1}, F'_{j+1})$. Then $C_i \sim SC(\mathit{Sub}(A, Q_1 \cup \cdots \cup Q_i))(1 \le i \le n)$. Especially on the case $i = n$, $C_n \sim SC(A)$.

Someone might have doubt on fusion as an incremental construction method for LR(0) which does not use information 'item set', because in the definition of fusion, as expression 4.1, it is seemed so that union operation on item set is needed. However, this doubt will be cleared in the next section. The expression 4.1 plays a role of identification of states in algorithms of incremental construction. We prepare another way to identification of states, which is based on inclusion relation between states.

4.3 Discussions on LR(0) Parsing Table (State Transition Graph)

In this section, we discuss on subjects which depends on notions essentially forms LR(0) graph. Basic notions and notations on this section are also summarized in Chapter 2. Most important results in this section are *MonoG*, *MaxInc* and properties on them, defined in Notation 4.3.3 and Definition 4.3.10, respectively. *MonoG* is an LR(0) graph induced from only one production rule, which has good properties for incremental construction of LR(0) graphs and for calculation of Look Ahead Sets without item set information. In the algorithms of incremental construction of LALR(1) graph, described in Section 4.5, it is assumed that both of initial graph, which is the start graph of construction, and augmenting graph are *MonoG*. *MaxInc* is used in order to identify states. On conventional construction algorithms for LR(0) graphs, as written in [1, 2], a state is identified with a subset of items which form the state, so called *kernel* or *core*. On contrary to the conventional methods, our approach adopts another information, i.e. *MaxInc*, in order to identify states, which does not consists of items, but consists of inclusion information between states. The efficiency of *MaxInc* will be discussed in Section 4.6.

Usually, LR(0) graph is constructed as a DFA. Using the notation described in Chapter 2, for any given CFG G, LR(0) graph is denoted by $\mathit{Eff}(SC(lr(G)))$. The fact that if a transition from q to q' by X exists, then it is clear that q must contain some item forms $Y \rightarrow \alpha \bullet X\beta$. Also, this feature is inherited to $\mathit{Eff}(SC(lr(G)))$, and used frequently in algorithms presented in Section 4.5. In fact, Bridge Transition ξ needed to achieve fusion operation is calculated using this information, as described in Algorithm 4.5.3. How many information is needed to achieve incremental construction of LR(0) graph? Making an extreme argument, if topology of LR(0) graph and reduce information for each state are given, we can recalculate item sets for each state. If we can decide topology of LR(0) graph with the other information, the role of item set information, or *kernel*, is needed only on identifying states. For such purpose, we introduce a notion *MaxInc* defined below. To calculate topology of LR(0) and *MaxInc* easily, we use notion *MonoG*. To calculate *MonoG*, we use no item set. Instead of item sets, we adopt a notion *Flow* defined below. Our approach can be illustrated as an iteration process of construction of *MonoG* and fusion of LR(0) graph and *MonoG*.

4.3.1 Mono-Graph

Firstly, we confirm a proposition stated below.

Lemma 4.3.1 *Let* $A = lr(G)$ *be an LR(0) graph,* $A' = SC(A) = (V \cup T,\ \mathit{Power}(\mathit{Item} \cup \{q_0\}),\ \delta',\ q'_0,\ \phi)$*, and* q *be a state of* $SC(A)$*. For some item* $X \rightarrow \alpha\ X_1 \cdots X_k\ \bullet\beta \in q$*, there exists a state of* $SC(A)$*, say* q'*, s.t.,* $\delta'(q',\ X_1 \cdots X_k) = q$*. Moreover, if , for some string* $w \in (V \cup T)*$*,* $\mid w \mid = k$*,* $\delta'(q'',\ w) = q$ *then* $w = X_1 \cdots X_k$*.*

(proof) By induction on k.//

Corollary 4.3.2

$$\begin{aligned} \mathit{Root}(q, X \rightarrow \alpha \bullet \beta) &= \{q' \subset \mathit{Item} \mid \delta'(q', \alpha) = q\} \\ &= \{q' \subset \mathit{Item} \mid \delta'(q', w) = q, \mid w \mid = \mid \alpha \mid\} \end{aligned}$$

Lemma 4.3.4 is one of most important results of this section, which concerns to *MonoG*. The proof of the lemma suggests an algorithm of effective construction of *MonoG*.

Notation 4.3.3 (Mono-Graph)
An LR(0) graph (DFA) which is obtained from only one production rule $Z \to X_1 \cdots X_n$ is denoted by $MonoG(Z \to X_1 \cdots X_n)$.

Lemma 4.3.4 *The number of states of $MonoG(Z \to X_1 \cdots X_n)$ is strictly $n+1$.*

(proof) Function $Index : Item(\{Z \to X_1 \cdots X_n\}) \to \mathbf{N}$, which assigns an integer for each item, is defined as,

$$\begin{array}{rcll} Index(Z \to \bullet X_1 \cdots X_n) &=& 0 & \\ Index(Z \to X_1 \cdots X_i \bullet \cdots X_n) &=& i & (1 \le i \le n), \end{array}$$

and, using *Index*, let $\overline{Index} : Power(Item(\{Z \to X_1 \cdots X_n\}) \to \mathbf{N}$ be defined as,

$$\overline{Index}(U) = \max_{item \in U} Index(item).$$

Index denotes either *Index* or $\overline{Index}$, if no ambiguity. From the definition of LR(0) graph A (εNFA), there is a sequence of $n+1$ states, starts with an item $Z \to \bullet X_1 \cdots X_n$ to $Z \to X_1 \cdots X_n \bullet$. It is obvious that the values of *Index* for each states are mutually distinct, which values are $0, 1, \ldots, n$. From this fact, the values of *Index* for each states of $SC(A)$ also exhausts 0 to n, because destination state by ε-transition from any state $Z \to X_1 \cdots \bullet X_j \cdots X_n$ on A is, if exists, only $Z \to \bullet X_1 \cdots X_n$. So, the value of *Index* on $\varepsilon C(Z \to X_1 \cdots \bullet X_j \cdots X_n)$ is also j. Thus, from initial state $\varepsilon C(Z \to \bullet X_1 \cdots X_n)$, for each $i = 0, \ldots, n$, there exists a state which includes an item $Z \to X_1 \cdots \bullet X_i \cdots X_n$ that gives value of *Index* of the state is reachable. Thus, the number of states of $SC(A)$ is more than $n+1$.

Before giving a proof of the converse, we certify a fact that if $X_1 \ne Z$, then kernel of each state contains only one element. Suppose a state q contains an item $Z \to \cdots \bullet Z \cdots$, then q must contain an item $Z \to \bullet X_1 \cdots X_n$. However, because of the supposition $X_1 \ne Z$, $Z \to \cdots Z \bullet \cdots$ and $Z \to X_1 \bullet X_2 \cdots X_n$ never occurs in same state. So, number of elements of kernel never increases.

Conversely, suppose there are two distinct states of $SC(A)$ which have same value of *Index*, say q and q', and $Index(q) = Index(q') = j$. Kernels of q and q' are distinct, because we assume $q \ne q'$. Thus there exists an item $Z \to X_1 \cdots X_i \bullet \cdots X_n \in q$, $Z \to X_1 \cdots X_i \bullet \cdots X_n \notin q'$ and $i < j$, and addtionally we can assume i as the maximum among such values. From assumptions that $Z \to X_1 \cdots \bullet X_j \cdots X_n \in q, q'$ and $Z \to X_1 \cdots X_i \bullet \cdots X_n \in q$, we can claim that $X_{j-i} \cdots X_{j-1} = X_1 \cdots X_i$ by Lemma 4.3.1 and there exist states p and p', s.t., q is reachable from p by $X_1 \cdots X_i$ and q' is reachable from p' by $X_1 \cdots X_i$, respectively, and $Z \to \bullet X_1 \cdots X_n \in p$, $Z \to \bullet X_1 \cdots X_n \notin p'$. From the fact discussed above, $X_{j-i} = X_1 = Z$ must hold. So, because p' contains item $Z \to X_1 \cdots \bullet X_{j-i} \cdots X_n$, p' must contain item $Z \to \bullet X_1 \cdots X_n$. It is contradiction.

Above all, states are identified by values of *Index*, thus, number of states is just $n+1$.//

To calculate topology of *MonoG*, and also LA discussed in the next section, we introduce a notion *Flow*. Intuitively, *Flow* holds information whether item $Z \to \alpha \bullet X\beta$

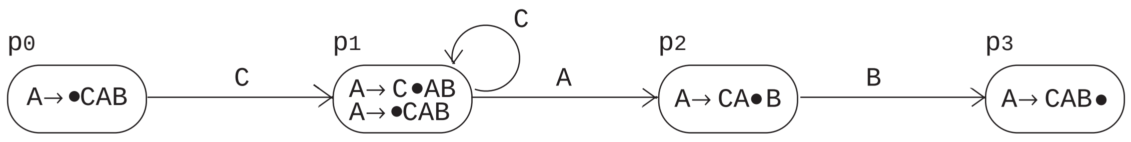


Figure 4.3: *Mono* for G_4

belongs to q, for each state q and each $X \in V$. If $(q, X, i) \in$ *Flow* then $Z \to \alpha \bullet X \ \beta \in q$ and $X_i = X$. In other words, under an interpretation of *Flow* being a function $Q_2 \times V \to$ *Power*$(\{1, \ldots n\})$, $\bigcup_{X \in V}$ *Flow*(q, X) gives item set description of state q. Precise arguments on use of *Flow* is described in Algorithm 4.5.1. *Flow* is calculated and used during calculating *MonoG*, and when the calculation of *MonoG* is completed, *Flow* is destructed.

Definition 4.3.5 (*Flow*)
For given MonoG$(Z \to X_1 \cdots X_n) = (V \cup T,\ Q_2,\ \zeta_2,\ \{Z \to \bullet X_1 \cdots X_n\},\ *)$*, Flow* $\subset Q_2 \times V \times \{1, \ldots, n\}$ *is defined as* $(q, X_i, i) \in$ *Flow iff* $Z \to X_1 \cdots X_{i-1} \bullet X_i \cdots X_n \in q$ *and* $X_i \in V$.

Note: Actual values of *Flow* are calculated in according to the structure of *MonoG* which is established in the proof of Lemma 4.3.4.

Example 4.3.6 (MonoG and Flow) *Here, we illustrate an example of MonoG and Flow for CFG* $G_4 = (\{A, B, C\},\ \phi,\ \{A \to C\,A\,B\},\ A)$ *(*$= lr(A \to C\,A\,B)$*). MonoG*(G_4) *is illustrated in Figure 4.3 with item sets. To construct this graph, firstly we calculate Flow for the graph. Flow is calculated, piling up εNFA* $lr(A \to CAB)$ *as shown in Figure 4.4. Strict value of the Flow for MonoG*$(lr(A \to C\,A\,B))$ *is as below,*

$$\mathit{Flow} = \{(p_0, C, 1), (p_1, A, 2), (p_1, C, 1), (p_2, B, 3)\}$$

4.3.2 *MaxInc*: a descriptor for state identification of LR(0)

From now on, we start arguments concerning to a method for state identification. As mentioned at the top of this chapter, we abandon information of item set for each state. Instead of it, we adopt inclusion relation between states. And, using the relation, an index for state identifier, named *MaxInc*, is introduced. Intuitively, for each state q, *MaxInc*(q) is a subset of states as its value. If *MaxInc*(q) is empty set, then q is a minimum state which has a unique item *item*, in the meaning that if $\exists q'$, s.t., *item* $\in q'$ then q' must involve q. If *MaxInc*(q) is not empty set, then q is obtained so as to fuse states of *MaxInc*(q). Of coarse, there are combinatorially many choices to express such a set. Trivial one choice is so as to collect all states which are included by q, but we adopt another way founded on a property of LR(0) graph, which is stated in Lemma 4.3.9. The way is to calculate a kind of maximal set, of which existence and uniqueness is stated in the lemma. Moreover, we show methods for incremental renewal of *MaxInc* through Lemma 4.3.12 to Theorem 4.3.18.

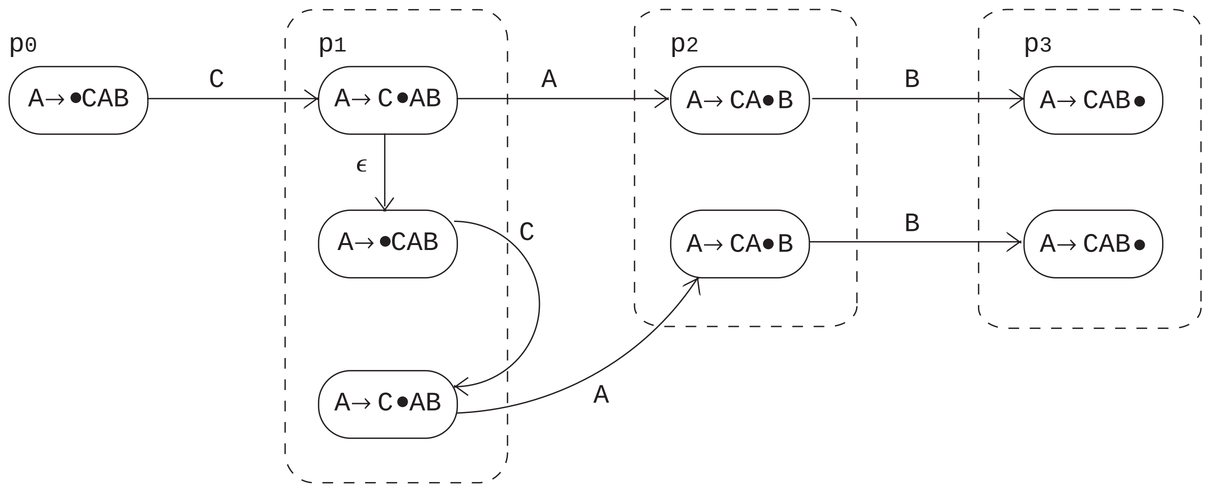


Figure 4.4: Calculation of *Flow*

Lemma 4.3.7
Suppose, given an LR(0) graph (DFA)

$$SC(lr(G)) = (V \cup T, Power(Item), \zeta, \varepsilon C(\{S' \to \bullet S\}), *).$$

If states $q_1, q_2 \subset Item$ are reachable from $\varepsilon C(\varepsilon Item(X_1))$ and $\varepsilon C(\varepsilon Item(X_2))$ for some $X_1, X_2 \in V$, respectively, and if $q_1 \cap q_2 \neq \phi$, then there is a non-empty state $q \subset q_1 \cap q_2$ which is reachable from $\varepsilon C(\varepsilon Item(X))$ for some $X \in V$.

(proof) Consider $X \to Y_1 \cdots Y_i \bullet\beta \in q_1 \cap q_2$. From the construction of $SC(lr(G))$, we can state that q_1 is reachable from an entrance $\varepsilon C(\varepsilon Item(X_1))$ with a symbol sequence $w_1 Y_1 \cdots Y_i$ for some $w_1 \in (V \cup T)*$, and also, q_2 is reachable from an entrance $\varepsilon C(\varepsilon Item(X_2))$ with a symbol sequence $w_2 Y_1 \cdots Y_i$ for some $w_2 \in (V \cup T)*$. Considering states p_1 and p_2, which are stated by $p_1 = \zeta(\varepsilon C(\varepsilon Item(X_1)), w_1)$, $p_2 = \zeta(\varepsilon C(\varepsilon Item(X_2))$, $\varepsilon C(\varepsilon Item(X)) \subset p_1 \cap p_2$ must hold. Thus, we can claim that $\zeta(\varepsilon C(\varepsilon Item(X)), Y_1 \cdots Y_i)$ $\subset q_1 \cap q_2$, and so, $q = \zeta(\varepsilon C(\varepsilon Item(X)), Y_1 \cdots Y_i) \subset q_1 \cap q_2$, which satisfies the proposition.//

Definition 4.3.8 (An index *Inc* for Inclusion Relation)
For given LR(0) graph $SC(lr(G)) = (V \cup T,\ Power(Item),\ \zeta,\ \varepsilon C(\{S' \to \bullet S\}),\ *)$, we write ES(Effective States) for a set of all states which are reachable from $\varepsilon C(\varepsilon Item(X))$ for some $X \in V$. We define $Inc : ES \to Power(ES)$, using

$$Inc'(q) = \{U \subset ES \mid q = \bigcup U \text{ and } q \notin U\},$$

as,

$$Inc(q) = \{U \subset ES \mid q = \bigcup U \,\text{and}\, q \notin U \text{ and } \forall q' \in U, Inc'(q') = \phi\}.$$

Lemma 4.3.9 (The existence of Maximal Set of $Inc(q)$)
If $Inc(q) \neq \phi$, then there exists a unique maximal set in $Inc(q)$, where a maximal set $\mu \in Inc(q)$ is a set of states which satisfies condition $\forall U \in Inc(q)$, $\forall q' \in U$, $\exists q'' \in \mu$, $q' \subset q''$.

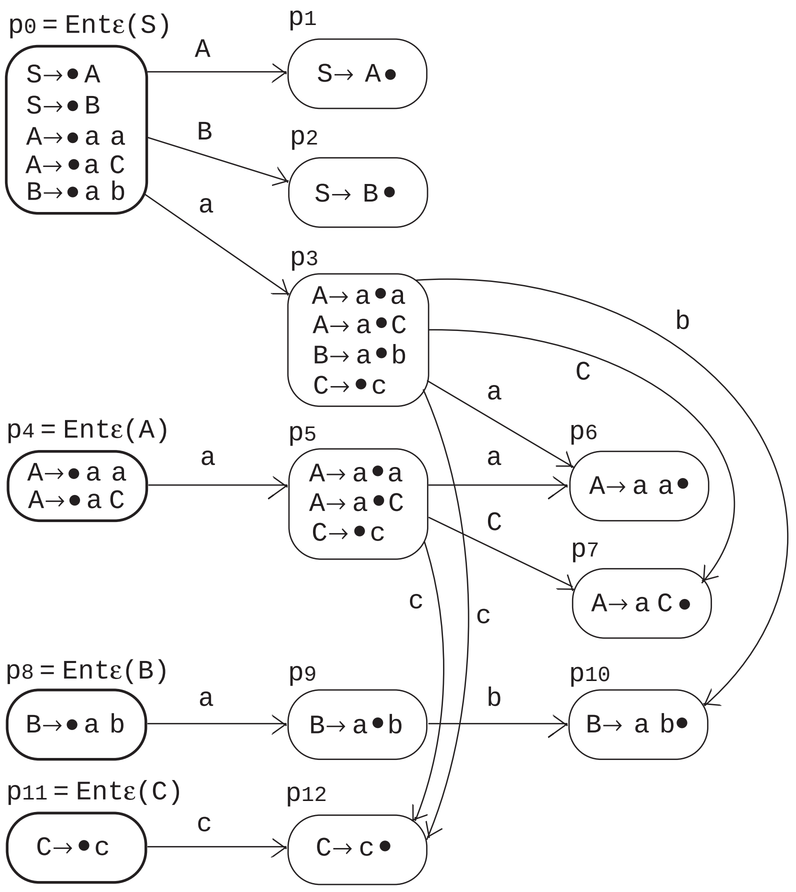


Figure 4.5: LR(0) graph for G_5

(proof) Suppose $Inc(q)$ is not empty. The existence of maximal set in $Inc(q)$ is trivial, because the co-domain of Inc is $Power(Power(Item))$ which is finite. If there exist distinct maximal sets in $Inc(q)$, say U_1 and U_2, there must be $p \in U_1$ and $p'_1, \ldots, p'_k \in U_2$, s.t., $p \neq p'_j$ and $p \cap p'_j \neq \phi$ for each $j = 1, \ldots, k$, moreover $p \subset p'_1 \cup \cdots \cup p'_k$. By Lemma 4.3.7, p consists of some other states, thus $Inc'(p) \neq \phi$. It is contradiction. So, the maximal set is unique.//

Definition 4.3.10 (*MaxInc*)

$$MaxInc(q) = \begin{cases} \phi & \text{iff } Inq(q) = \phi \\ \mu(= \text{maximals of } Inc(q)) & \text{otherwise.} \end{cases}$$

Example 4.3.11 (An Example of *Inc* and *MaxInc*) *Figure 4.5 illustrates $SC(lr(G_5))$ for CFG $G_5 = (\{S, A, B, C\}, \{a, b, c\}, \{S \to A, S \to B, A \to a\,a, A \to a\,C, B \to a\,b, C \to c\}, S)$. Values of Inc', Inc and $MaxInc$ for each states are as follows.*

	p_0, p_1, p_2	p_3	$p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}$
Inc'	ϕ	$\{\{p_5, p_9, p_{11}\}, \{p_5, p_9\}\}$	ϕ
Inc	ϕ	$\{\{p_5, p_9, p_{11}\}, \{p_5, p_9\}\}$	ϕ
$MaxInc$	ϕ	$\{p_5, p_9\}$	ϕ

Clearly, because p_1, p_2, p_4, p_6, p_7, p_8, p_9, p_{10}, p_{11} and p_{12} are unique states, all values of Inc', Inc and $MaxInc$ for their states are ϕ. For p_5, while $p_{11} \subset p_5$ holds, $Inc'(p_5) = Inc(p_5) = MaxInc(p_5) = \phi$, because p_5 contains unique item $A \to a \bullet a$. By same reason, $Inc'(p_0) = Inc(p_0) = MaxInc(p_0) = \phi$. p_3 is sole synthetic state in this example.

Of course, to calculate above values, information of inclusion relations must be held, as below.

	p_0	p_1, p_2,p_4,p_6, p_7, p_8, p_9, p_{10}, p_{11}, p_{12}	p_3	p_5
$\subset$	p_4, p_8	ϕ	p_5, p_9, p_{11}	p_{11}

Lemma 4.3.12 (Evolution on *MaxInc*)
For given $q_1, q_2 \in ES$, where $q_1 \cup q_2 \in ES$,

$$MaxInc(q_1 \cup q_2) = \begin{array}{rl} MaxInc(q_1) & \text{if } q_2 \subset q_1 \\ MaxInc(q_2) & \text{if } q_1 \subset q_2 \\ \{q_1, q_2\} & \text{if } MaxInc(q_1) = MaxInc(q_2) = \phi \\ \text{maximals of } MaxInc(q_1) \cup \{q_2\} & \text{if } MaxInc(q_1) \neq \phi \text{ and } MaxInc(q_2) = \phi \\ \text{maximals of } \{q_1\} \cup MaxInc(q_2) & \text{if } MaxInc(q_1) = \phi \text{ and } MaxInc(q_2) \neq \phi \\ \text{maximals of } MaxInc(q_1) \cup MaxInc(q_2) & \text{otherwise.} \end{array}$$

(proof) On cases of $q_2 \subset q_1$ or $q_1 \subset q_2$, they are trivial. On the case of $MaxInc(q_1) = MaxInc(q_2) = \phi$, it is obvious that $\phi \neq \{q_1, q_2\} \in Inc'(q_1 \cup q_2)$, $\{q_1, q_2\} \in Inc(q_1 \cup q_2)$. We obtain results $q_1, q_2 \in MaxInc(q_1 \cup q_2)$ and $q_3 \in MaxInc(q_1 \cup q_2) \Rightarrow q_3 = q_1$ or $q_3 = q_2$ by same discussion as in the proof of Lemma 4.3.9. Thus $MaxInc(q_1 \cup q_2) = \{q_1, q_2\}$. Equations remained are obtained same as above.//

Corollary 4.3.13 *For given $\sigma = \{q_1, q_2, \ldots, q_n\} \subset ES$, we assume $\forall q_i, q_j \in \sigma$, $q_i \not\subset q_j$. Let $\sigma_1 = \{q_i \in \sigma \mid Inc'(q_i) = \phi\}$, $\sigma_2 = \sigma \setminus \sigma_1$, then*

$$MaxInc(\bigcup \sigma) = \text{maximals of } (\sigma_1 \cup \bigcup_{q_i \in \sigma_2} MaxInc(q_i)).$$

Lemma 4.3.14 *For given $MonoG(Z \to X_1 \cdots X_n) = (V \cup T, Q_2, \zeta_2, \{Z \to \bullet X_1 \cdots X_n\}, *)$, and we assume that $Q_2 = \{q_1, \ldots, q_{n+1}\}$ those contents are same of them in the proof of Lemma 4.3.4, then,*

$$(q, Z, i) \in Flow \Leftrightarrow q_i \subset q.$$

(proof) Same discussion in Lemma 4.3.4.//

Lemma 4.3.15 *For each state q of $MonoG(Z \to X_1 \cdots X_n)$, $MaxInc(q) = \phi$.*

(proof) It is obvious from the fact that for each state $q_i \in Q_2$, if $q' \subset q_i$ then $Z \to X_1 \cdots \bullet X_i \cdots X_n \notin q'$.//

So far, we have shown definitions and some features of *MaxInc* and inclusion relations. Because our purpose is to construct an incremental construction method for LALR(1) parser, we must argue on a renewal method for *MaxInc*. Suppose a situation that CFG G and LR(0) graph $SC(lr(G))$ for G are given and we are going to calculate LR(0) graph for G' which is G augmented with a new production rule $Z \to X_1 \cdots X_n$. Let Q_1 be the set of states of $SC(lr(G))$, and Q_2 that of $MonoG(Z \to X_1 \cdots X_n)$. During computing $\mathit{Fus}(SC(lr(G)), MonoG(Z \to X_1 \cdots X_n))$, in order to express inclusion relation between state and values of *MaxInc*, we use $Q_1 \cup Q_2$ as a set of states. Because we assume that $Z \to X_1 \cdots X_n$ is a really new production rule, during computing, there happens no contradiction. To complete fusion operation, we have to rearrange inclusion relations and value of *MaxInc* for each state, because some states in $Q_1 \cup Q_2$ would be unreachable from any entrance $Ent\varepsilon(X)$, and some states would be newly introduced. Arguments following concern to this rearrangement. In the following, we assume that $G = (V, T, P, S)$, $G' = (V, T, P', S)$, where $P' = P \cup \{Z \to X_1 \cdots X_n\}$. And also, a set of all reachable states from an entrance of graph A is denoted by ES_A. In order to emphasize that it is on graph A, inclusion relation is denoted by $\subset_A$ and *MaxInc* is denoted by MaxInc_A. However, as intermediate values, notations $\subset_{(Q_1 \cup Q_2)}$ and $\mathit{MaxInc}_{(Q_1 \cup Q_2)}$ are used, for the purpose described above. In the following discussions, firstly, $\subset_{(Q_1 \cup Q_2)}$ and $\mathit{MaxInc}_{(Q_1 \cup Q_2)}$ are calculated from $\subset_{SC(lr(G))}$, $\mathit{MaxInc}_{SC(lr(G))}$, $\subset_M$ and MaxInc_M, where $M = MonoG(Z \to X_1 \cdots X_n)$. And then $\subset_A$, MaxInc_A are defined, where $A = \mathit{Fus}(SC(lr(G)), MonoG(Z \to X_1 \cdots X_n))$. Finally, it is proved that $\subset_A$ and MaxInc_A are equivalent to $\subset_{SC(lr(G'))}$ and $\mathit{MaxInc}_{SC(lr(G'))}$, respectively.

Definition 4.3.16 ($\mathit{MaxInc}_{(Q_1 \cup Q_2)}$, $\subset_{(Q_1 \cup Q_2)}$)
For each state $q \subset Q_1 \cup Q_2$ of given $A = \mathit{Fus}(SC(lr(G)), MonoG(Z \to X_1 \cdots X_n))$, where $Q_1 = Item(P)$, $Q_2 = Item(\{Z \to X_1 \cdots X_n\})$, we assume that $q = q_1 \cup q_2 \cup \cdots \cup q_k$ ($\forall i$, $q_i \subset Q_1$ or $q_i \subset Q_2$), and, let U_q is a set of maximal elements of $\{q_i \mid q_i \neq q, MaxInc(q_i) = \phi, i = 1, \ldots, k\}$. Then, $\mathit{MaxInc}_{(Q_1 \cup Q_2)}$, which holds intermediate values in order to calculate $\mathit{MaxInc}_{SC(lr(G'))}$, where G' is the grammar augmented to G by $MonoG(Z \to X_1 \cdots X_n)$, is defined as,

$$\mathit{MaxInc}_{(Q_1 \cup Q_2)}(q) = \text{maximals of } (U_q \cup \bigcup_{q'' \in \{q_1, \ldots, q_n\} \setminus U_q} \mathit{MaxInc}(q''))$$

(in the expression, MaxInc means either $\mathit{MaxInc}_{SC(lr(G))}$ or MaxInc_M in according to its argument, where $M = MonoG(Z \to X_1 \cdots X_n)$). We define inclusion relation $\subset_{(Q_1 \cup Q_2)}$ so as,

1) *$\mathit{MaxInc}_{(Q_1 \cup Q_2)}(q) = \phi$ and $\mathit{MaxInc}_{(Q_1 \cup Q_2)}(q') = \phi$ and $(q, q') \in \subset$*

$$\Rightarrow (q, q') \in \subset_{(Q_1 \cup Q_2)},$$

2) *$\mathit{MaxInc}_{(Q_1 \cup Q_2)}(q) = \phi$ and $\mathit{MaxInc}_{(Q_1 \cup Q_2)}(q') \neq \phi$
and $\exists p' \in \mathit{MaxInc}_{(Q_1 \cup Q_2)}(q'), (q, p') \in \subset$*

$$\Rightarrow (q, q') \in \subset_{(Q_1 \cup Q_2)},$$

3) *$MaxInc_{(Q_1 \cup Q_2)}(q) \neq \phi$ and $MaxInc_{(Q_1 \cup Q_2)}(q') = \phi$*
and $\forall p \in MaxInc_{(Q_1 \cup Q_2)}(q), (p, q') \in \subset$

$$\Rightarrow (q, q') \in \subset_{(Q_1 \cup Q_2)},$$

4) *$MaxInc_{(Q_1 \cup Q_2)}(q) \neq \phi$ and $MaxInc_{(Q_1 \cup Q_2)}(q') \neq \phi$*
and $\forall p \in MaxInc_{(Q_1 \cup Q_2)}(q), \exists p' \in MaxInc_{(Q_1 \cup Q_2)}(q'), (p, p') \in \subset$

$$\Rightarrow (q, q') \in \subset_{(Q_1 \cup Q_2)},$$

$\subset_{(Q_1 \cup Q_2)}$ is a minimum set which satisfies above conditions.
(also, in these expressions, $\subset$ means either $\subset_{SC(lr(G))}$ or $\subset_M$, where $M = MonoG(Z \rightarrow X_1 \cdots X_n)$).

Definition 4.3.17 (*$MaxInc_A$*, $\subset_A$)
For each $q \in ES_{SC(lr(G'))}$ and each $q' \in MaxInc_{(Q_1 \cup Q_2)}(q)$, let $PreInc(q, q')$ be

$$PreInc(q, q') = \{q'' \in ES_{SC(lr(G'))} \mid q' \subset_{(Q_1 \cup Q_2)} q'' \text{ and } q'' \subset_{(Q_1 \cup Q_2)} q\},$$

then $MaxInc_A$ is defined as,

$$MaxInc_A(q) = \begin{cases} \phi & \text{if } \exists q' \in MaxInc'(q), \\ & s.t., PreInc(q, q') = \phi \\ \text{maximals of } (\bigcup_{q' \in MaxInc'(q)} PreInc(q, q')) & \text{otherwise.} \end{cases}$$

where $MaxInc' = MaxInc_{(Q_1 \cup Q_2)}$. And also, $\subset_A$ is defined simply as

$$\subset_A = \subset_{(Q_1 \cup Q_2)} \cap (ES_A \times ES_A).$$

Theorem 4.3.18

$$\begin{aligned} \subset_A &= \subset_{SC(lr(G'))}, \\ MaxInc_A &= MaxInc_{SC(lr(G'))} \end{aligned}$$

(proof) From Theorem 4.2.14, $SC(lr(G'))$ and $A = Fus(SC(lr(G)), MonoG(Z \rightarrow X_1 \cdots X_n))$ are isomorphic. To mention precisely, we must argue that this proof is on the correspondence given by the isomorphism. However, to reduce description, states p of $SC(lr(G'))$, which are associated to state p' of $Fus(SC(lr(G)), MonoG(Z \rightarrow X_1 \cdots X_n))$ by the isomorphism, are identified with p'.

Let $SC(lr(G')) = (V \cup T, Power(Item(P')), \zeta, s, *)$. From the definitions of *Fus* and *MonoG*, for each states $q \subset q' \subset ES_{SC(lr(G'))}$, we can write $q = q_1 \cup \cdots \cup q_k$, $q' = q'_1 \cup \cdots \cup q'_{k'}$ ($q_i \subset Q_1$ or $q_i \subset q_2$, $q'_j \subset Q_1$ or $q'_j \subset Q_2$). From the assumption $q \subset q'$, for each $i = 1, \ldots, k$, we can select a minimum set $\{q'_{j_1}, \ldots, q'_{j_m}\}$ on q' so that $q_i \subset q'_{j_1} \cup \cdots \cup q'_{j_m}$. Of course, there might be alternative selections. If $m > 1$, by Lemma 4.3.7, $MaxInc(q_i) \neq \phi$ must hold. This case corresponds to the case 4) in Definition 4.3.16. It

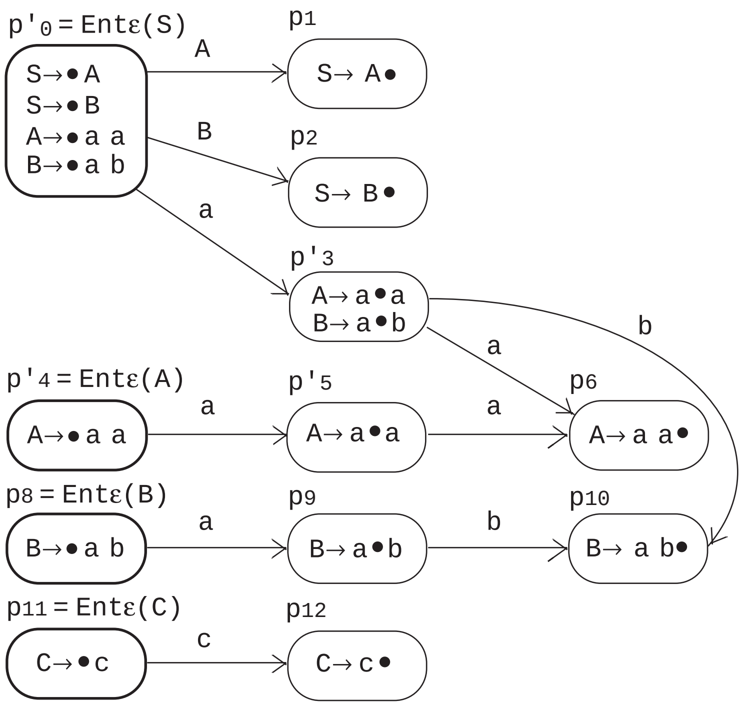


Figure 4.6: Before Evolution (MaxInc)

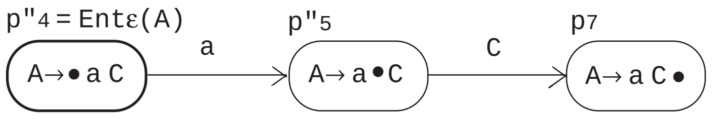


Figure 4.7: Augmenting *MonoG* for Evolution (MaxInc)

is easy to see that the other cases correspond to one of 1),. . ., 3) of Definition 4.3.16, with same discussions. Hence $(q, q') \in \subset_{SC(lr(G'))} \Rightarrow (q, q') \in \subset_A$ is obtained straightforwardly from the definitions. Conversely, $(q, q') \in \subset_A \Rightarrow (q, q') \in \subset_{SC(lr(G')}$ is obtained easily by same way.

Finally, we can conclude that $MaxInc_A = MaxInc_{SC(lr(G'))}$, because $SC(lr(G'))$ and $Fus(SC(lr(G)), MonoG(Z \to X_1 \cdots X_n))$ have same inclusion relation.//

Example 4.3.19 (Evolusion of MaxInc) *We consider a situation that from two LR(0) graphs Figure 4.6, say $SC(lr(G'_5))$, and Figure 4.7, LR(0) graphs Figure 4.5, i.e. of Example 4.3.11, is obtained by Fus operation. The values of Inc', Inc and MaxInc for Figure 4.6 is provided in Table 4.1, and for Figure 4.7 in Table 4.2. $Fus(SC(lr(G'_5)), MonoG(A \to a\ C))$ is given in Figure 4.8. The constructions of states are given in Table 4.3.*

From Lemma 4.3.12, Corollary 4.3.13 and Definition 4.3.16, $MaxInc(p_0) = MaxInc(p'_0 \cup p''_4)$, $MaxInc(p_3) = MaxInc(p'_3 \cup p''_5 \cup p_{11})$, $MaxInc(p_4) = MaxInc(p'_4 \cup p''_4)$ and $MaxInc(p_5) = MaxInc(p'_5 \cup p''_5 \cup p_{11})$ are calculated indivisually. Results of the calculations are shown

	p'_0, p_1, p_2	p'_3	p'_4, p'_5, p_6, p_8, p_9, p_{10}, p_{11}, p_{12}
Inc'	ϕ	$\{\{p'_5, p_9\}\}$	ϕ
Inc	ϕ	$\{\{p'_5, p_9\}\}$	ϕ
$MaxInc$	ϕ	$\{p'_5, p_9\}$	ϕ

Table 4.1: Inc', Inc and $MaxInc$ for Figure 4.6

	p''_4, p''_5, p_7
Inc'	ϕ
Inc	ϕ
$MaxInc$	ϕ

Table 4.2: Inc', Inc and $MaxInc$ for Figure 4.7

p_0	$=$	$p'_0 \cup p''_4$	p_1	$=$	p_1	p_2	$=$	p_2
p_3	$=$	$p'_3 \cup p''_5 \cup p_{11}$	p_4	$=$	$p'_4 \cup p''_4$	p_5	$=$	$p'_5 \cup p''_5 \cup p_{11}$
p_6	$=$	p_6	p_7	$=$	p_7	p_8	$=$	p_8
p_9	$=$	p_9	p_{10}	$=$	p_{10}	p_{11}	$=$	p_{11}
p_{12}	$=$	p_{12}						

Table 4.3: Construction of States for $SC(lr(G_5))$

$MaxInc(p_0)$	$MaxInc(p_3)$	$MaxInc(p_4)$	$MaxInc(p_5)$
$\{p'_0, p''_4\}$	$\{p'_5, p''_5, p_9, p_{11}\}$	$\{p'_4, p''_4\}$	$\{p'_5, p''_5, p_{11}\}$

Table 4.4: $MaxInc_{(Q_1 \cup Q_2)}$

$MaxInc_A(p_0)$	$MaxInc_A(p_3)$	$MaxInc_A(p_4)$	$MaxInc_A(p_5)$
ϕ	$\{p_5, p_9\}$	ϕ	ϕ

Table 4.5: $MaxInc_A$ for $SC(lr(G_5))$

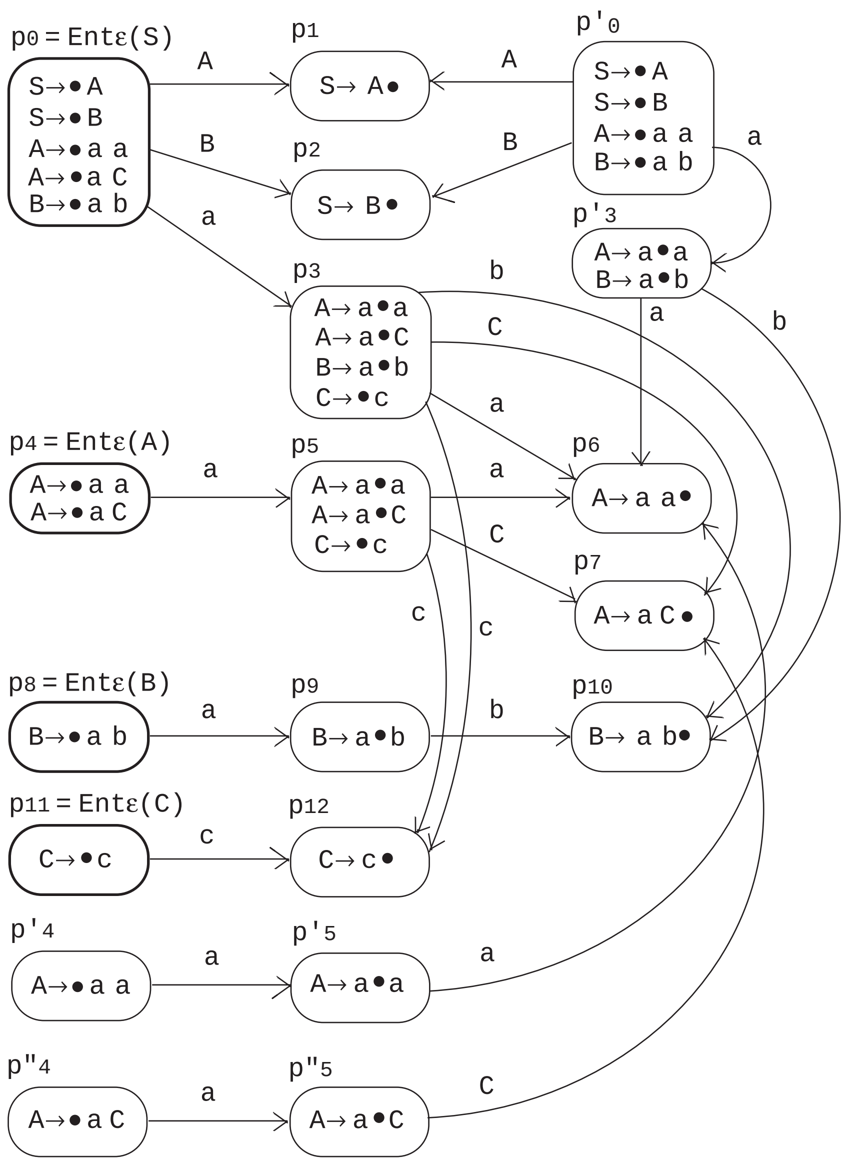


Figure 4.8: Result of *Fus* on Figure 4.6 and Figure 4.7

in Table 4.4. After the calculations of all of these values are completed, values of $\subset_{(Q_1 \cup Q_2)}$ *are calculated.*

Using values of $MaxInc_{(Q_1 \cup Q_2)}$*, finally we obtain values of* $MaxInc_A$ *by Definition 4.3.17, as shown in Table 4.5. The calculation of these values requires to seek states which are included in* $MaxInc_{(Q_1 \cup Q_2)}(q)$ *for each state* $q \in ES_{SC(lr(G_5))}$*, i.e. PreInc. For example, while* $MaxInc_{(Q_1 \cup Q_2)}(p_0) = \{p'_0, p''_4\}$*, because both of* p'_0 *and* p''_4 *are not included in* $ES_{SC(lr(G_5))}$*,* $MaxInc_A(p_0) = \phi$*. For* $MaxInc(p_3)$*,*

$$\bigcup_{q' \in MaxInc'(p_3)} PreInc(p_3, q') = \{p_5, p_9, p_{11}\}$$

is obtained. However, because $p_{11} \subset p_5$*,* $MaxInc(p_3) = \{p_5, p_9\}$*.*

4.4 Discussions on Look Ahead Symbol Set

In this section, we discuss on a way of calculation of Look Ahead Symbols Sets (LA) on LALR(1) parser. Typical point of the method discussed here is that the calculation is achieved without use of any item sets. Of course, some informations concerning to item sets is used. To illustrate the idea, firstly we observe a process of calculation of LA in conventional algorithm. Suppose LR(0) graph is given, and, we stand on a point of calculation of LA. An item $Y \to \alpha \bullet X\ \beta$ is included in some state q with LA θ, then q must include an item $X \to \bullet\gamma$ induced from $Y \to \alpha \bullet X\ \beta$, so, LA of it includes *First*(β) and , if ε is derived from β, θ is also included by the LA. *First*(β) and the condition 'if ε is derived from β' and θ possibly change as a result of augmentation of a production rule. Firstly, we focus on the condition 'if ε is derived from β'.

In simplest cases, if β contains a terminal symbol, ε is never derived from β, and if β = Z for some syntactic variable Z, the ε-productivity depends on the ε-productivity of Z. In general case, i.e. $\beta = Z_1 \cdots Z_n$, ε-productivity of β depends on ε-productivities of Z_1 to Z_n. So the condition must include,

$$\varepsilon\text{-}producible(Z_1) \wedge \cdots \wedge \varepsilon\text{-}producible(Z_n). \tag{4.2}$$

The order of predicates is not important. Suppose a grammar contains a production rule $Z_1 \to Y_1 \cdots Y_m$, if ε is derived from all of Y_k, and from all of Z_2 to Z_n, ε is derived from β. So the condition also must contain,

$$\varepsilon\text{-}producible(Y_1) \wedge \cdots \wedge \varepsilon\text{-}producible(Y_m) \wedge \varepsilon\text{-}producible(Z_2) \cdots \wedge \varepsilon\text{-}producible(Z_n). \tag{4.3}$$

If one of (4.2) and (4.3) satisfies, the condition becomes true. Thus, (4.2) and (4.3) must be composed by conjunction. To express such a condition, we introduce *Dependency Domain* (DD), which is defined in Definition 4.4.1. The condition above is denoted by $E\Delta(\beta)$ in following discussions. In most cases described in the algorithems of our method, $E\Delta$ is not computed directly from production rules, the exceptional case is on construction of *MonoG*. As a method for compute $E\Delta$, we present some kind of recurrence relations of $E\Delta$ on production rule set, shown in Theorem 4.4.9, using a few operations defined in Definition 4.4.3 and 4.4.8.

In addition to DD, we introduce $IndT\Delta$, $IndV\Delta$ and $Link\Delta$ in order to express an index, named $Ind\Delta$ for LA, instead of *First*. $IndT\Delta$ and $IndV\Delta$ concern to inner state production of LA, and $Link\Delta$ concerns to propagation of LA from another states. Here, we illustrate only our points of view. Precises are discussed through Definition 4.4.13 to Definition 4.4.23. In conventional calculation method for LA asigns LA to each items in each states. However, for example, in above discussion, $\mathit{First}(\beta)$ and θ is also asigned to item $X \to \bullet\gamma'$, if exists in the state. The item-wise asignment is an excessive quantitative choice. LA is common on each $\varepsilon Item(X)$. $IndT\Delta$ and $IndV\Delta$ give indeces of LA flow out to $\varepsilon Items$ which are induced in the state, with some conditions by DD. We also give a few operations in order to calculate their values in fully incremental manner.

The advantage of the method using $IndT\Delta$, $IndV\Delta$ and $Link\Delta$ are not only that there is no need to keep item set information, but also that there is an operation so as to decrease space usage, as a result, which operation increases efficiency on time to proceed computations. For example, constructing a parser for RCFG, when construction of initial parser completes, information about ε-productivity on syntactic variables other than those of $f(D)$, i.e. dynamically definable variables, has no need to be kept. In such a situation, we can eliminate conditions which contains ε-productivity on X. The operation, named *Domain Restriction*, is stated in Definition 4.4.30.

Through discussions in this section, we essentially depend on the grounds of a proposition, i.e.

$$\mathit{First}(\alpha\alpha\beta) = \mathit{First}(\alpha\beta),$$

stated as Proposition 2.6.8, explicitly or implicitly. This equation is one of typical properties of CFG. Using this property, we succeeded to give simple operations on $E\Delta$, $Top\Delta$, $Dep\Delta$ and $Link\Delta$. $Top\Delta$ and $Dep\Delta$ are functions to calculate values of $IndT\Delta$ and $IndV\Delta$, respectively.

4.4.1 Dependency Domain

From now on, notions which are introduced for the method of incremental construction of LALR(1) parser are stated.

Definition 4.4.1 (Dependency Domain (DD))
For any H_1, $H_2 \subset \mathit{Power}(\Omega)$, where Ω is an arbitrary finite set, firstly we define a pre-order $<$, such as,

$$H_1 < H_2 \Leftrightarrow \forall U \in H_2, \exists V \in H_1, V \subset U,$$

also define equation $=$,

$$H_1 = H_2 \Leftrightarrow H_1 < H_2, H_2 < H_1,$$

then, we obtain a quotient set of $\mathit{Power}(\mathit{Power}(\Omega))$ by $=$,i.e., $\mathit{Power}(\mathit{Power}(\Omega))\ /\ =$, denoted by $[\mathit{Power}(\mathit{Power}(\Omega))]$. We call $[\mathit{Power}(\mathit{Power}(\Omega))]$ Dependency Domain on Ω, or Disjunction Form on Ω. Additionally, for each $H \in [\mathit{Power}(\mathit{Power}(\Omega))]$, $[H]$ denotes the class to which H belongs.

Note: For each elements of $[\mathit{Power}(\mathit{Power}(\Omega))]$ are regarded as *Disjunctive Normal Form without Negative Literals.* On this interpretation, Ω is assumed to be a set of *Propositional Variables.* The element $\{\phi\} \in [\mathit{Power}(\mathit{Power}(\Omega))]$ can be interpreted as *Absolutely True*, and ϕ *Absolutely False*, which will be denoted by **true** and **false**, respectively.

Proposition 4.4.2 *For any $H \subset \mathit{Power}(\Omega)$,*

$$\eta, \eta' \in H, \eta \subset \eta' (\eta \neq \eta') \Rightarrow [H] = [H \setminus \{\eta'\}].$$

(proof) Straightforward from the definition.//

Definition 4.4.3 (Convolution on DD)
For given $H, H' \in [Power(Power(\Omega))]$, we define Convolution Operation $*$ on H, H' as

$$H * H' = [\{\eta \cup \eta' \mid \eta \in H, \eta' \in H'\}].$$

Definition 4.4.4 (ε-Productivity Decision)
Suppose given CFG $G = (V, T, P, S)$ and given a subset of V, say V'. For any symbol sequence $\alpha \in (V \cup T)*$, εDerivatioin Judgement $E\Delta(\alpha)$ is defined as

$$\begin{aligned} E\Delta(\alpha) = [\{\{X_1, \ldots, X_n\} \subset V' \quad \mid \quad & \alpha \overset{*}{\Rightarrow} X_1 \cdots X_n, \\ & X_i \overset{*}{\not\Rightarrow} \varepsilon, for\, 1 \leq i \leq n\}]. \end{aligned}$$

The value of $E\Delta(\alpha)$ is of $[Power(Power(V'))]$. To emphasize the value of $E\Delta$ is under G or P, we write $E\Delta_G(\alpha)$ or $E\Delta_P(\alpha)$. Moreover, to emphasize the value of $E\Delta$ is over V', we write $E\Delta(\alpha) \mid V'$, $E\Delta_G(\alpha) \mid V'$, $E\Delta_P(\alpha) \mid V'$.

Example 4.4.5 *Let CFG $G_6 = (V_6, T_6, P_6, S)$, where*

$$\begin{aligned} V_6 &= \{S, X, Y, A, B, C\}, \\ T_6 &= \{a, b, c\}, \\ P_6 &= \{\, S \to X\,Y,\ X \to A,\ X \to A\,X, \\ &\quad Y \to B,\ Y \to B\,Y, \\ &\quad Y \to C,\ Y \to C\,Y, \\ &\quad A \to a,\ B \to b,\ C \to c\,\}, \end{aligned}$$

$L(G_6) = (a+)(b + c)+$. No ε-derivation is achieved from any syntactic variables on this grammar. Intuitively, if new production rule $B \to \varepsilon$ is added to the grammar, B becomes ε-producible, and also Y ε-producible. So, $\{B\}$ must be included in $E\Delta(Y)$. From same reason, $\{C\} \in E\Delta(Y)$. Because syntactic variable sequences derived from Y are $(B+C)+$ $= \{B, C, BB, BC, CB, CC, \ldots\}$, $E\Delta(Y) = [H_1]$, where $H_1 = \{\{Y\}, \{B\}, \{C\}, \{B, C\}\}$. From the definition of DD, $E\Delta(Y) = [H_2]$, where $H_2 = \{\{Y\}, \{B\}, \{C\}\}$, because $H_2 \subset H_1$ indicates $H_1 < H_2$ from the definition of DD (Definition 4.4.1), and for $\{B, C\} \in H_1$, $\exists\{B\} \in H_2$, $\{B\} \subset \{B, C\}$, thus $H1$ and H_2 is equivalent.

The values of $E\Delta$ for each syntactic variables are given in the table, as follows.

	S	X	Y
$E\Delta$	$\{\{S\}, \{X, Y\}, \{A, Y\}, \{X, B\}, \{X, C\}, \{A, B\}, \{A, C\}\}$	$\{\{X\}, \{A\}\}$	$\{\{Y\}, \{B\}, \{C\}\}$

	A	B	C
$E\Delta$	$\{\{A\}\}$	$\{\{B\}\}$	$\{\{C\}\}$

Proposition 4.4.6 $E\Delta(\alpha\beta) = E\Delta(\alpha) * E\Delta(\beta)$.

(proof) Straightforward from the definition of Convolution on DD (Definition 4.4.3) and the definition of $E\Delta$ (Definition 4.4.4).//

Proposition 4.4.7 *If given $\alpha \in (V \cup T)*$ is ε-producible on G, then $E\Delta(\alpha) = [\{\phi\}]$.*

(proof) If α is ε-producible, then $\phi \in \{\{X_1, \ldots, X_n\} \subset V' \mid \alpha \overset{*}{\Rightarrow} X_1 \cdots X_n, X_i \overset{*}{\not\Rightarrow} \varepsilon, \text{for } 1 \leq i \leq n\}$ as a vacuous case that $n = 0$.//

Definition 4.4.8 (Substitution on DD)
For given $H, H' \in [Power(Power(\Omega))]$ and given $X \in \Omega$, Substitution of X in H' to H is defined as

$$\begin{aligned} H'[H/X] = [\{\eta'' \subset \Omega \quad \mid \quad & \eta' \in H', \eta \in H, \\ & X \notin \eta' \Rightarrow \eta'' = \eta', \\ & X \in \eta' \Rightarrow \eta'' = \eta \cup \eta' \setminus \{X\}\}]. \end{aligned}$$

Also, Substition of X in $E\Delta(\alpha)$ to $H \in [Power(Power(V'))]$ is defined as

$$\begin{aligned} E\Delta(\alpha)[H/X] = [\{\eta'' \subset V' \quad \mid \quad & \eta' \in E\Delta(\alpha), \eta \in H, \\ & X \notin \eta' \Rightarrow \eta'' = \eta', \\ & X \in \eta' \Rightarrow \eta'' = \eta \cup \eta' \setminus \{X\}\}]. \end{aligned}$$

Theorem 4.4.9 (Evolution of $E\Delta$)
For each production rule set $P' = P \cup \{X \to \gamma\}$, $\alpha \in (V \cup T)*$,

$$E\Delta_{P'}(\alpha) = [E\Delta_P(\alpha) \cup E\Delta_P(\alpha)[E\Delta_P(\gamma)/X]].$$

(proof) It is obvious that $E\Delta_P(\alpha) \cup E\Delta_P(\alpha)[E\Delta_P(\gamma)/X]$ denotes ε-producible condition for α on the restriction that use of production rule $X \to \gamma$ in each derivations on P' is restricted to at most once. Hence, it is clear that $E\Delta_P(\alpha) \cup E\Delta_P(\alpha)[E\Delta_P(\gamma)/X] \subset E\Delta_{P'}(\alpha)$. However, a set of all syntactic variables which occur in each symbol sequences on each derivations from α never increases, whether $X \to \gamma$ is used once or more than twice. Thus, each element of $E\Delta_{P'}(\alpha)$ must be included in $E\Delta_P(\alpha) \cup E\Delta_P(\alpha)[E\Delta_P(\gamma)/X]$.//

Example 4.4.10 (Evolution on G_6)
Suppose a situation that a new production rule $A \to \varepsilon$ is augmented to G_6 given in Example 4.4.5 above. From Definition 4.4.4, $E\Delta_{P_6}(\varepsilon) = \{\phi\}$. Using this value, values of $E\Delta_{P_6}[\{\phi\}/A]$ for each syntactic variables are as below.

	S	X	Y
$E\Delta_{P_6}[\{\phi\}/A]$	$\{\{S\},\{Y\},\{B\},\{C\}\}$	$\{\phi\}$	$\{\{Y\},\{B\},\{C\}\}$

	A	B	C
$E\Delta_{P_6}[\{\phi\}/A]$	$\{\phi\}$	$\{\{B\}\}$	$\{\{C\}\}$

From the definition of Substitution on $E\Delta$ (Definition 4.4.8), $E\Delta_{P_6}(S)[\{\phi\}/A] = [\{\{S\}, \{X,Y\}, \{Y\}, \{X,B\}, \{X,C\}, \{B\}, \{C\}\}]$. However, from the definition of DD (Definition 4.4.1), the elements $\{X,Y\}$, $\{X,B\}$ and $\{X,C\}$ can be omitted. To use theses values, values of $E\Delta_{P'_6}$ for each syntactic variables are calculated, where $P'_6 = P_6 \cup \{A \to \varepsilon\}$. On this case, they are same to $E\Delta_{P_6}[\{\phi\}/A]$.

Before continue the discussion, we summarize operations and properties on $E\Delta$. As an interpretation of DD given by the definition of $E\Delta$, DD can be interpreted to Disjunction Normal Form without Negative Literals.

Logical Sum	$[H \cup H']$
Logical Product	$H * H'$
Substitution over$Z$	$H'[H/Z]$
Evolution by$Z \to \gamma$	$[H \cup H[E\Delta(\gamma)/Z]]$

Each operations are defined in denotational forms of sets. However, it is easy to show that the implementations of them can be quite efficient. Imprementation problem is discussed in Section 4.7. Domain restriction operation for DD is also prepared other than these operations, which is discussed in Definition 4.4.30, with same operations on another elements.

4.4.2 $Ind\Delta$: an index for LA

From now on, an index for LA, say $Ind\Delta$, is discussed. The index is associated to each state q for each syntactic variable Y, i.e. $Ind\Delta(q,Y)$, which consists of three values $IndT\Delta$, $IndV\Delta$ and $Link\Delta$. Domains for each are as follows,

$$\begin{aligned}
IndT\Delta &: T \to [Power(Power(V'))], \\
IndV\Delta &: V \to [Power(Power(V'))], \\
Link\Delta &\subset \mathbf{N} \times V \times [Power(Power(V'))].
\end{aligned}$$

These form a core information of indeces for LA on each states. These three values are associated to each $\varepsilon Item(Y)$ in each state q, the triple is denoted by $Ind\Delta(q,Y) = (IndT\Delta, IndV\Delta, Link\Delta)$ defined in Definition 4.4.24 and Lemma 4.4.26 below. The meaning is that, in some context which leads to the state q, then a terminal symbol a follows to each $\varepsilon Item(Y)$ in q, when grammar satisfies the condition $IndT\Delta(a)$, a syntactic variable X follow to each $\varepsilon Item(Y)$ in q, when grammar satisfies the condition $IndV\Delta(X)$, and, if $(k,X,H) \in Link\Delta$ then all of LA which flows to X on the state preceding of q by k steps is propageted to the LA for $\varepsilon Item(Y)$ on q, when grammar satisfies the condition H. Our purpose is to state a scheme to calculate these data in incremental manner. To achieve it, we introduce functions $Top\Delta$ and $Dep\Delta$ which are similar notions to *First*, except that the value of $Top\Delta$ and $Dep\Delta$ have conditions on ε-productivity. Types of the functions are,

$$\begin{aligned}
Top\Delta &: (V \cup T)* \to (T \to [Power(Power(V'))]), \\
Dep\Delta &: (V \cup T)* \to (V \to [Power(Power(V'))]).
\end{aligned}$$

In order to define values of $Ind\Delta$, the functions $Top\Delta$ and $Dep\Delta$ are used.

Suppose a derivation sequence $\alpha \overset{*}{\Rightarrow} \gamma a \beta$, if ε is derived from γ, $a \in \mathit{First}(\alpha)$, if not, a might not be in $\mathit{First}(\alpha)$. Considering incremental calculation of LA, the condition of ε-productivity of γ may change. So, it is useful to supplement a condition to each symbol. Intuitively, this idea leads us to generalize the notion $\mathit{First} : (V \cup T)* \rightarrow \mathit{Power}(T)$ to $Fist\Delta : (V \cup T)* \rightarrow (T \rightarrow [\mathit{Power}(\mathit{Power}(V))])$. However, $\mathit{First}\Delta$ is not so easy to be reconstructed on augmenting a new production rule. So, we devide $\mathit{First}\Delta$ into two notions, i.e. $Top\Delta$ and $Dep\Delta$. The easiness of reconstruction of $Top\Delta$ and $Dep\Delta$ are shown in Theorem 4.4.19. Intuitively, $Dep\Delta(\alpha)(X)$ gives a condition so that if $\alpha \overset{*}{\Rightarrow} \gamma X \beta$, then ε-productivity of γ is included in the condition. To use values of $Top\Delta(\alpha)$, $Dep\Delta(\alpha)$ and $Top\Delta(\gamma)$ for each $Y \in V$ and for each $Y \rightarrow \gamma \in P$, it is easy to calculate $\mathit{First}(\alpha)$. A way to calculate $\mathit{First}(\alpha)$ from them is breafly discussed at the end of Section 4.5.

Following several definitions and propositions are given as basic operations.

Definition 4.4.11 (Operation $\cup$ on Domain $V \rightarrow$DD, $T \rightarrow$DD)
For each $\Lambda, \Lambda' : V \rightarrow [\mathit{Power}(\mathit{Power}(\Omega))]$, we define operation $\cup$, such as,

$$\Lambda \cup \Lambda' = \{(X, [H \cup H']) \mid X \in V, (X, H) \in \Lambda, (X, H') \in \Lambda'\},$$

in other words,

$$(\Lambda \cup \Lambda')(X) = [\Lambda(X) \cup \Lambda'(X)],$$

for arbitrary syntactic variable $X \in V$. Similarly, for each $\Lambda, \Lambda' : T \rightarrow [\mathit{Power}(\mathit{Power}(\Omega))]$, we define operation $\cup$, such as,

$$\Lambda \cup \Lambda' = \{(a, [H \cup H']) \mid a \in T, (a, H) \in \Lambda, (a, H') \in \Lambda'\},$$

in other words,

$$(\Lambda \cup \Lambda')(a) = [\Lambda(a) \cup \Lambda'(a)],$$

for arbitrary terminal symbol $a \in T$.

Definition 4.4.12 (Inclusion Relation on $V \rightarrow$DD, $T \rightarrow$DD)
For each $\Lambda, \Lambda' : V \rightarrow [\mathit{Power}(\mathit{Power}(\Omega))]$, if following condition satisfies, then $\Lambda \subset \Lambda'$ holds.

$$\forall X \in V, \Lambda(X) \subset \Lambda'(X).$$

Similarly, for each $\Lambda, \Lambda' : T \rightarrow [\mathit{Power}(\mathit{Power}(\Omega))]$,if following condition satisfies, then $\Lambda \subset \Lambda'$ holds.

$$\forall a \in T, \Lambda(a) \subset \Lambda'(a).$$

Moreover, occasionally, we denote $(\Lambda_t, \Lambda_d) \subset (\Lambda'_t, \Lambda'_d)$ for some $\Lambda_t, \Lambda'_t \in T \rightarrow [\mathit{Power}(\mathit{Power}(\Omega))]$ and $\Lambda_d, \Lambda'_d \in V \rightarrow [\mathit{Power}(\mathit{Power}(\Omega))]$ in the sense that $\Lambda_t \subset \Lambda'_t$ and $\Lambda_d \subset \Lambda'_d$.

Definition 4.4.13 ($Top\Delta$, $Dep\Delta$)
For given CFG $G = (V, T, P, S)$ and given $V' \subset V$, functions $Top\Delta : (V \cup T)* \rightarrow (T \rightarrow [\mathit{Power}(\mathit{Power}(V'))])$ and $Dep\Delta : (V \cup T)* \rightarrow (V \rightarrow [\mathit{Power}(\mathit{Power}(V'))])$ are defined as,

$$\begin{aligned} Top\Delta(\alpha) &= \{(a, \bigcup_{\alpha = \alpha' a \alpha''} E\Delta(\alpha')) \mid a \in T\}, \\ Dep\Delta(\alpha) &= \{(X, \bigcup_{\alpha \overset{*}{\Rightarrow} \alpha' X \alpha''} E\Delta(\alpha')) \mid X \in V\}. \end{aligned}$$

Similarly to $E\Delta$, to emphasize they are under G or under P, or over V', we write $Top\Delta_G$, $Dep\Delta_G$, $Top\Delta_P$, $Dep\Delta_P$, $Top\Delta_G \mid V'$, $Dep\Delta_G \mid V'$, $Top\Delta_P \mid V'$, $Dep\Delta_P \mid V'$, respectively.

Example 4.4.14 (Example of $Top\Delta$, $Dep\Delta$) *For CFG G_6 given in Example 4.4.5, the values of $Dep\Delta$ for each syntactic variables are given in followin table.*

	S	X	Y	A	B	C
$Dep\Delta(S)$	$\{\phi\}$	$\{\phi\}$	$\{\{X\},\{A\}\}$	$\{\phi\}$	$\{\{X\},\{A\}\}$	$\{\{X\},\{A\}\}$
$Dep\Delta(X)$	ϕ	$\{\phi\}$	ϕ	$\{\phi\}$	ϕ	ϕ
$Dep\Delta(Y)$	ϕ	ϕ	$\{\phi\}$	ϕ	$\{\phi\}$	$\{\phi\}$
$Dep\Delta(A)$	ϕ	ϕ	ϕ	$\{\phi\}$	ϕ	ϕ
$Dep\Delta(B)$	ϕ	ϕ	ϕ	ϕ	$\{\phi\}$	ϕ
$Dep\Delta(C)$	ϕ	ϕ	ϕ	ϕ	ϕ	$\{\phi\}$

From the definition of $Top\Delta$ (Definition 4.4.13), the values of $Top\Delta(\gamma)(x)$ for each syntactic variable sequence $\gamma \in V_6*$ and each terminal $x \in T_6$ are ϕ, because no terminal occurs immediately in γ. The function $Top\Delta$ is used to extract terminal symbols of 'inner produced' LAs. It is mostly used in the calculation of $MonoG$, which is achieved in Algorithm 4.5.1 shown in Section 4.5, and applied to symbol sequences of right-hand side of production rules.

Definition 4.4.15 (Restriction of $Top\Delta$ and $Dep\Delta$)
For given $H \in [Power(Power(V'))]$, Restriction of $Top\Delta$ and $Dep\Delta$, namely $Top\Delta * H$ and $Dep\Delta * H$, respectively, are defined as,

$$\begin{aligned} Top\Delta(\alpha) * H &= \{(a, H * H') \mid (a, H') \in Top\Delta(\alpha)\}, \\ Dep\Delta(\alpha) * H &= \{(X, H * H') \mid (X, H') \in Dep\Delta(\alpha)\}. \end{aligned}$$

Lemma 4.4.16

$$\begin{aligned} Top\Delta(\alpha\beta) &= Top\Delta(\alpha) \cup Top\Delta(\beta) * E\Delta(\alpha), \\ Dep\Delta(\alpha\beta) &= Dep\Delta(\alpha) \cup Dep\Delta(\beta) * E\Delta(\alpha). \end{aligned}$$

(proof) It is easy to show that,

$$\begin{aligned} & Top\Delta(\alpha\beta) \\ &= \{(a, H \cup H') \mid (a, H) \in Top\Delta(\alpha), (a, H') \in Top\Delta(\beta) * E\Delta(\alpha)\} \\ &= \{(a, H \cup H'' * E\Delta(\alpha)) \mid (a, H) \in Top\Delta(\alpha), (a, H'') \in Top\Delta(\beta)\} \\ &= Top\Delta(\alpha) \cup Top\Delta(\beta) * E\Delta(\alpha), \end{aligned}$$

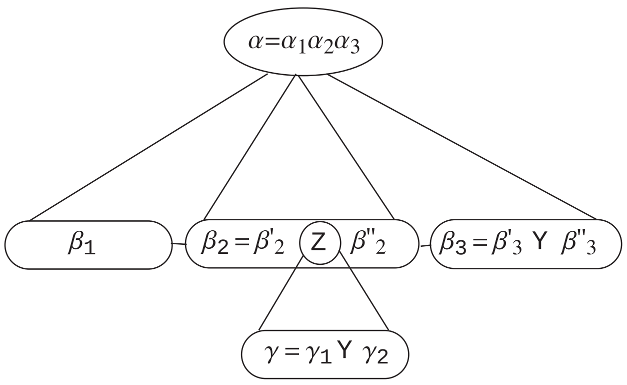


Figure 4.9: Derivation Trees derived from α

and,

$$\begin{aligned}
&Dep\Delta(\alpha\beta)\\
&= \{(X, H \cup H') \mid (X, H) \in Dep\Delta(\alpha), (X, H') \in Dep\Delta(\beta) * E\Delta(\alpha)\}\\
&= \{(X, H \cup H'' * E\Delta(\alpha)) \mid (X, H) \in Dep\Delta(\alpha), (X, H'') \in Dep\Delta(\beta)\}\\
&= Dep\Delta(\alpha) \cup Dep\Delta(\beta) * E\Delta(\alpha).
\end{aligned}$$

//

The purpose of following three arguments is to provide an incremental method of reconstruction of $Top\Delta$ an $Dep\Delta$ on augmenting a new production rule. The leading point of the arguments is how to give a differencial. There are probably many choices of U which satisfies $Top\Delta_P(\alpha) \cup U = Top\Delta_{P'}(\alpha)$, where $\cup$ holds the meaning given in Definition 4.4.11. Following two definitions give a choice from the point of view of easiness on their constructions. Figure 4.9 illustrates derivation trees derived from given symbol sequence α. Just before augmenting new production rule $Z \to \gamma$, where $\gamma = \gamma_1\ Y\ \gamma_2$, one of symbol sequences derived from α is $\beta_1\,\beta_2'\,Z\,\beta_2''\,\beta_3'\,Y\,\beta_3''$. Let W be a set of all syntactic variables contained in $\beta_1\,\beta_2'\,Z\,\beta_2''\,\beta_3'\,Y\,\beta_3''$, then some subset of W may be included in $Dep\Delta_P(\alpha)(Y)$. Suppose a situation that $Z \to \gamma$ is augmented to P. At least two elements would be in $Dep\Delta_{P'}(\alpha)(Y)$, i.e., W_1 which contains all syntactic variables occur in $\beta_1\,\beta_2'\,\gamma_1$, and W_2 which contains all syntactic variables occur in $\beta_1\,\beta_2'\,\gamma\,\beta_2''\,\beta_3'$. These two kind of elements must be reflected in the definition of differencial of $Dep\Delta$.

Definition 4.4.17 (Substitution on $Top\Delta$)
For given $Z \in V' \subset V$ and given $\gamma \in (V \cup T)*$, we define Substitution on Z in $Top\Delta(\alpha)$ to γ, such as,

$$Top\Delta(\alpha)[\gamma/Z] = \{(a, H[E\Delta(\gamma)/Z]) \mid (a, H) \in Top\Delta(\alpha)\}.$$

Definition 4.4.18 (Substitution on $Dep\Delta$)
For given $Z \in V' \subset V$ and given $\gamma \in (V \cup T)*$, we define Substitution on Z in $Dep\Delta(\alpha)$

to γ, such as,

$$Dep\Delta(\alpha)[\gamma/Z] = \{(Y, [H^{\gamma}_{Y} * H^{\alpha}_{Z} \cup H^{\alpha}_{Y}[E\Delta(\gamma)/Z]]) \mid Y \in V,\ H^{\gamma}_{Y} = Dep\Delta(\gamma)(Y),\ H^{\alpha}_{Y} = Dep\Delta(\alpha)(Y),\ H^{\alpha}_{Z} = Dep\Delta(\alpha)(Z)\}$$

Theorem 4.4.19 (Evolution of $Top\Delta$ and $Dep\Delta$)
For each production rule set $P' = P \cup \{Z \to \gamma\}$ and each symbol sequence $\alpha \in (V \cup T)*$,

$$\begin{aligned} Top\Delta_{P'}(\alpha) &= Top\Delta_{P}(\alpha) \cup Top\Delta_{P}(\alpha)[\gamma/Z], \\ Dep\Delta_{P'}(\alpha) &= Dep\Delta_{P}(\alpha) \cup Dep\Delta_{P}(\alpha)[\gamma/Z]. \end{aligned}$$

(proof) It is shown by same discussion on Theorem 4.4.9.//

Example 4.4.20 (Example of Evolution on $Dep\Delta$) *Suppose a situation that a new production rule $A \to \varepsilon$ is augmented, same as discussed in Example 4.4.10. The result of the augmentation is given as below,*

	S	X	Y	A	B	C
$Dep\Delta_{P_6'}(S)$	$\{\phi\}$	$\{\phi\}$	$\{\phi\}$	$\{\phi\}$	$\{\phi\}$	$\{\phi\}$

where $P_6' = P_6 \cup \{A \to \varepsilon\}$. The values other than $Dep\Delta(S)$ are not changed on this augmentation. For example, while $Dep\Delta_{P_6}(S)(Y) = \{\{X\}, \{A\}\}$ (see the table in Example 4.4.10), $Dep\Delta_{P_6'}(S)(Y) = \{\phi\}$, because from Definition 4.4.18,

$$\begin{aligned} Dep\Delta_{P_6}(S)[\varepsilon/A](Y) &= [H^{\varepsilon}_{Y} * H^{S}_{A} \cup H^{S}_{Y}[E\Delta(\varepsilon)/A]], \\ H^{\varepsilon}_{Y} &= Dep\Delta_{P_6}(\varepsilon)(Y) \\ &= \phi, \\ H^{S}_{Y} &= Dep\Delta_{P_6}(S)(Y) \\ &= \{\{X\}, \{A\}\} \\ H^{S}_{Y}[E\Delta(\varepsilon)/A] &= \{\phi\}. \end{aligned}$$

From same reason, $Dep\Delta_{P_6'}(S)(B) = Dep\Delta_{P_6'}(S)(C) = \{\phi\}$.

As described at the top of this section, $Link\Delta$ is introduced so as to present propagation of LA.

Definition 4.4.21 ($Link\Delta$)
$Link\Delta$ is a subset of $\mathbf{N} \times V \times [Power(Power(V'))]$.

Precise value of $Link\Delta$ will appear as the third elements of values of $Ind\Delta$ defined in Definition 4.4.24 and Lemma 4.4.26 below. Here, we illustrate an intuitive explanation of it. Each state q of LALR(1) graph is asigned its own $Link\Delta$ values with each syntactic variables. If $(k, X, H) \in Link\Delta$ for Y of q, where Y is a syntactic variable and k must be a positive integer, then all of LA which flows to X on the state preceding of q by k steps is propageted to the LA for Y on q with additional condition H.

Definition 4.4.22 (Restriction of $Link\Delta$)
For given $Link\Delta \subset \mathbf{N} \times V \times [Power(Power(V'))]$ and given $H \in [Power(Power(V'))]$, restriction of $Link\Delta$ by H is denoted by $Link\Delta * H$, and is defined as,

$$Link\Delta * H = \{(k, X, H' * H) \mid (k, X, H') \in Link\Delta\}.$$

Definition 4.4.23 (Substitution on $Link\Delta$)
For given $Link\Delta \subset \mathbf{N} \times V \times [Power(Power(V'))]$, given $H \in [Power(Power(V'))]$ and given $Z \in V$, Substitution of $Link\Delta$ on Z to H is denoted by $Link\Delta[H/Z]$, and is defined as,

$$Link\Delta[H/Z] = \{(k, X, H'[H/Z]) \mid (k, X, H') \in Link\Delta\}.$$

We modify LA to have enough data for augmentation of production rules. Following definitions and propositions are given in order to define indeces for LA, i.e. $Ind\Delta$, and in order to determine the equivalence of $Ind\Delta$ with LA of LALR(1).

Definition 4.4.24 (LA with ε-Productivity Decision)
For given CFG $G = (V, T, P, S')$ (G is assumed as Extended Grammar), let $SC(lr(G)) = (V \cup T, Q, \delta', q_0', *)$, where $Q = Power(Item \cup \{q_0\})$ and $q_0' = \varepsilon C(\delta, q_0)$, then a function $\lambda\Delta : Q \times Item \to ((T \to [Power(Power(V'))]) \times (V \to [Power(Power(V'))]))$, where $V' \subset V$ is fixed, is defined as,

$$\begin{aligned}
&\lambda\Delta(q_0', S' \to \bullet S) = (Top\Delta(\$), Dep\Delta(\$))\\
&A \to \alpha \bullet X\beta \in q,\ \ \delta'(q, X) = q'\\
&\quad \Rightarrow\ \ \lambda\Delta(q, A \to \alpha \bullet X\beta) \subset \lambda\Delta(q', A \to \alpha X \bullet \beta)\\
&A \to \alpha \bullet B\beta \in q,\ \ \lambda\Delta(q, A \to \alpha \bullet B\beta) = (\Lambda_t, \Lambda_d)\\
&\quad \Rightarrow\ \ \forall B \to \gamma \in P,\\
&\qquad\quad (Top\Delta(\beta) \cup \Lambda_t * E\Delta(\beta), Dep\Delta(\beta) \cup \Lambda_d * E\Delta(\beta)) \subset \lambda\Delta(q, B \to \bullet\gamma)\\
&\forall i \in Item, s.t., i \notin q \Rightarrow \lambda\Delta(q, i) = (Top\Delta(\varepsilon), Dep\Delta(\varepsilon))
\end{aligned}$$

(the value of $\lambda\Delta$ is minimum set that satisfies above conditions), where the inclusion relation $\subset$ is of defined in Definition 4.4.12.

Note: $Top\Delta(\varepsilon)(a) = \phi$ for all $a \in T$, and, $Dep\Delta(\varepsilon)(X) = \phi$ for all $X \in V$.

Lemma 4.4.25 *For each state $q \in Power(Item \cup \{q_0\})$ of $SC(lr(G))$ and each item $i \in Item$,*

$$First(\lambda(q, i)) = \{a \in T \mid \Lambda_t(a) = \mathbf{true}\} \cup \bigcup_{\Lambda_d(X) = \mathbf{true}} First(X),$$

where $(\Lambda_t, \Lambda_d) = \lambda\Delta(q, i)$.

(proof) From the fact that for any $H, H' \in [Power(Power(V'))]$, $H * H'$ is **true** if and only if both of H and H' are **true**, it is easy to see that, when $\lambda\Delta'$ is defined as,

$$\lambda\Delta'(q_0', S' \to \bullet S) = (\{\$\}, \phi)$$

$$\begin{array}{rl} & A \to \alpha \bullet X\beta \in q, \delta'(q,X) = q' \\ \Rightarrow & \lambda\Delta'(q, A \to \alpha \bullet X\beta) \subset \lambda\Delta'(q', A \to \alpha X \bullet \beta) \\ & A \to \alpha \bullet B\beta \in q, \lambda\Delta'(q, A \to \alpha \bullet B\beta) = (Ut, Ud) \\ \Rightarrow & \forall B \to \gamma \in P, \\ & \begin{array}{rcll} U't & = & \{a \in T \mid Top\Delta(\beta)(a) = \mathbf{true}\} \cup Ut & \text{if } E\Delta(\beta) = \mathbf{true} \\ & = & \{a \in T \mid Top\Delta(\beta)(a) = \mathbf{true}\} & \textit{otherwise}, \\ U'd & = & \{Y \in V \mid Dep\Delta(\beta)(Y) = \mathbf{true}\} \cup Ud & \text{if } E\Delta(\beta) = \mathbf{true} \\ & = & \{Y \in V \mid Dep\Delta(\beta)(Y) = \mathbf{true}\} & \text{otherwise}, \end{array} \\ & (U't, U'd) \subset \lambda\Delta'(q, B \to \bullet\gamma) \\ & \forall i \in Item, s.t., i \notin q \Rightarrow \lambda\Delta'(q,i) = (\phi, \phi) \end{array}$$

(the value of $\lambda\Delta'$ is minimum set that satisfies above conditions),

$\lambda\Delta'(q,i) = (\{a \in T \mid \Lambda_t(a) = \mathbf{true}\}, \{Y \in V \mid \Lambda_d(Y) = \mathbf{true}\})$ for $(\Lambda_t, \Lambda_d) = \lambda\Delta(q,i)$. Also, it is easy to see, by induction on length of context, for each state q and item i, $\lambda\Delta'(q,i) = (Ut, Ud)$, $First(\lambda(q,i)) = Ut \cup \bigcup_{X \in Ud} First(X)$, from the similarity of the definition of λ and $\lambda\Delta'$. So, we can conclude the equation

$$First(\lambda(q,i)) = \{a \in T \mid \Lambda_t(a) = \mathbf{true}\} \cup \bigcup_{\Lambda_d(X) = \mathbf{true}} First(X).$$

//

Lemma 4.4.26 *For each state $q \in Power(Item \cup \{q_0\})$ of $SC(lr(G))$ and each εItem $A \to \bullet\alpha$, $A \to \bullet\beta \in \varepsilon Item(A)$,*

$$\lambda\Delta(q, A \to \bullet\alpha) = \lambda\Delta(q, A \to \bullet\beta).$$

(proof) Trivial.//

Definition 4.4.27 (Introduction of $Link\Delta$)
For given CFG $G = (V, T, P, S')$ (G is assumed to Extended Grammar), let $SC(lr(G))$ $= (V \cup T, Q, \delta', q_0', *)$, where $Q = Power(Item \cup \{q_0\})$ and $q_0' = \varepsilon C(\delta, q_0)$, then a function $\Delta : Q \times Item \to ((T \to [Power(Power(V'))]) \times (V \to [Power(Power(V'))]) \times Power(\mathbf{N} \times V \times [Power(Power(V'))])$, where $V' \subset V$ is fixed, is defined as,

$$\Delta(q_0', S' \to \bullet S) = (Top\Delta(\$), Dep\Delta(\$), \phi)$$

$$\begin{array}{l} A \to \alpha \bullet X\beta \in q, \delta'(q,X) = q', \\ \Delta(q, A \to \alpha \bullet X\beta) = (\Lambda_t, \Lambda_d, L) \\ \quad \Rightarrow \quad (Top\Delta(\varepsilon), Dep\Delta(\varepsilon), Incr(L)) \subset \Delta(q', A \to \alpha X \bullet \beta) \\ A \to \alpha \bullet B\beta \in q, \lambda\Delta(q, A \to \alpha \bullet B\beta) = (\Lambda_t, \Lambda_d, L) \\ \quad \Rightarrow \quad \forall B \to \gamma \in P, \\ \qquad\quad (Top\Delta(\beta) \cup \Lambda_t * E\Delta(\beta), Dep\Delta(\beta) \\ \qquad\quad \cup \Lambda_d * E\Delta(\beta), L * E\Delta(\beta)) \subset \Delta(q, B \to \bullet\gamma) \\ \forall i \in Item,\ s.t., i \notin q \Rightarrow \Delta(q,i) = (Top\Delta(\varepsilon), Dep\Delta(\varepsilon), \phi) \end{array}$$

(value of Δ is minimum set that satisfies above condition).

Note : the third argument of Δ concerns to $Link\Delta$.

Lemma 4.4.28 *For each state $q \in Power(Item \cup \{q_0\}) of SC(lr(G))$ and each $\varepsilon Item$ $A \to \bullet\alpha$, $A \to \bullet\beta \in \varepsilon Item(A)$,*

$$\Delta(q, A \to \bullet\alpha) = \Delta(q, A \to \bullet\beta).$$

(proof) Trivial.//

We write $Ind\Delta(q, A) = \Delta(q, A \to \bullet\alpha)$.

Theorem 4.4.29 *Let a function $Exp : ((T \to [Power(Power(V'))]) \times (V \to [Power(Power(V'))]) \times Power(\mathbf{N} \times V \times [Power(Power(V'))])) \to ((T \to [Power(Power(V'))]) \times (V \to [Power(Power(V'))]))$ be,*

$$\begin{aligned} Exp(Ind\Delta(q,A)) &= \{a \in T \mid \Lambda_t(a) = \mathbf{true}\} \\ &\cup \bigcup_{\Lambda_d(X)=\mathbf{true}} First(X) \\ &\cup \bigcup_{(k,X)\in L} Exp(Ind\Delta(q(-k), X)), \end{aligned}$$

where $Ind\Delta(q,A) = (\Lambda_t, \Lambda_d, L)$ and $q(-k)$ is a state ,s.t.,$\delta'(q(-k), w) = q$ for $\mid w \mid = k$, then,

$$Exp(Ind\Delta(q, A)) = First(Ind\lambda(q, A)).$$

(proof) Same discussion as Lemma 4.4.25.//

4.4.3 Space Reduction Method on DD and $Ind\Delta$

As mentioned at the top of this section, we have a method so as to increase efficiency of computation, named *Domain Restriction.* The operations are defined on $E\Delta$, $Top\Delta$, $Dep\Delta$ and $Link\Delta$. Each operations ground on the operation on $E\Delta$. The other restriction method on $Dep\Delta$ can be invented, but ommitted here.

Definition 4.4.30 (Domain Restrictions)
For given $E\Delta : (V \cup T)* \to [Power(Power(V'))]$, $Top\Delta : (V \cup T)* \to (T \to [Power(Power(V'))])$, $Dep\Delta : (V \cup T)* \to (V \to [Power(Power(V'))])$ and $Link\Delta \subset \mathbf{N} \times V \times [Power(Power(V'))]$, each restrictions of Dependency Domain on V' to $V'' \subset V'$ are denoted by $E\Delta \downarrow V''$, $Top\Delta \downarrow V''$, $Dep\Delta \downarrow V''$ and $Link\Delta \downarrow V''$, respectively, and defined as,

$$\begin{aligned} E\Delta(\alpha) \downarrow V'' &= \{\eta \in E\Delta(\alpha) \mid \eta \subset V''\}, \\ Top\Delta(\alpha) \downarrow V'' &= \{(a, H \downarrow V'') \mid (a, H) \in Top\Delta(\alpha)\}, \\ Dep\Delta(\alpha) \downarrow V'' &= \{(X, H \downarrow V'') \mid (X, H) \in Dep\Delta(\alpha)\}, \\ Link\Delta \downarrow V'' &= \{(k, X, H \downarrow V'') \mid (k, X, H) \in Link\Delta\}, \end{aligned}$$

respectively.

4.4.4 A Supplement of the Section

As a final argument of this section, a notion named $Inherit\Delta$ is introduced. When fusing two states during incremental construction of LALR(1) graph, according to the direction of ε-transition caused by Bridge Transition ξ, some LA may be inherited from one state to the other. To express the inheritance, we use a function $Inherit\Delta : Q \to Power(V)$, where Q is the set of states of LALR(1) graph.

Definition 4.4.31 ($Inherit\Delta$)

$$Inherit\Delta(q) = \{X \in V \mid q = \varepsilon C(\varepsilon Item(X))\}.$$

Note: If $q \neq \varepsilon C(\varepsilon Item(X))$, say $Ent\varepsilon(X)$, for any $X \in V$, then $Inherit\Delta(q) = \phi$. On almost all cases, $Inherit\Delta(Ent\varepsilon(X))$ contains unique element X. However, given grammar has production rules which have mutual leftmost recursion, then $\#(Inherit\Delta(Ent\varepsilon(X)))$ might be greater than 1.

4.5 Algorithms

Here, we present five algorithms. Algorithm 4.5.1 shows a procedure to construct $MonoG$, which depends on the statements of Lemma 4.3.4, Definition 4.3.5, Lemma 4.3.14 and 4.3.15 for constructing LR(0) graph for $MonoG$, and Definition 4.4.27 and its reliances in previous section. Algorithm 4.5.2 shows a procedure to calculate $Ind\Delta$ for a fused state. Algorithm 4.5.3 shows a procedure of fusion operation given in Definition 4.2.13. Algorithm 4.5.4 and 4.5.5 are procedures for augmenting a new production rule to current LALR(1) graph, and for incremental construction of LALR(1) graph, respectively. The process of incremental construction is illustrated that, firstly constructing a $MonoG(S' \to S)$ as a core graph (by Algorithm 4.5.1), then constructing a $MonoG$ for next production rule (by Algorithm 4.5.1) and fusing the $MonoG$ and current graph (by Algorithm 4.5.3 and 4.5.2), and the iteration of it (by Algorithm 4.5.5). As mentioned so far, we treat LALR(1) graph as a kind of multi-entrance graph. So, fusion process must be influenced to each entrances and their succeeding states of the graph. Algorithm 4.5.4 gives a procedure for it.

Precise procedures for the operations on $E\Delta$, $Top\Delta$, $Dep\Delta$, $Link\Delta$ and $MaxInc$ are not presented here. Because it seems to be clear that the procedures are implemented straightforwardly from the definitions of them, and in fact, we have implemented the procedures in such a way. Some discussions on the procedures and the efficiencies on them are stated in the next section.

Whole algorithms are described in Pascal like style. Operations on sets, i.e. of set theoretical meaning, are immediately used. On such a cases that the order to take elements out of a set essentially influences results, we insert comments for them. Additionally, we adopt '=' for substitution, and, combinations '=' with other operators like C language, e.g., $A \cup = B$ means $A = A \cup B$. For accessing to an element of $Ind\Delta(q, X) = (IndT\Delta, IndV\Delta, Link\Delta)$, we write $Ind\Delta(q, X).IndT\Delta$, so on.

Finally, we must note that some outputs of algorithms are assumed as side-effect at the time when to call them. For example, only first output of $MonoG$ is explicitly used in the description of Algorithm 4.5.4, while there are seven outputs from $MonoG$.

Algorithm 4.5.1 (Construction Algorithm of $MonoG$**)**
Suppose the situation that CFG $G' = (V, T, P, S')$ is given and for each $X \in V$, $E\Delta_{G'}(X)$ and $Dep\Delta_{G'}(X)$ are already calculated. Here, we show an algorithm for construction of $MonoG(Z \to X_1 \cdots X_n)$ and calculating its $Ind\Delta$. Here, we suppose $Z \in V$, $X_1, \ldots, X_n \in V \cup T$. In the algorithm below, we write value of $Ind\Delta(q,i) = (Ind\Delta(q,i).IndT\Delta$, $Ind\Delta(q,i).IndV\Delta$, $Ind\Delta(q,i).Link\Delta)$.

Inputs:

- *a Production Rule $Z \to X_1 \cdots X_n$*
- *for each $X \in V$, values of $E\Delta_P(X)$ and $Dep\Delta_P(X)$*

Outputs:

- *$MonoG(Z \to X_1 \cdots X_n) = (V \cup T, Q_2, \zeta_2, q_{2,0}, *)$ (where Q_2 is represented by $\{p_1, \ldots, p_{n+1}\}$, by Lemma 4.3.7)*
- *for each $X \in V$, values of $E\Delta_{P'}(X)$ and $Dep\Delta_{P'}(X)$ (where $P' = P \cup \{Z \to X_1 \cdots X_n\}$)*
- *for each $p_i \in Q_2$, $X \in V$, values of $Ind\Delta(p_i, X)$*
- *Reduce Item Table, $Red : Q_2 \to Power(V \times \mathbf{N})$.*
- *$Inherit\Delta(p_i)$ for each $p_i \in Q_2$.*
- $\subset_{MonoG(Z \to X_1 \cdots X_n)}$
- *$MaxInc(q)$ for each state q of $MonoG(Z \to X_1 \cdots X_n)$*

Internal Data: • *$Flow \subset Q_2 \times V \times \{1, 2, \ldots, n\}$*

(* **Initialization** *)

```
- Find $Z$ in $X_1, \ldots, X_n$, then make occurrence position list $J = \{j_1, \ldots, j_m\}$,
    where $X_{j_i} = Z$ for each $1 \leq i \leq m$.
(* $J$ may be dynamically augmented by some values in $\{1, \ldots, n\}$ in calculation
    of $\zeta_2$ and $Flow$ stage *)
- Initialize all values of $Flow$ to $\phi$.
- Initialize $\subset_{MonoG(Z \to X_1 \cdots X_n)}$ to $\phi$.
- for all $q \in \{q_1, \ldots, q_{n+1}\}$, $MaxInc(q) = \phi$.
```

(* **Evolution of $E\Delta_P(X)$ and $Dep\Delta_P(X)$** *)

```
foreach $X \in V$ do
    $E\Delta_{P'}(X) = E\Delta_P(X) \cup E\Delta_P(X)[E\Delta_P(X_1 \cdots X_n)/Z]$;
    (* by Theorem 4.4.9 *)
    $Dep\Delta_{P'}(X) = Dep\Delta_P(X) \cup Dep\Delta_P(X)[X_1 \cdots X_n/Z]$;
    (* by Theorem 4.4.19 *)
end ;
```

(* **Calculation of ζ_2, $Flow$ and $\subset_{MonoG(Z \to X_1 \cdots X_n)}$** *)

```
for $i = 1$ to $n$ do
    $\zeta_2(p_i, X_i) = p_{i+1}$ ;
    if $X_i \in V$ then
        $\mathit{Flow}(p_i, X_i) \cup = \{i\}$
    end
end ;

$J = \{j_1, \ldots, j_m\}$;
$J' = \phi$;
while $J \neq \phi$ do
    $k \in J$ ;
    $J = J \setminus \{k\}$ ;
    $J' \cup = \{k\}$;

    for $i = 1$ to $n$ do
        if $i \neq k$ then
        - Add $(p_i, p_k)$ to $\subset_{MonoG(Z \to X_1 \cdots X_n)}$.
        end;

        if $X_i \in V$ then
            $\mathit{Flow}(p_k, X_i) \cup = \{i\}$
        end;

        if $X_i = Z$ and $k \notin J'$ then
            $J \cup = \{k\}$
        end ;

        if $\zeta_2(p_k, X_i) =$ **Undefined** then
            $\zeta_2(p_k.X_i) = p_{i+1}$ ;
            $k = p_{i+1}$
        else
            $k = \zeta_2(p_k, X_i)$
        end
    end
end;
```

(* Registration of Reduce Item *)

```
$\mathit{Red}(p_{n+1}) = \{(Z, n)\}$;
for $i = 1$ to $n$ do
    $\mathit{Red}(p_i) = \phi$ ;
end ;
```

(* Compute values of $Ind\Delta$ *)

- for each state, initialize values of $Ind\Delta to(Top\Delta(\varepsilon), Dep\Delta(\varepsilon), \phi)$.
- for each $i = 1, \ldots, n$, compute values of $Dep\Delta_{P'}(X_i \cdots X_n)$, $Top\Delta_{P'}(X_i \cdots X_n)$, and $E\Delta_{P'}(X_i \cdots X_n)$.

```
        (* for Fusion Process *)
        $Ind\Delta(p_1, Z).IndT\Delta = \{(\$, \mathbf{true})\}$;
            (* For Inheritance of Look Ahead Symbols when to fuse two states. *)

        foreach $(p, X, k) \in Flow$ do
            (* if $(p, X, k), (p, X, k') \in Flow$ and $k > k'$, then we assume the process for
                $(p, X, k)$ will be done first.
            *)
            if $X = Z$ and $X_1 = Z$ then
                (* the case of Left Recursion *)
                $Ind\Delta(p, X).IndV\Delta \cup = Dep\Delta_{P'}(X_{k+1} \cdots X_n) * E\Delta_{P'}(X_2 \cdots X_n)$ ;
                $Ind\Delta(p, X).IndT\Delta \cup = Top\Delta_{P'}(X_{k+1} \cdots X_n) * E\Delta_{P'}(X_2 \cdots X_n)$ ;
                if $k > 1$ then
                    $Ind\Delta(p, X).Link\Delta \cup = \{(k-1, Z, E\Delta_{P'}(X_{k+1} \cdots X_n) * E\Delta_{P'}(X_2 \cdots X_n))\}$
                end
            end ;

            $Ind\Delta(p, X).IndV\Delta \cup = Dep\Delta_{P'}(X_{k+1} \cdots X_n)$ ;
            $Ind\Delta(p, X).IndT\Delta \cup = Top\Delta_{P'}(X_{k+1} \cdots X_n)$ ;
            if $k > 1$ then
                $Ind\Delta(p, X).Link\Delta \cup = \{(k-1, Z, E\Delta_{P'}(X_{k+1} \cdots X_n))\}$
            else
                $Ind\Delta(p, X).IndV\Delta \cup = Ind\Delta(p, Z).IndV\Delta * E\Delta_{P'}(X_2 \cdots X_n)$ ;
                $Ind\Delta(p, X).IndT\Delta \cup = Ind\Delta(p, Z).IndT\Delta * E\Delta_{P'}(X_2 \cdots X_n)$ ;
            end

        end;

    (* Registration of $Inherit\Delta$ *)

        $Inherit\Delta(p_1) = \{Z\}$;
        for $i = 2$ to $n + 1$ do
            $Inherit\Delta(p_i) = \phi$
        end;

(* End of Algorithm *)
```

Algorithm 4.5.2 (Fusion of two States $FuseState(q_1, q_2)$)
We give here an algorithm to calculate new state q' obtained by Fusion of given two states q_1, q_2, where $q' = q_1 \cup q_2$. Let q_1 be Subjective State and q_2 be Dependent State. The relation between Subjective State and Dependent State is decided by the direction of ε-transition which is caused by Bridge Transition ξ defined in Fusion Operation on two DFAs. The source state of ε-transition by ξ is called Subjective State and the Destination State Dependent State. Some inheritance of LA may happen from Subjective State to Dependent State. To express the inheritance, we use values of $Inherit\Delta$ and values concerning to special symbol '\$' in $IndT\Delta$ of $Ind\Delta$. In conventional LR parsers, '\$' is used for "End of Text". This interpretation of '\$' remains in this paper as a special case for initial state.

Inputs:

- *Subjective State q_1, its values of $Ind\Delta(q_1, X)$ for each $X \in V$, and value of $Inherit\Delta(q_1)$.*
- *Dependent State q_2, its values of $Ind\Delta(q_2, X)$ for each $X \in V$, and value of $Inherit\Delta(q_2)$.*

Outputs:

- *Fused state $q' = q_1 \cup q_2$, its values of $Ind\Delta(q', X)$ for each $X \in V$, and value of $Inherit\Delta(q')$.*

Internal Data:

- *Δ as temporal data which has data type of $Ind\Delta$.*

```
(* Initialization *)

    - get a new state $q'$ as $q_1 \cup q_2$.

    (* copy $Ind\Delta$ for $q_2$ *)
    foreach $X \in V$ do
        $\Delta(X) = Ind\Delta(q_2, X)$
    end ;

  (* On the case of Inheritance *)

    foreach $Z \in Inherit\Delta(q_2)$ do
        foreach $Y \in_V$ do
            if $(\$, H) \in \Delta(Y).IndT\Delta$ then
                $\Delta(Y).IndT\Delta = \Delta(Y).IndT\Delta \setminus \{(\$, H)\}$;
                $\Delta(Y).IndT\Delta\cup = Ind\Delta(q_1, Z).IndT\Delta * H$;
                $\Delta(Y).IndV\Delta\cup = Ind\Delta(q_1, Z).IndV\Delta * H$;
                $\Delta(Y).Link\Delta\cup = Ind\Delta(q_1, Z).Link\Delta * H$
            end
        end
    end ;

  (* Main Process *)

    foreach $Y \in V$ do
        $Ind\Delta(q', Y).IndT\Delta = Ind\Delta(q_1, Y).IndT\Delta \cup \Delta(Y).IndT\Delta$;
        $Ind\Delta(q', Y).IndV\Delta = Ind\Delta(q_1, Y).IndV\Delta \cup \Delta(Y).IndV\Delta$;
        $Ind\Delta(q', Y).Link\Delta = Ind\Delta(q_1, Y).Link\Delta \cup \Delta(Y).Link\Delta$
    end;

  (* Inheritance of Inheritance *)
```

```
    $Inherit\Delta(q') = Inherit\Delta(q_1)$;
(* Register Reduce Items *)
    $Red(q') = Red(q_1) \cup Red(q_2)$;
(* End of Algorithm *)
```

Algorithm 4.5.3 (Fusion of Two Sub-Graphs $FuseGraph(q_1, q_2)$)
On process of this algorithm, there are leading problem, one is how to take information for Bridge Transition, and the other is how to distinguish or identify a state beging fused from existing states. As discussed in Sections 4.3, the former problem is resolved by use of information on transition, and the latter by use of MaxInc. The description of this algorithm is complicated, which is caused by division of cases. Important cases are on fusing initial states of given graphs, and on grammars contain production rules which form mutual left recursion. On the latter case, some entrances concerning to mutually different syntactic variables are identical. Here, we give an explanation of internal data used in the algorithm. Stack data are assumed to be elements of $Q_1 \times Power(Q_1) \times Power(Q_2) \times Power(V \cup T)$. Consider the definition of fuse operation. There are two directions on Bridge Transition, one is from subjective to dependent, the other from dependent to subjective. The first argument of stack data is concerned to a state of subjective as a source of Bridge Transition, the second a set of states of dependent as destinations, and, the third is a set of states of subjective as destinations. The forth argument of stack data designates a set of symbols on what transition must be focused at some points of remaining process.

Inputs:

- *Initial State q_1 of Subjective Sub-Graph and value of $Inherit\Delta(q_1)$.*
- *Initial State q_2 of Dependent Sub-Graph and value of $Inherit\Delta(q_2)$.*
  (we assume Dependent Sub-Graph is a Mono-Graph)
  (where we assume that $q_1 = \varepsilon C(\delta_1, \varepsilon Item(X_1)) = Ent\varepsilon(X_1)$ and $q_2 = \varepsilon C(\delta_2, \varepsilon Item(X_2))$, for $X_1, X_2 \in V$. δ_1 and δ_2 are of Definition 4.2.7)
- *Table of $Ent\varepsilon(X) = \varepsilon C(\zeta_1, X)$ for each $X \in V$*
- *Inclusion Information ($\subset$ and MaxInc for each graph)*

Outputs:

- *Fused Sub-Graph starts with Initial State q_0 and value of $Inherit\Delta(q_0)$.*
- *Values of $Ind\Delta(q, X)$ for each $X \in V$ and newly added state q.*
- *Inclusion Information ($\subset$ and MaxInc for fused graph, but it is partial information concerning to input graphs; subset of $\subset_{(Q_1,Q_2)}$ and $MaxInc_{(Q_1 \cup Q_2)}$ of Definition 4.3.16)*

Internal Data:

- *$Stack \in Q_1 \times Power(Q_1) \times Power(Q_2) \times Power(V \cup T)$, where Q_1 and Q_2 are Set of States of Graphs start with q_1 and q_2, respectively.*

(ζ_1 and ζ_2 denote Transition Function of Graphs start with q_1 and q_2 respectively *)*

```
(* Initialize *)

    $Stack = Empty$;

    (* Calculation of Initial State *)
    if $\zeta_1(q_1, X_2) \neq$ **Error** or $q_1 = Ent\varepsilon(X_2)$ then
                (* where **Error** is the abbreviation of $\phi$ *)
                (* this check is just $\xi$. *)
      (* make new Initial State for $Ent\varepsilon(X_1) = \varepsilon C(\varepsilon Item(X_1))$ *)

      if $\zeta_1(q_1, X_2) \neq$ **Error** and $\zeta_2(q_2, X_1) \neq$ **Error** then
         (* the case of mutual left-recursion. $Ent\varepsilon(X_1)$ and $Ent\varepsilon(X_2)$ will be
       identical. *)
         $inherit\Delta(q_2) \cup = inherit\Delta(q_1)$; (* some what tricky! *)
         $inherit\Delta(q_1) = inherit\Delta(q_2)$
      end;
      $q' = FuseState(q_1, q_2)$;

      if $\exists ! Y \in V$ s.t. $Y \neq X_2$ and $\zeta_2(q_2, Y) \neq$ **Error** and $\zeta_1(q_1, Y) =$ **Error** then
         $q' = FuseState(q', Ent\varepsilon(Y))$ ; (* because dependent is Mono-Graph *)

         - Register $q'$ as $Ent\varepsilon(X_1)$. (* However, state $q_1$ still remained. *)
         $MaxInc(q') = MaxInc(q_1 \cup q_2 \cup Ent\varepsilon(Y))$;
                (* will be $\phi$ in process after FuseGraph *)
         - Register $\forall p \subset q_1$, $p \neq \phi$, $p \subset q'$.
         - Register $\forall p \subset q_2$, $p \neq \phi$, $p \subset q'$.
         - Register $\forall p \subset Ent\varepsilon(Y)$, $p \neq \phi$, $p \subset q'$.

         foreach$X \in V \cup T$ do (* copy Transition *)
            $\zeta_1(q', X) = \zeta_1(q_1, X)$
         end;

         $q_1 = q'$;

         $Push(Stack, (q_1, \{q_2\}, \{Ent\varepsilon(Y)\}, V \cup T))$

      else

         - Register $q'$ as $\varepsilon C(\varepsilon Item(X_1))$. (* However, state $q_1$ still remained. *)
         $MaxInc(q') = MaxInc(q_1 \cup q_2)$;
                (* will be $\phi$ in process after FuseGraph *)
         - Register $\forall p \subset q_1$, $p \neq \phi$, $p \subset q'$.
         - Register $\forall p \subset q_2$, $p \neq \phi$, $p \subset q'$.
```

```
            foreach $X \in V \cup T$ do (* copy Transition *)
                $\zeta_1(q', X) = \zeta_1(q_1, X)$
            end;

            $q_1 = q'$;

            $Push(Stack, (q_1, \{q_2\}, \phi, V \cup T)$

        end
    else
        $Push(Stack, (q_1, \phi, \phi, V \cup T))$
    end;

(* Main Loop *)

    while not $Empty(Stack)$ do
        $(q, \sigma_1, \sigma_2, U) = Pop(Stack)$;
        $a \in U$;
        $U = U \setminus \{a\}$;
        if $U \neq \phi$ then
            $Push(Stack, (q, \sigma_1, \sigma_2, U))$
        end;

        $q' = \zeta_1(q, a)$;
        if $\zeta_1(q', X_2) \neq$ **Error** then
            $\sigma_1' = \{q_2\}$
        else
            $\sigma_1' = \phi$
        end;

        $\sigma_2' = \phi$;
        foreach $q'' \in \sigma_1$ do
            if $\zeta_2(q'', a) \neq$ **Error** then
                $\sigma_1' \cup = \{\zeta_2(q'', a)\}$ ;
                foreach $Y \in V$ do
                    if $\zeta_1(q', Y) =$ **Error** and $\zeta_2(\zeta_2(q'', a), Y) \neq$ **Error** then
                        $\sigma_2' \cup = \{Ent\varepsilon(Y)\}$
                    end
                end
            end
        end

        foreach $q''' \in \sigma_2$ do
            if $\zeta_1(q''', a) \neq$ **Error** then
                $\sigma_2' \cup = \{\zeta_1(q''', a)\}$
            end
        end
        foreach $q''' \in \sigma'_2$ do
            if $\zeta_1(q''', X_2) \neq \textbf{Error}$ then
                $\sigma'_1 \cup = \{q_2\}$ ;
                if $\exists! Y \in V$ s.t. $Y \neq X_2$ and $\zeta_2(q_2, Y) \neq \textbf{Error}$ then
                    $\sigma'_2 \cup = \{Ent\varepsilon(Y)\}$
                end
            end
         end;

    (* check Error case *)
        if $q' = \textbf{Error}$ and $\sigma'_1 = \phi$ and $\sigma'_2 = \phi$ then
            $\zeta_1(q, a) = \textbf{Error}$;
            continue
        end;

    (* check existing state *)
        if $\exists! p$ s.t. $MaxInc(p) = MaxInc(q' \cup (\bigcup \sigma'_1) \cup (\bigcup \sigma'_2))$ then
            $\zeta_1(q, a) = p$;
            continue
        end;

    (* Fuse States *)
        $q_3 = q'$;
        foreach $q'' \in \sigma'_1$ do
            $q_3 = FuseState(q_3, q'')$
        end;
        foreach $q''' \in \sigma'_2$ do
            $q_3 = FuseState(q_3, q''')$
        end;

    (* copy Transition *)
        foreach $X \in V \cup T$ do
            $\zeta_1(q_3, X) = \zeta_1(q', X)$
        end;

    (* renewal Transition, may be *)
        $\zeta_1(q, a) = q_3$;

    (* register inclusion relation *)
    $MaxInc(q_3) = MaxInc(q' \cup (\bigcup \sigma'_1) \cup (\bigcup \sigma'_2))$;
        - Register $\forall p \subset q'$, $p \subset q_3$
        foreach $q'' \in \sigma'_1$ do
            - Register $\forall p \subset q''$, $p \subset q_3$
        end;
        foreach $q''' \in \sigma'_2$ do
            - Register $\forall p \subset q'''$, $p \subset q_3$
        end

        $Push(Stack, (q_3, \sigma'_1, \sigma'_2, V \cup T))$;
```

```
        end;
(* Post Operations *)
        - Maximalize $MaxInc(q)$ for each state $q$.
            (* there is possibility for $MaxInc$ to have non-maximal elements *)
(* End of Algorithm *)
```

Algorithm 4.5.4 (Augmentation of a new Production Rule, *AugRule***)**

Assumptions: *$SC(lr(G))$ contains two kinds of states*

1. *Actually used states for parsing starts with initial state*
2. *$Ent\varepsilon(X) = \varepsilon C(\varepsilon Item(X))$ for each $X \in V$, and States which are able to be arrived from them.*
   Each state does not have information about Items which belong to the state. We only have inclusion relation between states.
   In the description,

   - $SC(lr(G)) = (V \cup T, Q_1, \zeta_1, q_1, *)$,
   - $MonoG(Z \to X_1 \cdots X_n) = (\Sigma, Q_2, \zeta_2, q_2, *)$

   are assumed.

Inputs:

- *a new Production Rule $Z \to X_1 \cdots X_n$ that will be augmented to $SC(lr(G))$.*

Outputs:

- *$SC(lr(G'))$, where G' is the new Grammar which is obtained so as to augment $Z \to X_1 \cdots X_n$ to G.*

```
(* Construct Mono-Graph for given new Production Rule *)
        $G_2 = MonoG(Z \to X_1 \cdots X_n)$; (* by Algorithm 4.5.1 *)
            (* Initial State of $G_2$ is $\varepsilon C(\varepsilon Item(Z))$ in the mean of $MonoG(Z \to X_1 \cdots X_n)$ *)
    (* Renewal existing $Ind\Delta$ to fit with augmented grammar *)
```

```
foreach $q \in Q_1$ do
    foreach $X \in V$ do
        (* $Dep\Delta(X_1 \cdots X_n)$ and $E\Delta(X_1 \cdots X_n)$ are already computed above.
    *)

        $Ind\Delta(q, X).IndT\Delta\cup = Ind\Delta(q, X).IndT\Delta[X_1 \cdots X_n/Z]$;
        $Ind\Delta(q, X).IndV\Delta\cup = Ind\Delta(q, X).IndV\Delta[X_1 \cdots X_n/Z]$;
        $Ind\Delta(q, X).Link\Delta\cup = Ind\Delta(q, X).Link\Delta[E\Delta(X_1 \cdots X_n)/Z]$;
    end
end;

(* Fuse Graphs *)
$Ent\varepsilon(S') = FuseGraph(Ent\varepsilon(S'), G_2)$;
$Ent\varepsilon(Z) = FuseGraph(Ent\varepsilon(Z), G_2)$;
        (* but this call enforces to fuse each initial states *)
foreach $X \in V \setminus \{S', Z\}$ do
    $Ent\varepsilon(X) = FuseGraph(Ent\varepsilon(X), G_2)$;
end;
```

```
(* Post Process *)

    - Calculate $\subset_{SC(lr(G'))}$ and $MaxInc_{SC(lr(G'))}$. (* by Definition 4.3.17 *)
    - Remove unreachable states from any $Ent\varepsilon(X)$.

(* End of Algorithm *)
```

Algorithm 4.5.5 (Incremental Construction of LALR(1) graph)
This procedure contains initialization and simply iterate adaptation of AugRule (Algorithm 4.5.4).

Inputs: • *Extended CFG* $G = (V, T, P, S')$

Outputs: • $SC(lr(G))$

```
(* Initialization *)

    foreach $X \in V$ do
        $E\Delta(X) = \{\{X\}\}$;

        foreach $a \in T$ do
            $Top\Delta(X)(a) =$ false
        end;

        foreach $Y \in V \setminus \{X\}$ do
            $Dep\Delta(X)(Y) =$ false
        end;
        $Dep\Delta(X)(X) =$ true ;
    end;
```

```
(* Main *)

  $Ent\varepsilon(S') = MonoG(S' \to S)$;
  foreach $X \to \alpha \in P \setminus \{S' \to S\}$ do
    $AugRule(X \to \alpha)$
  end;

(* End of Algorithm *)
```

As a final remark of this section, here we discuss on calculation of LA from $Ind\Delta$. To calculate $Exp(Ind\Delta(q, A))$, defined in Theorem 4.4.29, there needs to prepare an exclusive procedures for it outside of algorithms above. However, among terms of which the definition of Exp is constructed, the first term $\{a \in T \mid \Lambda_t(a) = \mathbf{true}\}$ is calculated straightforwardly from $Ind\Delta$, and for the second term $\bigcup_{\Lambda_d(X)=\mathbf{true}} First(X)$, it can be calculated by also in incremental manner. To illustrate this fact, we adopt a notation $First\Delta_P(X)$, i.e. production rule set P is emphasized, and defined values with DD,

$$First\Delta_P(X) = \bigcup_{X \to \gamma \in P} Top\Delta_P(\gamma).$$

Using it, $Exp(Ind\Delta(q, A))$ is written as,

$$\begin{aligned} Exp(Ind\Delta(q, A)) \;=\; & \{a \in T \mid \Lambda_t(a) = \mathbf{true}\} \\ & \cup \{a \in T \mid \Lambda_d(X) = \mathbf{true}, First\Delta_P(X)(a) = \mathbf{true}\} \\ & \cup \bigcup_{(k,X) \in L} Exp(Ind\Delta(q(-k), X)). \end{aligned}$$

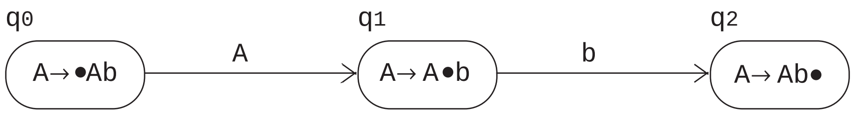
When new production rule $X \to \gamma'$ is augmented to the grammar, $First\Delta_P(X)$ is renewed such as,

$$First\Delta_{P'}(X) = First\Delta_P(X) \cup First\Delta_P(X)[\gamma/X] \cup Top\Delta_{P'}(\gamma'),$$

where in the equation, operation $\cup$ is on $Top\Delta$, defined in Definition 4.4.12. The equation is easily established from Theorem 4.4.19 and Definition 4.4.12. To embed the equation of $First_{P'}$ in Algorithm 4.5.4, most of work for calculation of Exp is completing. Thus, remaining process in order to calculate $Exp(Ind\Delta(q, A))$ is a closure process so as to trace links given by $Link\Delta$.

4.6 Examples

Example 4.6.1 (Mono-Graph(1))
Figure 4.10 illustrates a Mono-Graph of production rule $A \to Ab$. As shown in Lemma 4.3.4, there are three states consisted of. Induce information for LA, $IndT\Delta$, $IndV\Delta$ and $Link\Delta$, and state identifier MaxInc are presented beside each states. Values of MaxInc of each states are ϕ, as shown in Lemma 4.3.15. Terminal symbol '\$' has the meaning of "end of text" same as on conventional LALR(1) parser, and moreover, as described in Algorithm 4.5.2, it has a new interpretation in that it denotes 'inheritance' of LA, when fused. So, because $q_0 = Ent\varepsilon(A)$, $IndT\Delta(A)(\$)$ is **true** *unconditionally. $IndT\Delta(A)(b)$*

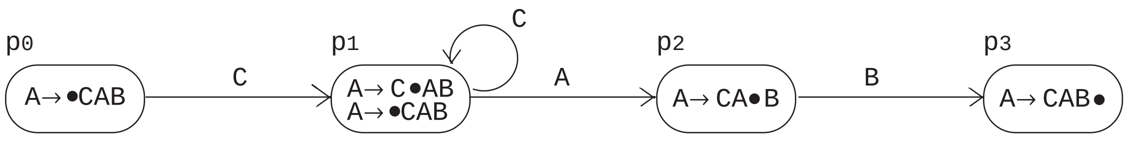


MaxInc	ϕ	
Inclusion	ϕ	
IndTΔ	(A)($)	**true**
	(A)(b)	**true**
IndVΔ	ϕ	
LinkΔ	ϕ	

MaxInc	ϕ
Inclusion	ϕ
IndTΔ	ϕ
IndVΔ	ϕ
LinkΔ	ϕ

MaxInc	ϕ
Inclusion	ϕ
IndTΔ	ϕ
IndVΔ	ϕ
LinkΔ	ϕ

Figure 4.10: Mono Graph (1)



MaxInc	ϕ	
Inclusion	ϕ	
IndTΔ	(A)($)	**true**
	(C)($)	{{A, B}}
IndVΔ	(C)(A)	**true**
	(C)(B)	{{A}}
LinkΔ	ϕ	

MaxInc	ϕ	
Inclusion	{p0}	
IndTΔ	ϕ	
IndVΔ	(A)(B)	**true**
	(C)(A)	**true**
	(C)(B)	{{A}}
LinkΔ	(A)	{(1, A,{{B}})}
	(C)	{(1, A,{{A, B}})}

MaxInc	ϕ	
Inclusion	ϕ	
IndTΔ	ϕ	
IndVΔ	ϕ	
LinkΔ	(B)	{(2, A,**true**)}

MaxInc	ϕ
Inclusion	ϕ
IndTΔ	ϕ
IndVΔ	ϕ
LinkΔ	ϕ

Figure 4.11: Mono Graph (2)

is also **true**, *because an item $A \to \bullet Ab$ belongs to q_0, dot is followed by syntactic variable A, and A is followed by b directly. In q_1 and q_2, dot is followed by a syntactic variable or no symbol. So, whole values of $IndT\Delta$ and $IndV\Delta$ are* **false**. *We should emphasize repeatedly that in our approach, any item set is not used during whole computation. Items described in Figure 4.10 are only for facilitation to understand the graph.*

Example 4.6.2 (Mono-Graph(2))
Figure 4.11 illustrates $MonoG(A \to CAB)$ which grammar is identical to Example 4.3.6. By same reason as described in Example 4.6.1, in state p_0, $IndT\Delta(A)(\$) =$ **true**. *Excepting this vacuous case, we concentrate on syntactic variable C during calculating values of $IndT\Delta$ and $IndV\Delta$, because only syntactic variable C occurs just right of dot in the item consisted of by p_0. A terminal '\$' is influenced to syntactic variable C through AB which is situated just right of C, so, $IndT\Delta(C)(\$) = \{\{A, B\}\}$, $IndV\Delta(C)(A) =$* **true** *and $IndV\Delta(C)(B) = \{\{A\}\}$. In state p_1, $Link\Delta(A)$ has value $\{(1, A, \{\{B\}\})\}$. It means that LA for all $\varepsilon Item(A)$ induced in the state inherit LA of $\varepsilon Item(A)$ in previous states by one step, with a condition if some new production rules are added so as that ε is derived*

from B. We can understand that the value of LinkΔ in p_1 is valid, from the fact that p_1 includes item $A \to C \bullet AB$, dot is followed by A and A is followed by B. In order for item $A \to \bullet CAB$ to inherit LA of $A \to C \bullet AB$, ε must be derived from B. By similar discussion, Link$\Delta(C)$ in p_1 has value $\{(2, A, \{\{A, B\}\})\}$.

*To calculate all of these value, in Algorithm 4.5.1, Flow, which is defined at Definition 4.3.5, is calculated first. Its process is just related to the calculation of Subset Construction. As the result, Flow $= \{(p_0, C, 1), (p_1, A, 2), (p_1, C, 1), (p_2, B, 3)\}$. Induce information for LA is calculated by use of Flow, and initial values of Dep$\Delta(X)$ for each $X \in V = \{A, B, C\}$, i.e., Dep$\Delta(X)(X) =$ **true**, Dep$\Delta(X)(Y) =$ **false** for $X \neq Y$, and $E\Delta(X) = \{\{X\}\}$ for each $X \in V$. On second step of Algorithm 4.5.1, i.e. "Evolution of $E\Delta_P(X)$ and Dep$\Delta_P(X)$" following initialization, only $E\Delta(A)$ and Dep$\Delta(A)$ are evolved, because no $E\Delta(X)$ and Dep$\Delta(X)$ contain A if $X \neq A$, at the time. For a moment, we trace the calculation, following the algorithm faithfully.*

$$\begin{aligned}
E\Delta_{P'}(A) &= E\Delta_P(A) \cup E\Delta_P(A)[E\Delta_P(CAB)/A] \\
&= \{\{A\}\} \cup \{\{A\}\}[\{\{A, B, C\}\}/A] \\
&\quad \textit{(by Definition 4.4.4 and Proposition 4.4.6)} \\
&= \{\{A\}\} \cup \{\{A, B, C\}\} \\
&\quad \textit{(by Definition 4.4.8)} \\
&= \{\{A\}\} \\
&\quad \textit{(by Definition 4.4.1)} \\
\mathit{Dep}\Delta_P(CAB) &= \mathit{Dep}\Delta_P(C) \cup \mathit{Dep}\Delta_P(AB) * E\Delta_P(C) \\
&\quad \textit{(by Lemma 4.4.16)} \\
&= \{(C, \{\phi\})\} \cup (\mathit{Dep}\Delta_P(A) \cup \mathit{Dep}\Delta_P(B) * E\Delta_P(A)) * \{\{C\}\} \\
&= \{(C, \{\phi\})\} \cup (\{(A, \{\phi\})\} \cup \{(B, \{\phi\})\} * \{\{A\}\}) * \{\{C\}\} \\
&= \{(C, \{\phi\})\} \cup (\{(A, \{\phi\})\} \cup \{(B, \{\{A\}\})\}) * \{\{C\}\} \\
&\quad \textit{(by Definition 4.4.15)} \\
&= \{(C, \{\phi\})\} \cup \{(A, \{\phi\}), (B, \{\{A\}\})\} * \{\{C\}\} \\
&\quad \textit{(by Definition 4.4.11)} \\
&= \{(C, \{\phi\})\} \cup \{(A, \{\{C\}\}), (B, \{\{A, C\}\})\} \\
&= \{(A, \{\{C\}\}), (B, \{\{A, C\}\}), (C, \{\phi\})\} \\
\mathit{Dep}\Delta_{P'}(A) &= \mathit{Dep}\Delta_P(A) \cup \mathit{Dep}\Delta_P(A)[CAB/A] \\
&= \{(A, \{\phi\})\} \cup \{(A, \{\phi\})\}[CAB/A] \\
&= \{(A, \{\phi\})\} \cup \{(A, \{\{C\}\} * \{\phi\}), (B, \{\{A, C\}\} * \{\phi\}), (C, \{\phi\} * \{\phi\})\} \\
&\quad \textit{(by Definition 4.4.18)} \\
&= \{(A, \{\phi\})\} \cup \{(A, \{\{C\}\}), (B, \{\{A, C\}\}), (C, \{\phi\})\} \\
&\quad \textit{(by Definition 4.4.3)} \\
&= \{(A, \{\phi\}), (B, \{\{A, C\}\}), (C, \{\phi\})\} (\text{byDefinition 4.4.11})
\end{aligned}$$

To use these values, in Algorithm 4.5.1, calculations are going on.

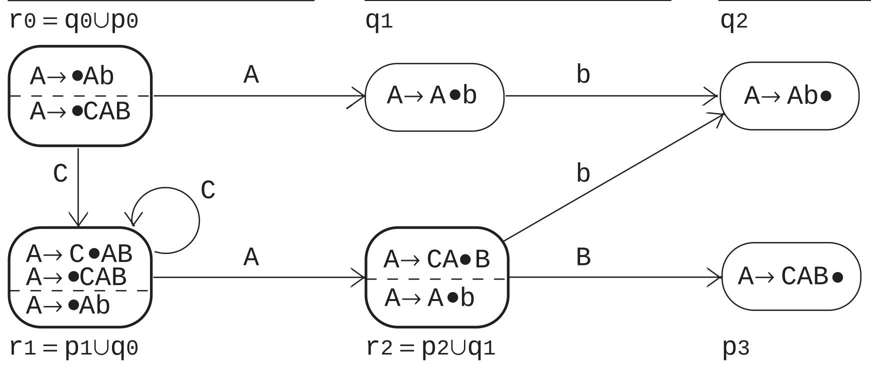
Example 4.6.3 (Fusion)
*Figure 4.12 illustrates the result of fusion of two MonoGs that are presented in previous examples. Ind$T\Delta(A)(\$) =$ **true** of q_0 is inherited to Ind$T\Delta(C)(\$) = \{\{A, B\}\}$ of r_0*

$\text{MaxInc}_{Q \cup P}$	$\{q_0, p_0\}$
$\text{Inclusion}_{Q \cup P}$	$\{q_0, p_0\}$

MaxInc	ϕ	
Inclusion	ϕ	
IndTΔ	(A)($)	**true**
	(A)(b)	**true**
	(C)($)	{{A, B}}
	(C)(b)	{{A, B}}
IndVΔ	(C)(A)	**true**
	(C)(B)	{{A}}
LinkΔ	ϕ	

MaxInc	ϕ
Inclusion	ϕ
IndTΔ	ϕ
IndVΔ	ϕ
LinkΔ	ϕ

MaxInc	ϕ
Inclusion	ϕ
IndTΔ	ϕ
IndVΔ	ϕ
LinkΔ	ϕ



MaxInc	ϕ	
Inclusion	$\{r_0\}$	
IndTΔ	(A)(b)	**true**
	(C)(b)	{{A, B}}
IndVΔ	(A)(B)	**true**
	(C)(A)	**true**
	(C)(B)	{{A}}
LinkΔ	(A)	{(1, A,{{B}})}
	(C)	{(1, A,{{A, B}})}

$\text{MaxInc}_{Q \cup P}$	$\{q_0, p_1\}$
$\text{Inclusion}_{Q \cup P}$	$\{q_0, p_0, p_1, r_0\}$

MaxInc	ϕ	
Inclusion	$\{q_1\}$	
IndTΔ	ϕ	
IndVΔ	ϕ	
LinkΔ	(B)	{(2, A,**true**)}

$\text{MaxInc}_{Q \cup P}$	$\{q_1, p_2\}$
$\text{Inclusion}_{Q \cup P}$	$\{q_1, p_2\}$

MaxInc	ϕ
Inclusion	ϕ
IndTΔ	ϕ
IndVΔ	ϕ
LinkΔ	ϕ

Figure 4.12: Example of Fusion

through inheritance symbol '\$' and its value of $IndT\Delta(C)(\$) = \{\{A, B\}\}$ of p_0. The inheritance is calculated in the part of "On the case of Inheritance" in Algorithm 4.5.2. See the value of $IndV\Delta(C)(C)$ at r_1. Suppose each context leads to state r_1, e.g. $A \Rightarrow C \bullet AB \Rightarrow C \bullet CABB$, $IndV\Delta(C)$ must includes value of $Dep\Delta(ABB)$. $Dep\Delta(ABB)(A) =$ **true** *is trivial, because A occurs at the top of ABB. By Definition 4.4.13, $Dep\Delta(ABB)(B) = [\{\{A\}, \{A, B\}\}]$, and $\{A\} \subset \{A, B\}$, hence, $Dep\Delta(ABB)(B)$ is also $[\{\{A\}\}]$. Thus, $IndV\Delta(C)(B) = \{\{A\}\}$ at state r_1.*

In Figure 4.12, $MaxInc_{(Q\cup P)}$ and $\subset_{(Q\cup P)}$ are presented for each fused states, where $Q = \{q_0, q_1, q_2\}$ and $P = \{p_0, p_1, p_2, p_3\}$. See the values of MaxInc, $MaxInc_{(Q\cup P)}$ at r_1. So r_1 is the result of fusion on q_0 and p_1, and $MaxInc(q_0) = \phi$ and $MaxInc(p_1) = \phi$, $MaxInc_{(Q\cup P)}$ has value $\{q_0, p_1\}$ for $r_1 = q_0 \cup p_1$. However, both of q_0 and p_1 are unreachable from $Ent\varepsilon(X)$ for each $X \in V = \{A, B, C\}$, where $Ent\varepsilon(A) = r_0$, $Ent\varepsilon(B) = Ent\varepsilon(C) = \phi$. It is the reason why $MaxInc(r_1) = \phi$ which is calculated by Definition 4.3.16. We can under stand validity of $MaxInc(r_1) = \phi$ by the fact that r_1 contains a unique item $A \to C \bullet AB$.

4.7 Discussions on Efficiency of the Algorithms

4.7.1 Worst case Analysis of $E\Delta$

Worst case on calculation of $E\Delta$ is quite bad. To see it, we consider on maximum number of elements which are included in a member of DD = $[Power(Power(\Omega))]$. It is trivial that the maximum member of $Power(Power(\Omega))$ is $Power(\Omega)$, which number of elements is $2^{\#(\Omega)}$. However our interest is on $[Power(Power(\Omega))]$. Generally, we can select smallest member as a representative. So

$$\max_{H\in[Power(Power(\Omega))]} \ \min_{H'\in[H]} \#(H')$$

indicates worst case for $E\Delta$, i.e. ${}_nC_{n/2}$, where C gives number of combinations, for $n = \#(\Omega)$, and, ${}_nC_{n/2} \in \Theta(2^{cn})$. As an example of worst case, we present a grammar by production rule set,

$$\begin{aligned}
&Z_2 \to X_1X_2,\ Z_3 \to Z_2X_3, \cdots,\ Z_n \to Z_{n-1}X_n,\\
&Z_{2,1} \to X_2,\ Z_{2,2} \to X_1,\\
&\ldots\\
&Z_{k+1,1} \to Z_{k,1}X_{k+1}, \ldots,\ Z_{k+1,k} \to Z_{k,k}X_{k+1},\ Z_{k+1,k+1} \to Z_k,\\
&\ldots\\
&Z \to Z_{n,1} \mid \ldots \mid Z_{n,n},
\end{aligned}$$

which gives ${}_nC_{n/2}$ for $\Omega = \{X_1, \ldots, X_n\}$. If we adopted number of production rules as a measure, say m, worst case complexity would be $\Theta(2^{c\sqrt{m}})$.

4.7.2 About implementations and complexities of $E\Delta$, $Top\Delta$, $Dep\Delta$

It is not difficult problem how to implement domains of $E\Delta$, $Top\Delta$ and $Dep\Delta$, and operations on them, they are straightforwardly obtained from definitions. So, domains of $Top\Delta$ and $Dep\Delta$ have function types, simplest implementation of them is to prepare tables. However, this choice leads to immense usage of strages. In practical grammars, $Top\Delta$ has almost everywhere **false** values, so does $Dep\Delta$, as seen on examples below. Costs for calculation of operations on $E\Delta$, $Top\Delta$ and $Dep\Delta$ are,

$$\begin{aligned}
[H \cup H'] :&\quad N_V \times \#(H) \times \#(H'),\\
H * H' :&\quad {N_V}^2 \times \#(H) \times \#(H') \times (\#(H) + \#(H')),\\
H'[H/X] :&\quad N_V \times \#(H) \times \#(H')^2,\\
\Lambda \cup \Lambda' :&\quad N_T \times N_V \times m^2,\\
\Lambda * H :&\quad N_T \times {N_V}^2 \times m \times \#(H),\\
\Lambda[\gamma/Z] :&\quad N_T \times N_V \times {m'}^3,
\end{aligned}$$

where $N_V = \#(V)$, $N_T = \#(T)$, $H, H' \in [Power(Power(V))]$, $\Lambda, \Lambda' \in T \to [Power(Power(V))]$, $m = \max_{a \in T} \#(\Lambda(a))$, and $m' = \max\{\#(E\Delta(\gamma)), m\}$, on worst case of simplest implementation. From the discussion on worst case of $E\Delta$ above, these computational costs seem very expensive. However, in practical cases, $\#(H) \ll N_V$ holds, and moreover, almost all values on $[Power(Power(V))]$ which occur during computation are very close to 1 or 2. So, we might be able to adopt followings as a measure for plactical cases,

$$\begin{aligned}
[H \cup H'] :&\quad \ll N_V\\
H * H' :&\quad \ll {N_V}^2\\
H'[H/X] :&\quad \ll N_V,\\
\Lambda \cup \Lambda' :&\quad \ll {N_T}^2,\\
\Lambda * H :&\quad \ll {N_T}^3,\\
\Lambda[\gamma/Z] :&\quad \ll {N_T}^2.
\end{aligned}$$

4.7.3 Worst case analysis of fusion

Here is given an example for worst case of fusion, which is presented in [15] and was constructed by Alan Demer. As described in [15], the example is an inheritance of a general feature of LR parsing scheme.

$$\begin{aligned}
S_0 &\to a\,S_0 \mid b\,S_0,\\
S_0 &\to c\,S_1,\\
S_1 &\to a\,S_2 \mid b\,S_2,\\
S_2 &\to a\,S_3 \mid b\,S_3,\\
&\cdots\\
S_{n-1} &\to a\,S_n \mid b\,S_n,\\
S_n &\to d.
\end{aligned}$$

Number of states of LR(0) graph for above grammar is $4n + 6$. However, if we add a rule $S_0 \to aS_1$ to the grammar, number of states grows to $2^n + 4n + 6$. So, we can conclude that worst case complexity of fusion process is exponential.

4.7.4 About implementation of *MaxInc* and its Time/Space complexity

It is easy to see the efficiency on space usage of *MaxInc* and inclusion information between states. In our approach on state identification, each state identifier, which is naturally given as an integer, represents some set of items. If the item set contains a unique item, then *MaxInc* for the state is ϕ. Thus, with respect with the observation, complexity on space usage never exceeds that of conventional representation of states, i.e. by use of kernel.

The definition of evolution process of *MaxInc*, defined at Definition 4.3.16 and 4.3.17, also ensured the equivalence by Theorem 4.3.18, might seem to be intricate. However, the implementation is quite simple. Firstly, we illustrate calculation process of $\mathit{MaxInc}_{(Q_1 \cup Q_2)}$. Suppose a situation that a new state is obtained by fusion on $q_1, q_2, \ldots, q_k$. To calculate $\mathit{MaxInc}_{(Q_1 \cup Q_2)}$, 1) expand $\{q_1, q_2, \ldots, q_k\}$ to $U = \{q_1, q_2, \ldots, q_k\} \cup \bigcup_{i=1}^{k} \mathit{MaxInc}(q_i)$. 2) find states q', s.t. $\mathit{MaxInc}(q') \neq \phi$ and $\mathit{MaxInc}(q') \subset U$, and add each elements of $\mathit{MaxInc}(q')$ to U. 3) maximalize U. On worst case, at 1) $n \times k$ times of addition of elements are needed, where n is the number of whole states. At 2), to determine relation $\mathit{MaxInc}(q') \subset U$, $n \times n$ times of comparisons on elements in U might be achieved on worst case, and the determination should be done for each states. Thus, the iteration of 2) requires n^3 times of comparisons on elements of U, on worst case. However, as seen at Example 4.6.3, values of *MaxInc* almost of all states are ϕ, in general, time complexity needed to calculate *MaxInc* of new state is very close to n.

Calculation process of MaxInc_A is also achieved by expansion and reduction of elements, same as $\mathit{MaxInc}_{(Q_1 \cup Q_2)}$, excepting such a case that if unreachable states are contained, the value of *MaxInc* is determined directly to be ϕ.

4.8 About LR Parsing on RCFG

Example 3.2.2 is one of typincal cases which bring out the difference between CFG and RCFG. It is cleared by Greibach Normal Form, explained in [14] etc., that candidates of first terminal symbols of CFL are predictable from given production rule set. In other words, if first terminal symbol is given, candidates of production rules used first on derivation is predictable on CFG. This feature is not affirmative on RCFG.

Suppose LR parsing for RCFG, there are RCFG and inputs which rise up a situation that initial parse table does not provide any action on initial state for given input, but given RCFG must accepts the input without no ambiguity. Example 3.2.2 provides such a case. LALR(1) graph calculated from initial production rule set P given in RCFG G_3 has empty initial state. In conventional meaning, the situation means error.

In this section, we treat two kinds of restrictions on RCFG so that LALR(1) parsing scheme is applicable to those of restrictions on RCFG classes. In this paper, only the outline is presented.

First candidate of restrictions is of a simple choice, which restricts RCFG to a class, named RCFG-S, so that RCFG such that causes tricky case is not allowed. It concerns to the case that the definition of derivation on RCFG, see Definition 3.1.5, is restricted so that constraints $A \rightarrow X_1 \cdots X_n \in P_1$ and $\mathbf{p} \rightarrow [X_0 \rhd X_1 \cdots X_n] \in P_0$ are adopted instead of $A \rightarrow X_1 \cdots X_n \in P_{n+1}$ on case 1) and $\mathbf{p} \rightarrow [X_0 \rhd X_1 \cdots X_n] \in P_1'$ on case 2), respectively. Figure 3.2 illustrates the effective range of embedded production rule on

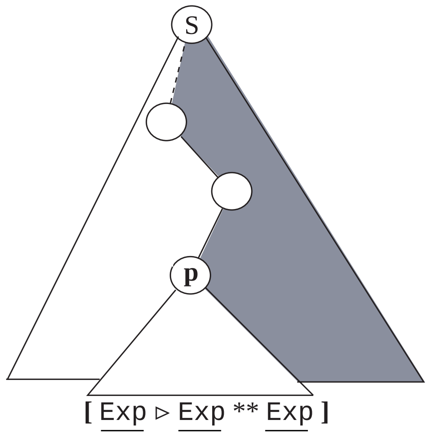


Figure 4.13: Effective Range of restricted RCFG

derivation tree for normal RCFG, while Figure 4.13 illustrates the restricted one. The difference appears on a boundary. On normal RCFG definition, newly added production rules are enabled to be used as ancestors of $\mathbf{p} \in \mathbf{Aug}$ on derivation trees, which cause the augmentations, while restricted RCFG does not allow this feature.

When this simple restriction is adopted, parsing using LALR(1) scheme is simple, described as below,

1) construct initial LALR(1) graph by the incremental method,

2) restrict domains of DD from V to $f(D)$, using restriction method defined in Definition 4.4.30,

3) calculate LA by use of Exp,

(start parsing)
repeat 4), until parse completes,

4) if reduce action is done with an item $\mathbf{p} \to \alpha\bullet$, where $\mathbf{p} \in \mathbf{Aug}$, then augmenting a new rule by use of Algorithm 4.5.4, and calculate LA for new LALR(1) graph obtained by augmentation.

Of cource, if Shit-Reduce or Reduce-Reduce confliction occurs as a result of augmentation at 4), the system discontinues to parse it as an error.

The second choice is illustrated as following,

1) construct initial LALR(1) graph by the incremental method,

2) restrict domains of DD from V to $f(D)$, using restriction method defined in Definition 4.4.30,

3) calculate LA by use of Exp,

(start parsing)
repeat 4), 5), until parse completes,

4) if reduce action is done with an item $\mathbf{p} \to \alpha\bullet$, where $\mathbf{p} \in \mathbf{Aug}$, then augmenting a new rule by use of Algorithm 4.5.4, calculate LA for new LALR(1) graph obtained by augmentation,

5) if parser stops with error, that means no action at that time, then try to detect embedded portion in the rest of input. If detected a rule, augmenting the rule to current LALR(1) graph, but mark on reduce action of detected rule so that the reduce action is effective after the detected position.

Conventional LALR(1) graph corresponds to the graph starts from $Ent\varepsilon(S')$ on our method. To achieve detection of 5), we can use another sub-graphs start from $Ent\varepsilon(\mathbf{p})$ for each $\mathbf{p} \in \mathbf{Aug}$. Process of 5) must be non-deterministic.

Chapter 5

Conclusion and Future Works

We formalize a formal language model as RCFG, which deals with texts including augmentations of part of grammar rules, show RCFG has good properties and its language class, i.e. in between CFL and CSL, and give an efficient parsing algorithm. Most of formal language systems which have expressive power stronger than CFG have no efficient parsing algorithms, and have to be restricted to some subsets of them. Such a restriction is one of main causes of loss of clarity and readability of their descriptions. We consider that RCFG will be helpful base model for system programmers to construct self-referential systems or to use it as a rapidly prototyping system, because of the simple and natural feature of RCFG

An incremental construction method of LALR(1) parser in which method any item sets are not used, and applications of the method to parsers for RCFG are illustrated. We introduce notions Mono-Graph, fusion, *MaxInc*, DD, $Ind\Delta$ and operations on them in order to achieve it, and establish some properties on them, especially establish that the method proposed in this paper induces LALR(1) parser. And also, we present a set of algorithms for the method, and discuss on the efficiency of the method, on worst case. Typical points of our work are the introduction of notions for incremental construction for both of LR(0) graphs and indices on Look Ahead Sets in fully incremental manner and no use of item sets.

To use this method in practice, there are several problems must be solved. As parsing problems, ways to treat ambiguous grammars and frameworks for error handling are remained. Additionally, most important problem that how to give semantic descriptions to newly added production rules are remained as future works. A hint to the problem on semantics is discussed in Appendix A. Additionally, $Ind\Delta$, index for incremental calculation of LA, gives us a possibility of static analysis of grammars. Because informations provided by $Ind\Delta$ indicates dependencies between production rules, it may be useful on grammar debugging.

Appendix A

Discussions on Semantic Descriptions for Augmented Rules

A.1 Motivation on Semantic Description for Augmented Rules

It is one of most important problems how to give a semantics for dynamically extensible grammars. Of course, the problem depends on formalisms of such extensible grammars. However, we can observe that most of extensible grammars based on CFG have mechanisms to restrict domains of production rules which will be augmented at parse-time. That is to say, it does not mean that arbitrary production rules are augmented to initial production rule set. Only production rules which are supposed to be augmented in the initial grammar are augmented. RCFG is one of such formal language systems. With respect to this observation, we can consider some models on giving semantics to dynamically augmented production rules. Followings are both extremes in such models,

1) giving semantics in advance for all production rules which might be augmented,

2) giving frame works for users to define semantic rules for augmented production rules in identical manner of the other rules.

Model 1) gives us an impression that there is no merit to have dynamically extensibility on syntax. However, it is not true. For example, we should consider the case of the overloading mechanism on operators in C++. gcc gives us a good example for it. In C++, no new operator is enabled to be defined as a new token. Only enabled is over-loading. What function is substituted to an operator, in other words, how to give a semantics to an operator, is remained for users (programmars). Under this sense, we can conclude that C++ has flexibility, and is not a language which has merely fixed semantics. In fact, in the implementation of gcc, a unique function is assigned as a semantic function to every production rules concerning to 'Expression', in which function codes for expressions are generated accoding to their types or classes. Semantic assignment in this way is one of choices. In such a way, it is important problem how to model a mechanism in semantic functions to assign semantics to each syntactical entities, and the mechanism seems to be valuable as an object of studies. However, we do not adopt the model.

We tentatively call model 2) *Compile-time Reflection.* In this paper, we introduce two kinds of implementation models on Compile-time Reflection. Essentially, Compile-

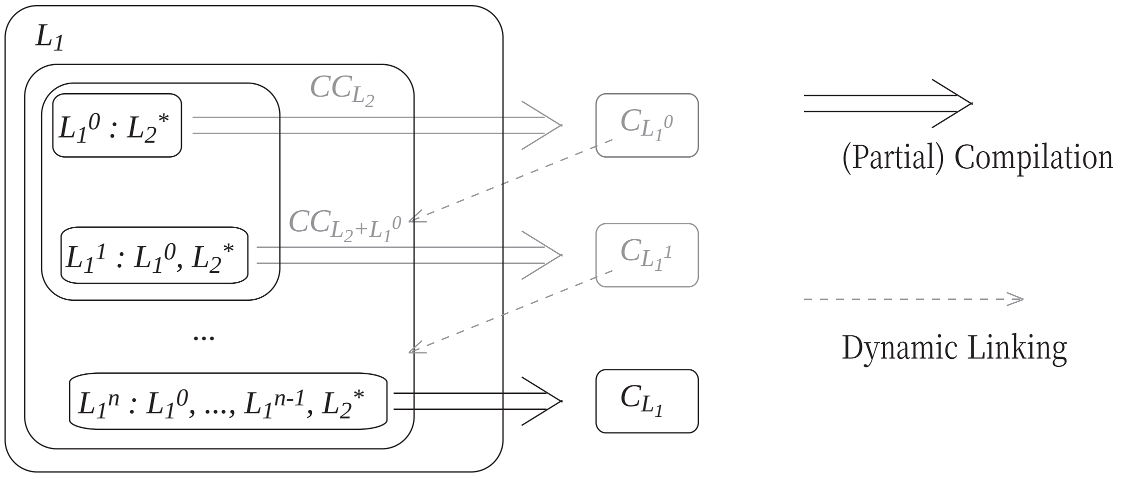


Figure A.1: Diagram of Compile-time Reflection

time Reflection is regarded as a mechanism to embed object codes, i.e. generated by the compiler, into the compiler process itself, or some other mechanism which is similar to the embedment (see Figure A.1). This mechanism will supply facilities to developpers of programming language systems on some aspects. Adopting the mechanism, developpers will be provided a step-by-step development environment. As another aspect, embedding object codes into compiler process itself means that compiler can seize higher mechanisms which are realized as functions of programming languages which are processed by the compiler. This leads us to an expectation of easiness to develop compilers.

Frankly speaking without fear to misleading, Reflection can be regarded as confusion of meta objects and first order objects, or, inclusion of objects in either directions. However, the core parts of any reflective systems are initially stated. Reflective features of such systems are constructed or proceeded starting from the core parts and piling up on them. This aspect is explained with the words of *meta-circularity* or *meta-circular interpreter* [24, 29, 32] . On structures of systems which include reflective mechanisms in the descriptions of compilers of them, there are hierarchies similar to [29]. (Figure A.2)

It is able to discuss on properties of programming languages of which compilers have the reflective hierarchical structure, i.e. explained above, but, we focus on features of development process of programming languages, using compiler-compiler which has mechanisms including the hierarchical structure. For any software, it is able to divide its each parts into some hierarchy, although the devision is not clear. For example, basic operations such as memory managements, further higher level processes which use the basic operations, main operations, user interfaces, and so on. Usually, during development, developpers continue coding and debugging process, mutually iterating them. They can shift jobs to coding and/or debugging on higher level parts of the system after completing verification on lower level components. Total jobs form a spiral of development cycle. On developments of programming language systems, using development tools which have reflective hierarchical structure, developpers obtain freedom so as to rise up higher mechanisms of target systems into the description of compiler itself at some time of development cycle.

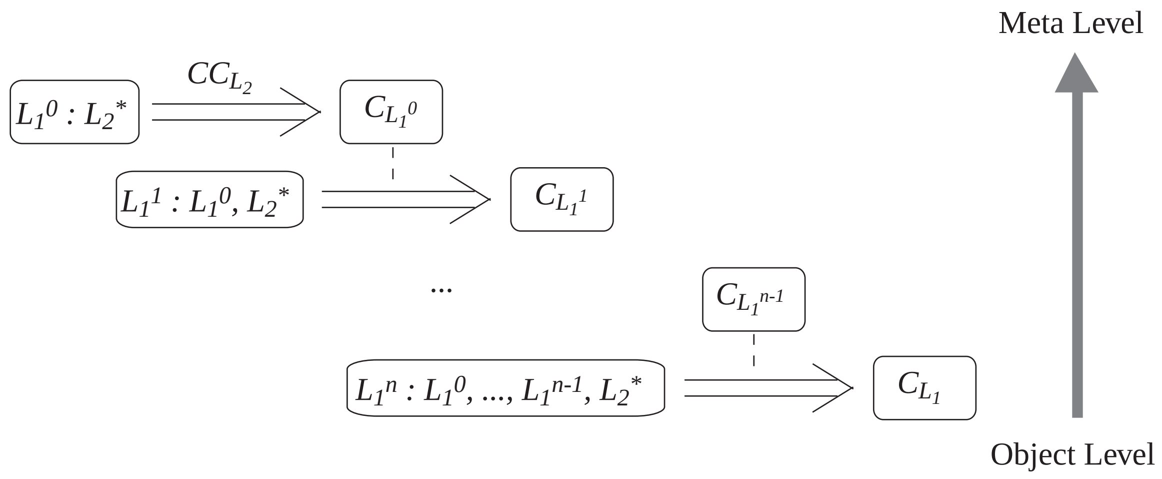


Figure A.2: Meta-circularity

For example, here we consider development of a programming language which has function of garbage collection on memory management. When completing coding and debugging of the part of memory management, if the function of the part can be involved in the description of the compiler, developper obtains the merit of the function at the phase of development of the compiler itself. As similar examples, we can enumerate type checking mechanism, calculation of array indeces which deeply depends on types, and pattern matching mechanism, so well. Of course, it is obvious that these examples are realized if there are compatibility between paradigm of compiler description language and that of target language, and exists an interface between them.

A conceptual diagram which reflects the intuition on the circularity of reflective hierarchy on compiling process, to say meta-circular compilation, which is explained so far, is given in Figure A.1. This diagram explains an implementation model of meta-circular compilation, under which object codes are partially generated and the codes might be involved in compiler via dynamic linking process. To say, the diagram describes the involvement of functions of target languages. Besides this implementation model, we can consider an another implementation model in which some parts of compiler codes are embedded in object codes. (Figrue A.3) There seems to be at least two implementation models of meta-circularity via compilation process, such that,

1) involving object codes into compiler process,

2) embedding of some parts of compiler codes into object codes.

Focusing on efficiency of generation of codes, in general, it is easy to predict that model 1) provides an enfficient frame work. Before easily concluding so, we try to discuss on each models, because it is expected that there is significant difference depending on where reflective descriptions occur, and on how to implement reflective functions.

We start discussion with an example. Here, we suppose that target system of development is a language system, to say further a compiler, and the object of discussion is a frame work for compiler-compiler with which compilers that have reflective functions

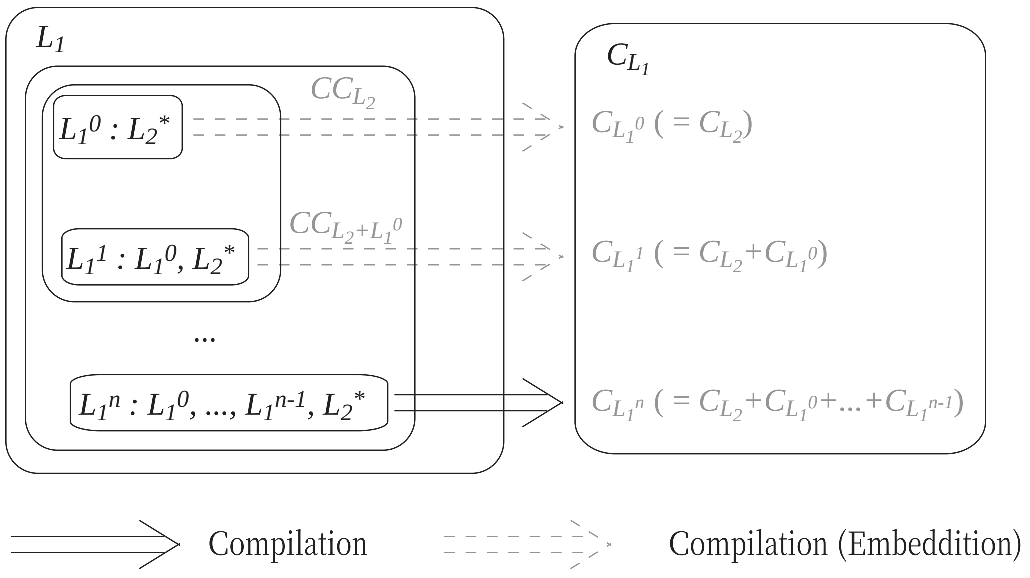

L_1
$L_1^0 : L_2^*$
CC_{L_2}
$CC_{L_2+L_1^0}$
$L_1^1 : L_1^0, L_2^*$
...
$L_1^n : L_1^0, ..., L_1^{n-1}, L_2^*$
C_{L_1}
$C_{L_1^0}$ (= C_{L_2})
$C_{L_1^1}$ (= C_{L_2}+$C_{L_1^0}$)
$C_{L_1^n}$ (= C_{L_2}+$C_{L_1^0}$+...+$C_{L_1^{n-1}}$)
Compilation
Compilation (Embeddition)

Figure A.3: A way of inprementation by embedding meta-codes

are described. Individual reflective programming languages are not the objects of the discussions here. To clear our stance, we are conscious of description levels of programming languages, such as,

A) level compiler-compiler (Level CC),

B) level compiler (Level C),

C) level object codes (Level Obj).

Moreover, we adopt RCFG for a base grammar. From the property of RCFG (Theorem 3.3.7), there is no problem to assume CFG as a base grammar, excepting the dynamical extensibility of RCFG. For semantic descriptions, we adopt Attributted Grammars (AG) [20, 21] on Level C, and Syntax Directed Translation (SDT) which has been used to describe flexible syntax languages, such as [23] and its resemblances.

A.2 An example of description by SDT

In this section, we give an example to introduce a string notation into the syntax of pure-Prolog. In this example, introduced string is interpreted as a list which consists of atoms, each atom has one-length name, such that "*abc*" is interpreted to $[a, b, c]$. Firstly, we suppose that production rules,

$$\begin{array}{rcl} Term & \rightarrow & List \mid Functor \\ List & \rightarrow & '[' ']' \mid '['Term\text{-}Seq']' \mid '['Term\text{-}Seq' \mid' Term']' \\ Term\text{-}Seq & \rightarrow & Term \mid Term','Term\text{-}Seq \\ Functor & \rightarrow & Id \mid Id'('Term\text{-}seq')' \end{array}$$

are included in the syntax of pure-Prolog. We consider that the introduction of string notation causes augmentation of new production rule

$$Term \rightarrow \ '"' \ Id \ '"'.$$

This introduction of string notation is given by SDT description as below.

$$[\underline{List} \triangleright \ " \ Id \ " \ ] \tag{A.1}$$

$$\textbf{synfn}(" \ Id(str) \ "):\underline{List} \tag{A.2}$$

```
begin
    if $str$ = emptyString then $[\,]$
    else $[Id \downarrow \textbf{head}(str) \ | \ "Id \downarrow \textbf{rest}(str)"]$
end
```

A.1 represents an augmentation of new production rule, and the description A.2 provides semantics for the rule. Formal definition of SDT is stated in later section. Here, we give an intuitive explanations for it. (" $Id(str)$ ") following **synfn** is a *Signature* with parameters. The head line of SDT **synfn**(" $Id(str)$ "):$\underline{List}$ means that it provides the mean for a production rule which left-hand side is $\underline{List}$ and which right-hand side is " Id ". *str* represents a set of attribute values of Id, and we assume that *string attribute value* is included in it. In this example, we assume that every symbols which occur in signatures have *string* as an attribute, and that every *string* attributes hold terminal symbol sequeces which is derived from symbols concerned to the attributes. This description lacks accuracy, because input string sequeces for parse module are sequences of tokens cut out by lexcal analyser, which do not occur in program texts, on actual compilers. Values of *string* attributes concern rather to values which are held in a variable yytext of YACC. In the meaning of SDT of this example, the signature " Id " represents a requirment that on some portion of input of parser, if there is a occurence of sequence " Id ", then the attribute of second symbol Id must be assigned to *str*. If the value of *string* attibute held in *str* is empty string, then result of SDT is $[\,]$, empty list, otherwise, the result is constructed in a way that firstly constructing a list for (" $Id \downarrow \textbf{rest}(str)$ "), which means a list concerns to a string of *str* from which the first symbol is taken away, then appending the first symbol of *str* to the list. $Id \downarrow < string >$ represents a generation of new syntactic entity, and a coercion of an attribute set to the generated entity. From a given symbol sequence "*abc*", using SDT given by A.1 and A.2, a translation sequence will be achieved as

$$"abc" \Rightarrow [a \mid "bc"] \Rightarrow [a, b \mid "c"] \Rightarrow [a, b, c \mid [\,]],$$

then, finally obtained symbol sequence which represents a list will be parsed by the parser with initial semantic interpretation.

If we focus on the function of translations of strings which are achieve by SDT, SDT might be seen as a macro language, like GNU m4. However, SDT is differ from macro languages such as GNU m4 on the point that an SDT description is defined with a new production rule, so the definition causes an extension of grammar. While macro expansions are achieved by sub-string matching, SDT requires on its applications that each translatees have context which the signature of the SDT should have. Moreover, because SDT is applied during parsing, informations which are fixed on parsing, e.g. types of expressions, are available to be used in SDT. So, on SDT, it is able to choose translations according to types of translatees. These are the essentially different points from macro languages.

A.3 An Implementation of SDT

Here, we try to generalize SDT to a system which has enough features to be called as compile-time reflection. There seems to be at least two ways of generalizations. One way is to adopt some reflective mechanism on the parser as an implementation of SDT instead of string replacements. The other is to enable to use user defined functions alike system functions such as **if then else**, **head**, **rest** used in above example. User defined functions indicates functions of the language system which is being constructed using SDT.

We begin discussions firstly on the former stance so as to lead us to the later stance, adopting Attributted Grammars as a description frame work.

A.3.1 Attributted Grammars

Attributted Grammars (AG) had been proposed in [20, 21], and have been used for descriptions of Syntax Oriented Systems. For given CFG $G = (V, T, P, S)$, two kinds of attributted values, so called *Inherited Attribute* and *Synthesized Attributes* are augmented for each X in V. For each terminal symbols in T, only Synthesized Attributes are usually augmented. Intuitively, Inherited Attributes of X can be regarded as inputs of sub-trees of parse trees which has X as root node. Also Synthesized Attributes of X can be regarded as outputs, e.g. [19]. Rules for calculation of attributted values, called *Attribution* are appended to each production rules. In each attributions, a list of procedures which calculate Synthesized Attributes of the symbol on left-hand side of the production rule and Inherited Attributes of the symbols on right-hand side, from Inherited Attributes of the symbol on left-hand side and Synthesized Attributes of the symbols on right-hand side, is provided. An example illustrating a tiny system in which values for binary strings are calculated are given as below.

An example of AG

$$\begin{aligned} G &= (V, T, P, S) \\ V &= \{S, B\},\ T = \{0, 1\}, \\ P &= \{S \to B, \\ &\quad B \to 0,\ B \to 1, \\ &\quad B \to B\,0,\ B \to B\,1\} \end{aligned}$$

$$In(S) = \phi,\ \ Syn(S) = \{value\}$$

$$In(B) = \{figure\},\ \ Syn(B) = \{value\}$$

$$S \to B \qquad \{B.figure = 1;\ S.value = B.value\}$$

$$
\begin{array}{rl}
B \to 0 & \{B.value = 0\} \\
B \to 1 & \{B.value = B.figure\} \\
B \to B\,0 & \{B[1].figure = B.figure * 2; \\
 & B.value = B[1].value\} \\
B \to B\,1 & \{B[1].figure = B.figure * 2; \\
 & B.value = B[1].value + B.figure\}
\end{array}
$$

In this paper, we adopt a notation that each symbols appears in right-hand side of production rules are appended with a number of its occurence position in order to distinguish different occurences of same symbols, e.g. $B[1]$ designates symbol B which occurs at first position of right-hand side of its rule.

It is not a trivial problem to determine evaluation order of each attributions in AG. From descriptions of Attributions, dependency relations between attribute values can be established. Generally, depending on given parse tree, the dependency may consist of circularities. AGs which contain no circularity on the dependencies between attribute values, for any words generated from G, are called Well-defined Attributted Grammars (WAG). Sevral sub-classes of WAG have been proposed from sevral point of view [4, 17, 18], such as LR Attributted Grammar (LR-AG) [17, 18] which is intended that each attribute values are calculated during LR parsing are going.

A.3.2 An implementation of SDT using AG

Here, we give an implementation of SDT under the frame work of AG.

Example A.3.1 (Example of a semantics for *synfn*)

Inherited Attributes:
(*string* $*$ *Type list*)
BODY.id_list, *SF_EXP.id_list*, *BOOL_EXP.id_list*

Synthesized Attributes:
(*string* $*$ *Type*)
SIGN.id

string $*$ *Type*) *list*
SIGN_LIST.id_list

(*string*)
identifier.w

(*Type*)
SDTYPE.type, *SMTYPE.type*, *STYPE.type*, *BASETYPE.type*, *TYPE.type*

($\lambda expression$)
SIGN_LIST.exp
BODY.exp
SF_EXP.exp

$FUN1ARG.exp$

(where $Type = string$)

$$
\begin{aligned}
&SYNFUN \rightarrow \textbf{synfn}\,identifier\,(\,SIGN\_LIST\,) : SDTYPE\,BODY\\
&\qquad\{SIGN\_LIST[4].pre\_list = [];\\
&\qquad BODY.id\_list = SIGN\_LIST[4].id\_list;\\
&\qquad \textbf{reflect}((SDTYPE[7].type, SIGN\_LIST[4].type),\\
&\qquad (\lambda\mu : string.SIGN\_LIST[4].exp)BODY[8].exp)\}
\end{aligned} \tag{3}
$$

$SDTYPE \rightarrow X$ (where $X \in D$)
$\{SDTYPE.type = X.w\}$

$SMTYPE \rightarrow Y$ (where $Y \in M$)
$\{SMTYPE.type = Y.w\}$

$STYPE \rightarrow Y$ (where $Y \in T$)
$\{STYPE.type = Y.w\}$ Note: $D \subset M \subset T$

$BASETYPE \rightarrow STYPE$
$\{BASETYPE.type = STYPE[1].type\}$
$BASETYPE \rightarrow bool$
$\{BASETYPE.type = bool\}$
$BASETYPE \rightarrow char$
$\{BASETYPE.type = char\}$
$BASETYPE \rightarrow string$
$\{BASETYPE.type = string\}$
$BASETYPE \rightarrow tree$
$\{BASETYPE.type = tree\}$

$TYPE \rightarrow' identifier$
$\{TYPE.type =' :: identifier[2].w\}$
$TYPE \rightarrow BASETYPE$
$\{TYPE.type = BASETYPE[1].type\}$
$TYPE \rightarrow TYPElist$
$\{TYPE.type = TYPE[1].type :: list\}$

$SIGN \rightarrow STYPE$
$\{SIGN.type = STYPE[1].type;$
$SIGN.id = (\textbf{emptySymbol}, STYPE[1].type)\}$
$SIGN \rightarrow STYPE(identifier)$
$\{SIGN.type = STYPE[1].type;$
$SIGN.id = (identifier[3].w, STYPE[1].type)\}$

$SIGN\_LIST \rightarrow SIGN$
$\{SIGN\_LIST.type = SIGN[1].type;$
$SIGN\_LIST.id\_list = [SIGN[1].id];$

$SIGN\_LIST.exp$

$$= \lambda\pi_1(SIGN[1].id) : \pi_2(SIGN[1].id) \uparrow \mu\} \qquad (4)$$

$SIGN\_LIST \rightarrow SIGN,\ SIGN\_LIST$

$\{SIGN\_LIST.type = SIGN[1].type :: SIGN\_LIST[3].type;$

$SIGN\_LIST.id\_list = SIGN[1].id :: SIGN\_LIST[3].id\_list;$

$SIGN\_LIST.exp = \lambda\pi_1(SIGN[1].id) : \pi_2(SIGN[1].id).SIGN\_LIST[3].exp\}$

$BODY \rightarrow$ **begin** SF_EXP **end**

$\{SF\_EXP[2].id\_list = BODY.id\_list;$

$BODY.exp = SF\_EXP.exp$

$\}$

$SF\_EXP \rightarrow identifier$

$\{SF\_EXP.exp = \mathbf{getValue}(SF\_EXP.id\_list, identifier[1].w)\}$

$SF\_EXP \rightarrow []$

$\{SF\_EXP.w = \varepsilon; SF\_EXP.tree = (\varepsilon, [])\}$

$SF\_EXP \rightarrow\ if\ BOOL\_EXP\, then\, SF\_EXP\, else SF\_EXP$

$\{BOOL\_EXP[2].id\_list = SF\_EXP.id\_list;$

$SF\_EXP[4].id\_list = SF\_EXP.id\_list;$

$SF\_EXP[6].id\_list = SF\_EXP.id\_list;$

$SF\_EXP.exp$

$= if(BOOL\_EXP[2].exp, SF\_EXP[4].exp, SF\_EXP[6].exp)\}$

$SF\_EXP \rightarrow FUN1ARG(SF\_EXP)$

$\{SF\_EXP[3].id\_list = SF\_EXP.id\_list;$

$SF\_EXP.exp = FUN1ARG.exp(SF\_EXP[3].exp)\}$

$SF\_EXP \rightarrow (SF\_EXP)$

$\{SF\_EXP[2].id\_list = SF\_EXP.id\_list;$

$SF\_EXP.exp = SF\_EXP[2].exp\}$

$SF\_EXP \rightarrow SF\_EXP :: SF\_EXP$

$\{SF\_EXP[1].id\_list = SF\_EXP.id\_list;$

$SF\_EXP[3].id\_list = SF\_EXP.id\_list;$

$SF\_EXP.exp = \mathbf{concatinate}(SF\_EXP[1].exp, SF\_EXP[3].exp)\}$

$SF\_EXP \rightarrow SMTYPE \downarrow SF\_EXP$

$\{SF\_EXP[3].id\_list = SF\_EXP.id\_list;$

$SF\_EXP.exp = \mathbf{getToken}(SMTYPE[1].type,$

$\mathbf{getvalue}(SF\_EXP.id\_list,$

$SF\_EXP[3].exp))\}$

$BOOL\_EXP \rightarrow empty(SF\_EXP)$

$\{SF\_EXP[3].id\_list = BOOL\_EXP.id\_list;$

$BOOL\_EXP.exp = \mathbf{empty}(SF\_EXP[3].exp)\}$

$BOOL\_EXP \rightarrow SF\_EXP = SF\_EXP$

$\{SF\_EXP[1].id\_list = BOOL\_EXP.id\_list;$

$SF\_EXP[3].id\_list = BOOL\_EXP.id\_list;$

$BOOL\_EXP.exp = \mathbf{equal}(SF\_EXP[1].exp, SF\_EXP[3].exp)\}$

$BOOL\_EXP \rightarrow (BOOL\_EXP)$

$\{BOOL\_EXP[2].id\_list = BOOL\_EXP.id\_list;$

$$
\begin{array}{l}
\quad\quad BOOL\_EXP.exp = BOOL\_EXP[2].exp\} \\
BOOL\_EXP \rightarrow not\, BOOL\_EXP \\
\quad\quad \{BOOL\_EXP[2].id\_list = BOOL\_EXP.id\_list; \\
\quad\quad BOOL\_EXP.exp = \mathbf{not}(BOOL\_EXP[2].exp)\} \\
BOOL\_EXP \rightarrow BOOL\_EXP\, or\, BOOL\_EXP \\
\quad\quad \{BOOL\_EXP[1].id\_list = BOOL\_EXP.id\_list; \\
\quad\quad BOOL\_EXP[3].id\_list = BOOL\_EXP.id\_list; \\
\quad\quad BOOL\_EXP.exp = \mathbf{or}(BOOL\_EXP[1].exp, BOOL\_EXP[3].exp)\} \\
BOOL\_EXP \rightarrow BOOL\_EXP\, and\, BOOL\_EXP \\
\quad\quad \{BOOL\_EXP[1].id\_list = BOOL\_EXP.id\_list; \\
\quad\quad BOOL\_EXP[3].id\_list = BOOL\_EXP.id\_list; \\
\quad\quad BOOL\_EXP.exp = \mathbf{and}(BOOL\_EXP[1].exp, BOOL\_EXP[3].exp)\}
\end{array}
$$

$$
\begin{array}{ll}
FUN1ARG \rightarrow head & \\
 & \{FUN1ARG.exp = \mathbf{head}\} \\
FUN1ARG \rightarrow rest & \\
 & \{FUN1ARG.exp = \mathbf{rest}\} \\
FUN1ARG \rightarrow root & \\
 & \{FUN1ARG.exp = \mathbf{root}\} \\
FUN1ARG \rightarrow children & \\
 & \{FUN1ARG.exp = \mathbf{children}\}
\end{array}
$$

Note: the attributes *SF_EXP.exp*, *BOOL_EXP.exp* and *FUN1ARG.exp* hold λ expressions as their values. It does not means that we treat λ expressions directly, but means that each λ expression expresses a continuation for evaluations of attribute values.

In this example, a continuation of calculation of a symbol sequence which will be the result of SDT is going to be constructed in attributes *SF_EXP.exp* and *BOOL_EXP.exp*. The continuation obtained by the example is to calculate a symbol sequence, not to calculate attribute values for the symbols sequence provided by SDT. A procedure to calculate expected symbol sequence is assigned by system function **reflect**, i.e. used in (3), to the production rule which is designated by the signature of SDT description. $\uparrow$ in expression (4) parses its argument and constructs its parse tree. The result of $\uparrow$ is a function from a collection of Inherited Attributes of the root of the parse tree to a collection of Synthesized Attributes of the root, the function which is explained in previous section. On this aspect, out approach is similar to Higher Order Attributted Grammar (HAG) [28].

HAG is a model of multi-pass processes. On first pass, data which has tree structure are calculated in a way of AG frame work. On later passes, using tree structured data as a parse tree, which are calculated on previous pass, and adapting AG frame work again, targer objects are obtained finally. In our implementation scheme for SDT given here, when SDT is parsed, a symbol sequence, which may include some non-terminal symbols, is calculated as a attribute value, then the symbols sequence is parsed again by same parser, and finally parser obtains a collection of Attribution for given SDT.

We can assume that this implementation of SDT indicates that the compiler involves an interpreter for SDT. (Figure A.4)

As an alternative imprementation of SDT, instead that compiler includes SDT interpreter, we suppose to implement SDT so as to embed SDT processes into target objects. It is to say SDT compiler. The idea for the implementation is quite simple. To explain

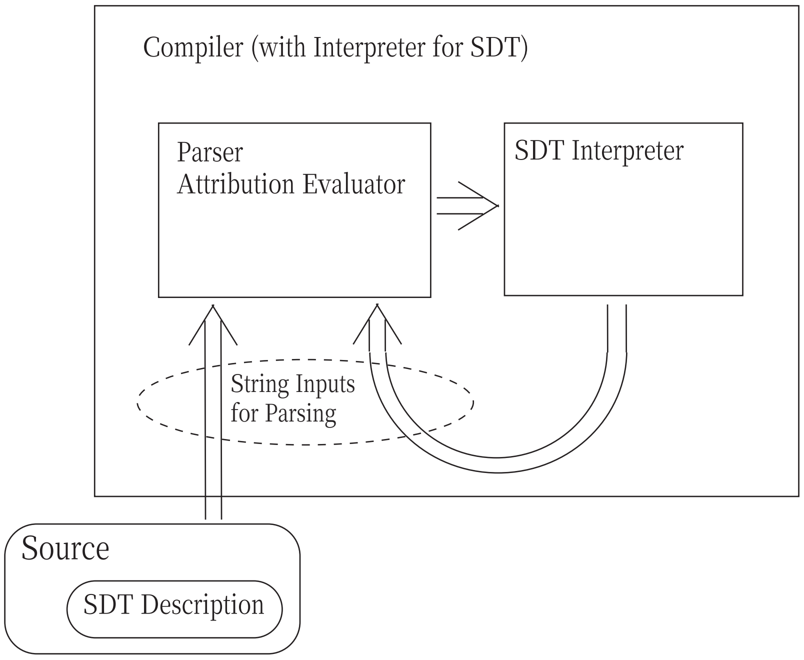


Figure A.4: SDT Interpreter

it with above example, in this implementation, system functions, like **if**, **head**, **rest**, . . ., are not assumed for the functions which evaluate their values immediately, but functions which calculate code streams, i.e. continuations, to evaluate original system functions. This methodology corresponds to Unfolding process which is discussed in the sphere of program translations. We must note that branches in generated code streams are caused not only by system function **if**, but also by a process which concerns to a parsing process caused by ”↑”.

A.3.3 Discussion on the Choice of Implementation; Interpretation or Embedment

So far, for the methodology of implementing SDT, we have discussed on two ways, one is to involve SDT interpreter in compiler, and the other to embed SDT procresses in target object codes via Unfolding. Here, we discuss and compare them from the points of view of software development stages. We call the former *Interpretation method*, the latter *Embedment Method*, in followings.

On the efficiency of generated codes, as described in the first section, it is expected that Interpreter Method has advantage. In spite of inefficiency of Embedment Method, it has advantages. One of them is the posibility to realize a system which includes self-compiler in it. Such a system is realized via embedded codes concerning to SDT interpreter, and the code generates codes and/or data which can be regarded as function closures. However, this aspects of Embedded Method is not the subject of this paper. It is assumed to be harmful under the discussions stated below. The subject here is development cycles of programming language systems. On this view point, most important feature of Embedment Method is for developpers to debug and/or varification of developping systems on object codes level.

Considering the advantage of Interpreter Method on efficiency of code generation and the advantage of Embedment Method on debugging environment, there is one posibility not to select one from these two methods, but to select one in accoding to stages of development. On first stage of development, developpers achieve coding and debugging on core parts of the system. On middle stages of development, using Embedment Method, developpers are going on coding and debugging without obstacles such as stack data for parsing which are already varified. On final stage, using Interpreter Method, efficient codes are obtained. We can adopt such a step wise development process. The varification of the step-wise development on actual systems is a future work.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] A. V. Aho and J. D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[3] A.V. Aho and S.C. Johnson. LR parsing. *ACM Computing Surveys*, 6(2):99–124, 1974.

[4] G. V. Bochmann. Semantic evaluation from left to right. *CACM*, 19:55–62, 1976.

[5] B. Bollobas. *Modern Graph Theory.* Springer-Verlag, 1998.

[6] B. Burshteyn. Generation and recognition of formal languages by modifiable grammars. *SIGPLAN Notices*, 25(12):45–53, 1990.

[7] B. Burshteyn. On the modification of the formal grammar at parse time. *SIGPLAN Notices*, 25(5):117–123, 1990.

[8] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Notices*, 25(11):35–44, 1990.

[9] A. Colmerauer. Metamorphosis grammars. In *Lecture Notes in Computer Science 64*, pages 133–189. Springer-Verlag, 1978.

[10] C. Donnelly and R. Stallman. *Bison: The YACC-compatible Parser Generator Bison Version 1.25*, Nov. 1995. http://www.gnu.org/manual/bison/index.html.

[11] J. Earley. An efficient context-free parsing algorithm. *Comm. ACM*, 13(2):94–102, 1970.

[12] G. Fischer. *Incremental LR (1) parser construction as an aid to syntactical extensibility.* PhD thesis, Dortmund Univ., 1980. Tech. Report 102, Department of Computer Science, Univ. of Dortmund, Federal Republic of Germany, 1980.

[13] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990. Also in SIGPLAN Notices, 24(7):179-191, 1989.

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

[15] R. N. Horspool. Incremental generation of LR parsers. *J. of Computer Languages*, 15(4):205–223, 1990.

[16] S. C. Johnson. *YACC: yet another compiler-compiler.* Bell Laboratories, unix supplementary documents, vol. 1 edition, 1986.

[17] N. D. Jones and M. Madsen. Attribut-influenced LR parsing. In *LNCS*, volume 94, pages 393–407. Springer-Verlag, 1980.

[18] U. Kastens. Ordered attribute grammars. *Acta Infomatica*, 13:229–256, 1980.

[19] D. Kato. A proposal of categorial attributted grammars. *Computer Software*, 12(2):52–66, 1995. in Japanese.

[20] D. E. Knuth. Semantics of context free languages. *Math. Sys. Theory*, 2(2):127–145, 1968.

[21] D. E. Knuth. Semantics of context free languages. *Math. Sys. Theory*, 5(1):95–96, 1971.

[22] C. H. A. Koster. Affix grammars. In *Proc. of the IFIP Work. Conf. on Algol 68 implementation*, pages 95–109, Amsterdam, 1972. North Holland.

[23] D. Sandberg. LITHE: a language combining a flexible syntax and classes. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 142–145. ACM, 1982.

[24] B. C. Smith. Reflection and semantics in Lisp. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 23–35, 1984.

[25] A. Tanaka and T. Watanabe. An extensible LR parser generator - a case study of composable metalevel extensions -. In *IWPSE99 Proceedings*, pages 1–5, 1999. http://dontaku.csce.kyushu-u.ac.jp/IWPSE99/Proceedings/17.pdf.

[26] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and Algorithms.* John Wiley & Sons, 1992.

[27] A. van Wijngaarden. Orthogonal design and description of formal languages. Technical Report MR 76, 1965.

[28] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. *ACM SIGPLAN Notices*, 24(7):131–145, 1989.

[29] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings 1986 ACM Symposium on LISP and Functional Programming*, pages 298–307, August 1986.

[30] B. Wegbreit. *Extensible programming languages.* Harward University, Cambridge, Massachusetts, 1970. Garland Publishing Inc., New York & London, 1980.

[31] B. Wegbreit. The ECL programming system. In *Proc. of FJCC 39*, pages 253–261. AFIPS, 1971.

[32] R. Weyrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13(1):133–170, 1980.

Publications

[1] D. Kato, “A Proposal of Categorial Attributted Grammars”, Computer Software, Vol. 12, No. 2, pp. 52-66, 1995. (in Japanese)

[2] D. Kato, “A Proposal of Reflective Context Free Grammars”, in Proc. of ITC-CSCC2001 in Tokushima, pp. 556-559, July, 2001.

[3] D. Kato, “Incremental Construction of LALR(1) Parsers and its Application to RCFG”, (under submission).

[4] D. Kato, “Reflective Context-Free Grammar”, (under submission).