Functional 언어와 Data Flow Machine

朴 熙 淳
(圓光大 工大 敎授)

■ 차 례 ■

1. 머릿말
2. Imperative 언어와 Functional 언어
2.1 Imperative 언어
2.2 Functional 언어
2.3 Functional 언어의 장점
2.4 Functional 언어의 문제점
3. Data Flow Graph와 Data Flow처리 방식
4. Data Flow Machine 모델
4.1 MIT모델
4.2 Manchuster모델
4.3 Rumbaugh모델
4.4 DNLI모델
5. 맺는 말
참고문헌

1 머릿말

高速 수퍼 컴퓨터를 만들기 위한 중추적 컴퓨터 構造로서 Control Driven Processor의 並列연결, Data Driven, Demand Driven, Pattern Driver방식들이 연구되고 있다. Control Driven 방식(von Neumann식)은 4世代에 걸친 현재의 컴퓨터 발전에 크게 寄與하여 왔음은 分明하나 고속 並列處理를 위한 시스템에의 適用에는 많은 문제점을 안고 있다. Control Driven Processor의 並列연결에 의하여 大量 데이타의 並列 처리를 試圖하고 있으나 그 適用 범위에 限界가 있고 適用방법 또한 경우에 따라 다르거나 複雜하여 획기적인 發展을 보여주지 못하고 있다.

Data Driven(Flow) 방식은 基本的으로 Neumann식 구조, 동작, 제어 방식과는 다르고 並列처리 能力에 있어 많은 可能性을 갖는다. Neuman식이 Program Counter에 의한 順次的 命令실행인 것에 반하여 Data Driven Machine은 실행될 데이타(operand)가 정해진 位置에 存在(도착)하면 곧 실행한다. 따라서 operand 데이타의 存在 如否(availability)가 명령 실행 여부를 決定하는 주요 사항이며 명령 실행의 순서는 중요시되지 않는다. 이 때문에 非同期 처리가 가능하고 동시 처리 능력이 높아진다.

한편 Data Driven 방식은 High Level 언어인 Functional Language를 사용하거나 Data Flow Graph에 相応하는 Low Leyel언어를 사용하고 있다. Functioal언어는 既存의 Imperative언어(Foratan, Pascal등)에 비하여 많은 장점을 갖고 특히 並列처리 시스템 適用에 强点을 갖는다. 本稿에서는 기존의 Imperative언어의 특징과 Functional언어의 특징 및 장단점을 검토하고 현재까지 발표된 Data Flow Machine중에서 대표적인 모델들의 구조 및 동작 과정을 고찰한다.

2 Imperative 언어와 Functional언어

Functional언어는 Applicative Language, Data Flow Language, Reduction Language 등의 用語로도 표현되며 데이타에 대해 技能(算術, 論理演

算 또는 複寫, 選択, 比較등)을 適用 시킨다는 특징을 갖고 이로 인해 Side effect 등의 문제를 해결한다. Functional 언어의 특징을 검토하기 위해 기존의 Imperative언어와 비교하여 보면 다음과 같다.

2.1 Imperative 언어

Fortran, Pascal등 기존의 Imperative언어의특징들을 순서없이 列擧하면 다음과 같다.

○Imperative언어는 기본적으로 Von Neumann식 컴퓨터의 機械語 명령의 "high level version"이며 주요 기능적 目標는 계산상 필요한 機械(CPU, Register, 메모리 등) 內部의"現在 狀態"를 変更시키는 일이다.

○Programmer가 결정한 変数名(variables) 은 기계어 코드에서 메모리 주소로 변경되고 이 주소로 해당 데이타가 保管된다.

○If-then-else와 같은 制御用 命令文은 條件에 해당하는 데이타를 확인하여 true이면 then이하를 false이면 else이하를 실행하는 jump명령이다.

○置換文은 메모리로부터 데이타의 Load 및 Store이다.

○Program Counter, Register, Stack, 메모리등 장치를 이용한 모든 데이타의 "현재상태"의 변경으로서 프로그래머는 계산 논리상 필요한 일 이외의 일에 대해서도 많은 관심을 가져야 한다.

○동일 메모리 장소를 다른 変数名으로 사용하는 여러개의 procedure는 그 사용 순서등에 따라 예기치 않은 에러가 발생할 수 있다(Side effect).

○프로그래머는 순수한 문제의 해결 작업에 노력하기 보다 데이타의 불필요한 操作에 시간을 소비한다.

○두개의 독립된 프로세스가 서로 側面 效果(side effect)를 가질때 並列 처리 可能性의 解析이 어렵다.

○프로그램의 評價가 어렵고 다른 언어로의 번역이 쉽지 않다.

2.2 Functional언어

Functional언어는 "現在狀態"라든가 Program Count, Storage등의 概念을 필요로 하지 않는다. 프로그램은 단지 入力 데이타에 대한 기능(function)의 適用이며 適用시킨 자체는 프로그램의 出力으로 取扱된다. 예를 들어 "Plus"가 두 수의 합을 구하는 기능이라고 하면 4+5 계산을 위하여 Plus를 입력 데이타 조합 〈4, 5〉에 適用 시킨다. 즉,

Plus : 〈4, 5〉 → 9

로 표현한다. 계산된 결과는 프로그램(expression)에 사용된 단어인 Plus, 4, 5등에만 관계할뿐 계산의 과정(Computational history)은 중요시 되지 않는다. 또한 프로그램(expression)자체는 그것을 포함하는 expression에 대해 그대로 入力으로 역할하며 이는 反復적으로 사용할 수 있다. 따라서 "Plus : 〈4, 5〉"와 "9"는 동일 대상에 대해 서로 다른 방법으로 표현한 것으로 생각한다.

대표적인 Functional언어들로서 Lisp, Backus의 FP, Hope, Val, Id등을 들 수 있다. Backus의 FP(Functional Programming)는 치환문 형태가 사용되지 않으며 Val과 Id는 단순 置換 언어(Single Assignment Language)라고 표현되기도 한다.

Functional언어 프로그램의 예로서 백타積(matrix multiplication)계산의 경우를 보기로 하자. 우선 백타 內積을

IP = (Reduce Plus) · (Map Times)

로 표시하여

IP : 〈〈1, 2, 3〉, 〈4, 5, 6〉〉
= (Reduce Plus) o, (Map Time) : 〈〈1, 2, 3〉, 〈4, 5, 6〉〉
= (Reduce Plus) : 〈4, 10, 18〉
= 32

로 계산된다. matrix multiplication(MM)은

MM : 〈〈〈1, 2, 3〉, 〈4, 5, 6〉〉, 〈〈−1, −2〉, 〈−3, −4〉, 〈−5, −6〉〉〉
= (Apply ToAll (ApplyToAll IP)) ○ Pair ○ [First, Transpose ○ Second] :
〈〈〈1, 2, 3〉, 〈4, 5, 6〉〉, 〈〈−1, −2〉, 〈−3,

−4⟩, ⟨−5, −6⟩⟩⟩
= (ApplyToAll (ApplyToAll IP)) ○ Pair :
⟨⟨⟨1,2,3⟩, ⟨4,5,6⟩⟩, ⟨⟨−1, −3, −5⟩
⟨−2, −4, −6⟩⟩⟩
= (ApplyToALL (ApplyToAll IP)) : ⟨⟨⟨⟨1,
2,3⟩, ⟨−1, −3, −5⟩⟩, ⟨⟨1,2,3⟩,
⟨−2, −4, −6⟩⟩⟩, ⟨⟨⟨4,5,6⟩, ⟨−1, −3,
−5⟩⟩, ⟨⟨4,5,6⟩, ⟨−2, −4, −6⟩⟩⟩⟩
=⟨⟨−22, −28⟩, ⟨−59, −64⟩⟩

로 계산되어 2 * 3 matrix와 3 * 2 matrix의 적은 2 * 2 matrix를 출력된다. 위의 프로그램은 Imperative언어를 사용할 경우 데이타의 위치 變更, 操作, 保管을 위하여 많은수의 문장(Statement)을 필요로 할뿐 아니라 문제 해결 방법 또한 여러 가지일 수 있다.

2.3 Functional언어의 장점

○계산 体系의 논리성에만 집착하여 큰 문제를 전체적, 포괄적, 体系적으로 표현하므로 그대로 high level language이고 계산상 불필요한 狀況 파악이 제거된다.

○따라서 프로그램 line당 많은양의 Algorithm을 포함하므로 프로그래밍의 生産性이 높다.

○변수(Imperative언어에서)의 access方法 여하에 따라 발생하는 sidecffect, aliasing문제가 제거된다.

○문제 해결을 위한 표현 方法이 응축되어 最善의 方法(optimal expression)을 발견하기 위한 논리성이 정확하고 따라서 프로그램 評價가 수월하다.

○표현의 논리성이 높으므로 並列처리 가능성의 발견을 위한 解析이 용이하다.

2.4 Functional언어의 문제점

○초보자에게는 프로그래밍이 쉽지않다.

○순차적 처리 동작이 불가피한 경우(I/O)등 반드시 기능의 適用이란 概念의 명령으로만 해결될 수 없는 경우가 있다. 따라서 Imperative언어 概念의 명령을 필요로 한다. 이들의 적절한 혼합(hybrid driven)에 대한 硏究가 있어야 할 것이다.

○범용 데이타 처리를 위한 모든 세부적인 작업 또는 hardware cantrol에 functional언어의 처리 概念이 전부 適用될 수 있을 것인지는 이제부터의 硏究 과제이다.

3 Data Flow Graph와 Data Flow 처리 방식

Data Flow Machine의 Low level명령인 기계어 코드 또는 Node Token들은 대부분 Data Flow Graph(DFG)에 근거한다. 그림 1은 n=(3+4)−(1+2) 프로그램을 DFG로 표시한 것이고 화살표를 Arc(입력 또는 출력)라 하며 연산 operater를 포함한 원을 Node라 한다.
그림 1의 연산 프로그램에 대해 노드 토큰을 작성하면 표 1과 같이 표시할 수 있다.

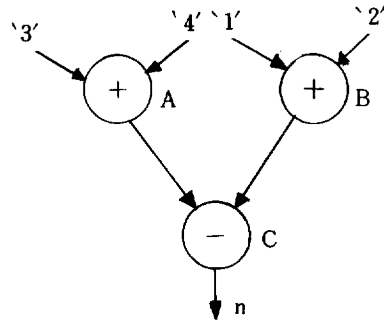

그림 1. n=(3+4)−(1+2)

표 1. Node Token

Token 번 호	토큰위치 (메모리住所)	OP코드	Operand 1	Operand 2	결과를 보낼 목적지	Operand 존재여부	
TokenA	A	ADD	3	4	C의 Left	1	1
TokenB	B	ADD	1	2	C의 Right	1	1
TokenC	C	SUBT	*	**	n	0	0

예를 들면 메모리 번지A의 위치에 TokenA의 Node Token (instruction) 인

ADD 3, 4, C－L

코드를 保管하고 이명령의 Operand인 3, 4 가 모두 존재하므로 Operand존재여부 Flag를 1, 1 로 set하여 둔다. 이후 프로그램 實行 시작은 Operand 존재여부 Flag가 모두 1, 1인 Token 을 fetch하여 並列 연결된 프로세서에서 별도로 동시 처리(연산)하게한다. TokenA는 實行(Fire)된 후 그 결과값을 TokenC의 ＊표 위치에 貯臟하고 Operand존재 여부 Flag중 왼쪽 0을 1로 set시킨다. 따라서 위 프로그램의 경우 TokenA는 프로세서 1에서 TokenB는 프로세서 2에서 동시 처리되고(TokenA와 TokenB는 모두 flag가 1, 1이므로) 각각＊표 위치 및 ＊＊표 위치에 그 결과값을 保管한후 TokenC의 Operand 존재 여부 Flag를 1, 1로 set시킬 것이다.

같은 方法으로 Flag가 1, 1인 TokenC가 fetch되어 實行되고 결과는 n위치에 保管하는 것으로 實行은 끝난다. 이때 fetch된 Token들은 그것의 Operand존재 여부 Flag를 모두 0, 0으로 Reset시켜야 함이 필요하다.

위의 과정에서 Operand존재 與否를 확인하는 동작이 명령 實行 與否를 결정하게 되고 이는 System Clock Pulse가 불필요함을 암시한다. 즉 국부적인 Communication용 Signal은 필요하나 시스템 전체를 관장하는 Clock은 필요치 않다. 이상의 方法은 Data Flow Machine의 대표적 일례이고 토큰의 코딩 方法과 프로세서의 구조등은 지금까지 발표된 7모델들이 서로 다르다. 다음 절에서는 대표적 3모델과 필자등이 제안한 DNL1 모델을 소개한다.

4 Data Flow Machine 모델

4.1 MIT모델 (Dennis' Arvind)

그림 2와 같이 5개의 Unit로 구성되고 다음과 같은 기능을 갖는다.

(1) Activity Storage : Node Token즉 Data Flow 프로그램을 保管하고 Operand도착 與否에 관한 정보를 포함한다.

(2) Update : Operation Unit로부터 계산 결과(result packet)를 받아 packet이 동반한 목적지 住所에 따라 Activity Storage에 결과값을 保管한다. 동시에 Activity Storage내의 Token들 중 Operand가 모두 도착된 Token을 확인하여 그 住所를 Instruction Que로 보낸다.

(3) Instruction Que : Update로부터 Operand가 모두 도착된 Token을 확인하여 그 住所를 Instruction Que로 보낸다.

(3) Instruction Que : Update로부터 Operand가 모든 존재하는 명령(Enabled Token) 즉 實行 가능한 명령의 住所를 받아 순차적으로 Fetch Unit로 보낸다.

(4) Fetch : 實行 가능한 명령을 Activity Storage로부터 읽어 Operation Unit로 보낸다. 이때 해당 명령의 實行 결과를 보낼 목적지 住所도 함께 묶어 (Operation Packet) Update로 보낸다.

(5) Operation Unit : 명령문의 Op코드에 따라 해당 Operater (+, －, ＊, /, AND, OR 등 hardware)에 보내 명령 實行하고 그 결과값을 목적지 住所와 함께 (Result Packet) Update로 보낸다.

이상의 구성 및 동작方法은 다음과 같은 특징를 갖는다.

○데이타의 흐름이 Ring Type프로세서 형태이다. 따라서 Pipelining이 가능하다.

○처리속도는 통로의 갯수와 Pipelining속도에 따라 제한된다.

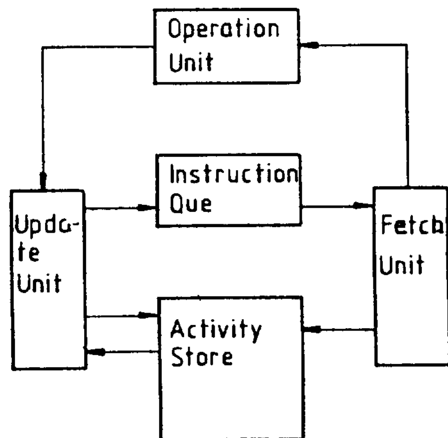

그림 2. MIT모델

○여러개의 packet들이 서로 다른 위치의 통로를 통하여 비동기적으로 전달된다.

○각 Unit들은 2개 이상이 並列 연결되어질 수 있고 이에따라 처리속도를 높일 수 있다.

4.2 Manchester 모델

그림 3과 같은 구조를 갖고 5개 Unit로 구성된다.

(1) Node Storage : Data Flow 프로그램을 保管하고 Matching Unit로부터 Result Packet(Opera-and와 목적지 住所)을 받아 OP 코드와 함께 Processing Unit로 보낸다.

(2) Processing Unit : Node Storage로부터 Instruction Packct을 받아 OP 코드에 따라 實行하고 결과치와 목적지 住所를 Token Que로 보낸다.

(3) Token Que : FIFO Buffer로서 Result Packet의 temporary storage이다.

(4) Matching Storage : Token Que로부터 보내진 결과치들중 동일 목적지 住所를 갖는 packet를 발견(match)하여 Node Storage로 보낸다. partner를 발견하지 못한 packet들은 그것이 나타날때까지 기다린다.

(5) Switch : Host Computer와의 Interface 로서 프로그램 load와 결과치 확인용 Switch이다.

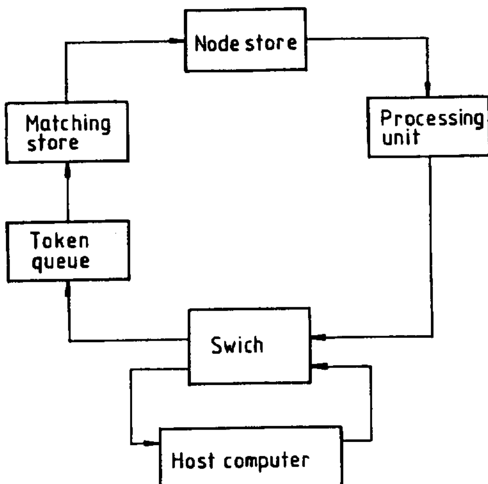

그림 3. Manchester 모델

이상보아 Machester모델은 다음 특징을 지닌다.

○전체적으로 Ring Type 實行 동작으로 MIT 모델과 비슷하다.

○Matching Storage : MIT의 Update 및 Instruction Que의 동작과 비슷한 기능을 갖고 實行 가능한 Enabled Node를 발견한다. 단지 MIT 모델은 結果치를 목적지에 write 한후 Enabled Node를 check하고 Manchester모델은 결과치 write 하기 전에 동일 목적지 住所를 갖는 pair를 발견하도록 한 것이 다르다.

○Manchester모델은 1981년 실제로 製作完了되어 현재 가동중에 있고 Data Flow Machine 중 가장 發展된 것으로 알려져 있다.

4.3 Rumbaugh 모델

그림 4와 같은 構造를 갖고 각 Unit의 기능은 다음과 같다.

(1) Activation Processor(AP) : 각각 1개의 명령을 實行하고 Local Memory를 갖는다.

(2) Scheduler (SC) : AP간의 實行 사이클을 제어 조정한다.

(3) Structure Memory (STM) : Local Memory에 貯臟 불가능한 대형 structured data를 별도 保管하는 메모리이다.

(4) Structure Controller : structured data에 대한 제어 및 조정.

(5) Swap Memory (SWM) : 처리중의 임시 데이타(Intermediate or dormant data)용 메모리

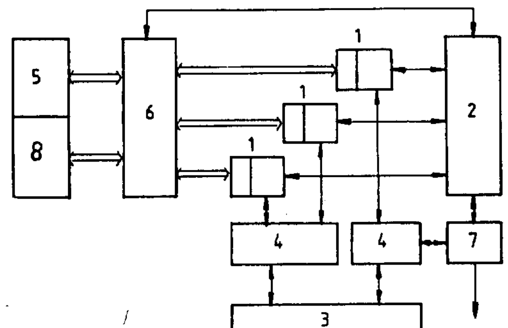

그림 4. Rumbaugh 모델의 프로세서 구조

(6) Swap Network (SWM) : SWM 및 PM과 AP사이의 interface장치

(7) Peripheral Processor (PP) : I/O용 프로세서.

(8) Program Memory (PM) : Data Flow 프로그램을 貯臧한다.

4.4 DNL 1 모델

DNL 1 (Data Flow Machine with Node Labelling No. 1) 모델의 프로세서 구조는 그림 5 와 같고 각 Unit의 技能은 다음과 같다. 그림 5 의 프로세서를 PM (Processing Module) 이라 부르고 PM은 並列 연결에 의해 확장된다.

(1) NTM (Node Token Memory) : op 코드와 1개 또는 2개의 목적지 住所를 保管한다. enabled Node Token은 operand 데이타와 동시에 읽혀져 Proecssing Unit로 보내지고 처리된다.

(2) DTML/R (Data Token Memory Left/Right) : NTM과 동일 住所에 위치한 Node가 사용할 operand데이타를 保管하는 메모리로서 ETF 로 부터 addressing된다.

(3) CF (Control Flag Memory) : control Node가 필요로 하는 제어용 데이타 ("1" 또는 "0") 을 保管하는 1 bit메모리. NTM, DTML/R 및 CF의 내용은 ETF로 부터의 matching signal 에 의해 동일 住所의 내용이 동시에 읽혀져 PU로 보내진다.

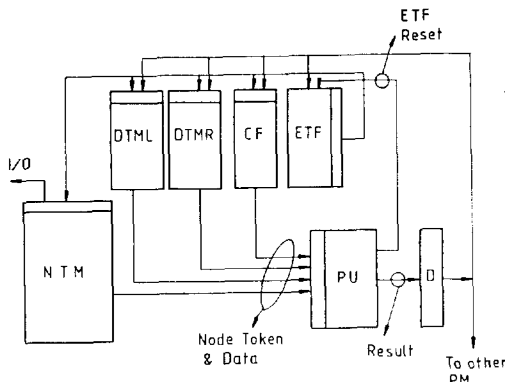

그림 5. DNL1 모델의 프로세서 (PM) 구조

(4) ETF (Enabled Token Flag) : Node의 實行 가능 與否를 표시하는 flag로서 L/R/C값은 각각 DTML/R 및 CF내의 데이타가 도착되었는지의 與否를 나타낸다. Distributor가 임의 PM에서 발생된 Result Token을 해당 목적지 PM의 메모리에 write한후 상대적 동일 위치의 L, R, C 값을 "1"로 set한후 곧 같은 위치의 L/R/C 가 "111"인가를 확인 (read) 한다.

(5) PU (Processing Unit) : Node Token의 op code에 따라 operand 데이타에 대하여 해당 function을 實行하는 unit로서 算術, 논리 및 제어용 operator를 내장한다. 實行된 결과는 Distributor로 보내지고 목적지로 전송된다.

(6) Distributor : PU에서 계산된 결과를 목적지에 따라 각 PM의 DTML/R에 write한다.

5 맺는 말

Functional언어는 프로그래밍의 논리성이 높고 評價 方法이 단순하며 並列 처리성의 解析이 용이하다는 점등 많은 장점을 갖는다. 그러나 현존하는 데이타 처리상의 모든 문제들에 適用되어질 수 있는지는 계속 硏究, 경험되어져야 할 문제이다.

한편 Data Flow Machine은 並列 처리 능력이 높고 pipelining의 適用이 쉬워 고속 처리 가능하며 Functional언어 槪念의 처리 方法에 適合하다는 등 많은 가능성을 갖는다. 그러나 이것또한 실제의 모든 컴퓨터 작업중에 나타나는 세부 사항들의 처리에 전적으로 대신되어 질 수 있겠는지는 이제부터의 硏究 과제이다. 이에 대해 Control Driven, Data Driven, Demand Driven등의 方法들을 서로 복합시켜 "Hybrid-Driven"시스템등으로 발전시켜 나가는 方法도 고려되어 질 수 있다고 본다.

참 고 문 헌

1) Davis, "Computer Architecture", 83/11, IEEE Spectrum, pp. 94-97.

2) Treleaven, "Future computers : Logic, Data Flow,. . . . , Control Flow ?", 84/03, Computer, pp. 47 57.

3) Vegdahl, "A Survey of Proposed Architecture for the Exection", 84/12, IEEE Trans., pp. 1050-1071.
4) Treleven, "Data-Driven and Demand-Driven Computer Architecture", 82/03, Computing Surveys, vol. 14, No. 1.
5) Dennis, "Data Flow Supercomputers", 80/09, Computer, pp. 48-56.
6) Plase et al, "LAU System Architecture : A Parrallel Data Driven Processor Based on Single Assignment", 76/—, IEEE Proc. pp. 293-302.
7) Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", 78/—, ACM Proc. pp. 210-215.
8) Watson, Gurd, "A Practical Data Flow Computer", 82/02, Computer.
9) Rumbaugh, "A Data Flow Multiprocessor", 75/—, IEEE Proc., pp. 220-223.
10) Sowa, Maruta, "A Data Flow Computer Architecture with Program and Token Memories", 82/09, IEEE Trans., vol. 31, pp. 820-824.
11) Dennis, "The Varieties of Data Flow Computer", 79/—, IEEE Proc., pp. 430-439.
12) Hwang, "Computer Architecture and Parallel Processing", MGH, pp. 745-748.

〈P57에서 계속〉

參考文獻

1) 金玄在, "制御入力과 複合機械", 電子工學會誌 第14巻 第15號 pp. 1～4, 4月, 1977年.
2) 金玄在, "複合機械의 入力構造에 關하여", 電子工學會誌, 第18巻 第1號, pp. 1～6, 2月, 1982年.
3) 金玄在, "複合機械像에 依한 디지탈 시스템의 一設計過程", 電子工學會誌 第19巻 第6號. pp 9～16, 12月, 1982年.
4) Taylor L. Booth, "Sequential Machines and Automata Theory". pp. 68～116, John-Wiley and Sons, Inc., NewYork, 1967.
5. Christopher r. Clare, "Designing Logic Systems Using State Machine, pp, 1～5, McGraw-Hill, Book Co., 1973.
6) Martin A編著, 三宅 監譯, 言語學事典(La Linguistique Guide Alphabetique), pp. 242～255, 大修館書店, 東京, 1972.
7) C. E. Shannon and J. McCarthy, Automata Studies, pp. 3～41, Princeton Univ. Press, Princeton, U. S. A., 1956.