CS 58

RECURSIVE FUNCTIONS OF REGULAR EXPRESSIONS IN LANGUAGE ANALYSIS

BY

VINCENT TIXIER

TECHNICAL REPORT NO. 58
MARCH 20, 1967

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

RECURSIVE FUNCTIONS OF REGULAR EXPRESSIONS IN LANGUAGE ANALYSIS

by

Vincent Tixier

March 20, 1967

BLANK PAGE

ACKNOWLEDGEMENTS

My thanks go to Professors Friedman, Wirth, Arbib, Gries and McKeeman for their efforts in reading this paper and for much helpful advice during preparation of the manuscript. I am particularly indebted to my main advisor, Professor Joyce Friedman, for giving her time, science and humor unsparingly; steadily prodding me and exercising patience beyond all normal expectations. I want also to thank Professor Michael Arbib for his highly critical and constructive reading of an earlier draft.

This work was made possible by a scholarship from the Délégation Générale à la Recherche Scientifique et Technique, Comité Calculateurs; I am most grateful toward my two correspondents to the Comité, Professors Carteron and Arsac, and also to Professor Pélegrin for their constant support and encouragement.

The typing and editing of this paper as a departmental report was supported by an Air Force contract under the direction of Professor Friedman.

I want to express my sincere appreciation to Mrs. Phyllis Winkler for her outstanding typing.

I dedicate this work to my wife.

V. T.

TABLE OF CONTENTS

INDEX OF IMPORTANT DEFINITIONS AND NOTATIONS

INTRODUCTION

We discuss first the origins of our work, then we describe its organization in some detail; after that we try to make clear some of the basic ideas which guided us; eventually we shall sketch the background of this paper. In this introduction we do not give references since it will be elaborated upon in the rest of the paper.

Let us examine the origins of this study.

a) One of the central problems of syntax analysis is how to go from a grammar to a recognizer, i.e., from a declarative definition of a language to an analytic one, from extension to comprehension. When the process can be precisely described, it is possible to specify it to a computer and devise what is called a compiler compiler or meta-compiler. The problem is complicated by the further requirement that the analyzers generated by a meta-compiler be comparable in speed and economy to those written by hand, using heuristics.

In this respect, the class of context-free grammars has appeared to be an unsatisfactory metalinguistic tool because it is both too wide and too narrow: wide enough to define extremely baroque sets, so that its mathematical properties are complex and its handling inefficient; too narrow to permit the expression of many important well-formedness conditions in actual programming languages. Furthermore, the meta-syntactical language it offers is somewhat poor; this being a matter of convenience, rather than power.

Not waiting for the theory to catch up with the needs, programmers have developed a few highly successful compiler compilers from more or

less precisely defined restrictions of context-free languages.

In the study of these, in particular at a seminar organized by Dana Scott, we became convinced that Kleene's regular expressions played a significant role in that field, both because they were used implicitly and because some constructs causing difficulties could be described by regular expressions.

At the same seminar we noticed how little is known about the elementary transformations of context-free grammars which leave a language invariant. This is important because most parsing algorithms can work only when the grammar has a given form or given properties. Unfortunately, we have been able to show that no complete axiom system for the equality of context-free grammars can be constructed. This came as a serious blow to our initial hopes that the field of syntax could be open to the axiomatic method, as the desire had been expressed by Church in his Introduction to Mathematical Logic (Section 08). But we may after all remark that the situation is no more comfortable in the case of arithmetic.

Note the following technicality: we often wanted to use substitution of equals for equals without taking each time all sorts of precautions; a basic decision was then to introduce a set union symbol, for which the $+$ of regular expressions was adopted rather than the $|$ of BNF, and replace grammars by systems of equations where one equation $A = \alpha_1+\ldots+\alpha_n$ corresponds to the n rules of the grammar having A on the left side: $A \to \alpha_1,\ldots,A \to \alpha_n$. This may seem a minor technicality, but it forced us to revise a number of notions of syntax analysis.

One of the fastest and simplest analysis methods used by compiler compilers is the one character look ahead top-down scheme of Schorre's Meta series; hardly any theoretical results were known about its scope and power; it makes use of regular expressions, more or less explicitly, with the result that its notation is very convenient.

A last observation we made was that symbols for complementation and intersection are quite convenient to describe regular sets for circuit design and there is no theoretical reason why they could not be used in defining artificial languages.

Let us now give a summary of our paper.

b) After specifying the notations and recalling the basic notions of the algebra of strings, we start in Section 3 with the study of regular expressions using all the Boolean connectives. Continuing Salomaa's and Aanderaa's work, we give an axiom system for these expressions and prove its completeness. This proof is centered around systems of equations of precisely a form we are interested in; its by-products are a simple theorem on the equality of regular expressions and new constructive proofs of some old theorems on finite state automata.

In Section 4 we study the analysis problem, in particular for regular sets and we show that the difficulties encountered in applying certain analysis methods correspond to a known automaton-theoretic notion.

Considering now the most natural method for analyzing regular sets, i.e., by finite state functions, we ask (Section 5) whether they can be used recursively to analyze without backtracking some context-free

languages. We note that a generalization of Algol 60, Euler, is in the scope of this extremely fast method as far as its context-free syntax is concerned. We define the notion of separability of two sets of strings and, by applying it together with the tools developed in Section 3, we formalize this approach to syntax analysis and define a class of context-free languages which we call regular context-free (RCF). We give alternate characterizations of this class, one of which is automaton-theoretic, and we relate them to other recently defined classes, the s-languages and the languages defined by for grammars. We study some of the usual unsolvability and closure questions and some unusual cancellation properties linked with the notion of separability. In particular, we examine a semi-decision procedure by which we can show that no complete axiom system for the equality of context-free grammars can exist.

Examining in Section 6 how this model fits programming languages, we conclude that it is necessary to extend it in a direction going outside the class of context-free languages. We briefly study the problems linked to the introduction of symbols for intersection and complementation in the metasyntactical language; then we introduce conditional regular expressions and lay axiomatic foundations for their algebra; we submit that to use recursive functions of regular expressions, just as recursive regular expressions are used in RCF languages, will essentially be to do in a formal and well understood fashion what is already done more or less formally in various ways, in particular when people confuse syntax and semantics. We conclude by remarking that the scope of syntactical analysis is presently underestimated and by indicating some avenues for further studies.

In the appendices, we have put some material which we felt was not in the main stream of our development, although a large proportion of it is new.

c) Permeating our work are some basic attitudes toward programming theory and practice. Let us try to make them clear in order to open them more readily to discussion.

i) We would rather nowadays see a programming language defined by its recognizer written in Algol 60 or Lisp than by metasyntactical constructs from which nobody knows how to get a recognizer; if the metasyntactical language is furthermore unreadable, the whole exercise makes little sense to us. In other words we think that the justifications for metasyntactical descriptions are not just rigour and formality, but, as important, readability and translatability into a recognition algorithm. In fact, we want to see a metasyntactical description as specifying both the syntax and the recognizer, i.e., the component sets of strings and the relations between their characteristic functions; so that the declarative definition of the language is analytic at the same time.

This idea is as old as metasyntactical definition, but the early difficulties with the use of unrestricted context-free grammars have made it fall largely into oblivion.

ii) A computer being universal, automaton-theoretic characterizations of sets of strings are to be understood as measures of their computational complexity and not as programming strategies. This has always been

clear to most theoreticians; some programmers have been misled and it has cost them a high price in loss of efficiency, chiefly when non-deterministic automata and backtracking algorithms were involved.

iii) The basic language and notations are a very essential part in a research field. But naturally it is very hard to choose them because a priori we do not know where we shall go, gropingly, building and testing models with them; we must have recourse to our intuition of the nature of the field. This paramount role of notations and language is most apparent when one thinks about those many famous combinatorial problems, sometimes quite puzzling, which have appeared as solvable by trivial computations when expressed in graph theory. Our intuition is that the terminology of computational linguistics should adopt a number of well-established graph theoretical notions, that the notion of derivative of a set of strings with respect to a set of strings provides a natural link between computational linguistics and automata theory and should likely be made central to the former, that conditional forms are a natural tool of computer science and should be used systematically in this discipline.

d) The background of our work is naturally that part of computer science which deals with the more theoretical aspects of programming and in particular of compilation. We make use of the basic terminology and notions of such closely interrelated disciplines as computational linguistics, automata theory, recursive function theory and symbolic logic; as we have seen, a large part of this paper is relevant to the theories of regular sets of strings (or regular events) and of context-

free languages. We also use some very elementary terminology of graph theory and Algol 60.

Last, we may emphasize that this is not merely a theoretical paper but that constant attention is paid to the practical aspects of implementation, as can be expected from a work in computer science.

SECTION 1

NOTATIONS AND CONVENTIONS

T — The alphabet or vocabulary. A finite set of symbols called terminal symbols or letters. denoted by $a,b,\ldots,a_1,b_1,\ldots$.

T^* — The free monoid with cancellation generated by T . The non-commutative operation called concatenation is denoted by juxtaposition. The elements of T^* , denoted by $\alpha,\beta,\ldots,\alpha_1,\beta_1,\ldots,$ are finite strings of letters; the unit, called null string is denoted by λ .

Θ — $\Theta = 2^{T^*}$, the set of all subsets of T^* . Its elements are denoted by $A,B,\ldots,A_1,B_1,\ldots$; the empty set is denoted by $\emptyset$. Θ is a Boolean ring with unit T^* and zero $\emptyset$.

$=$ — The equality sign will be considered as part of the syntax language and substitution will not be mentioned as a rule of inference. In subalgebras of Θ we will consider

S — systems S of equations, always of the form $X_i = f_i(X_1,\ldots,X_n)$ $i = 1,\ldots,n$, where X_i is a variable.

I_s — $I_s = \{X_i | i = 1,\ldots,n\}$, set of variables of S , called also metavariables or intermediate symbols (I when S is understood). The following relations are defined in I_s :

$X_i > X_j$ — if X_j appears in f_i (Read: "depends directly on");

$X_i >^* X_j$ for the transitive closure of $>$ (Read: "depends on"), $>^*$ is a relation of order;

$X_i \# X_j$ $\Leftrightarrow X_i >^* X_j$ and $X_j >^* X_i$ (Read: "depends recursively on"), $\#$ is an equivalence relation.

The dependency graph of S is the finite directed graph $\langle I_s; > \rangle$, where the arc (X_i, X_j) is oriented from X_i to X_j if $X_i > X_j$. S can be represented as a labeled graph, G_s, obtained by labeling each arc (X_i, X_j) of $\langle I_s; > \rangle$ by f_i. We shall speak of S as of G_s, without making the distinction. We shall use the basic terminology of graph theory as defined in Berge [1958/1962].

Ω Set of positive integers and zero.

card card(X) where X is a set, denotes its cardinality.

if...then...else... We shall freely make use of conditional expressions formed with this ternary operator. For a formal introduction see McCarthy [1963].

The words "set" and "language" will be used indifferently for sets of strings.

$\vdash$ "It is provable (in some understood logical system) that".

$\models$ "It is true (in some understood interpretation) that".

SECTION 2

STRINGS

The properties of T^* are well-known; an axiomatic definition, closely resembling Peano's axiom system for integers is given in Appendix 1.

Two important relations are

(i) the refinement rule: $\alpha\beta = \gamma\delta \Rightarrow (\exists\xi)[\alpha\xi = \gamma \vee \xi\beta = \delta]$

(ii) the left cancellation rule: $\alpha\beta = \alpha\gamma \Rightarrow \beta = \gamma$.

In what follows we shall always assume that strings are uniquely readable, because we do not want any "coding problem" at this level and because we are interested in models of situations where this is the case.

In the Linear Lisp fashion (McCarthy [1960]) two unary operations are defined in T^* :

first(α) yielding the first letter of α from left to right, first(λ) is undefined.

rest(α) yielding what remains of α when first(α) has been deleted. rest(λ) is undefined. rest(a) = λ .

A more formal definition is given in Appendix 1.

first and rest are extended to sets of strings:

$$\text{first}(A) = \{a \mid \gamma \in A \wedge \text{first}(\gamma) = a\} \ ,$$

$$\text{rest}(A) = \{\alpha \mid \gamma \in A \wedge \text{rest}(\gamma) = \alpha\} \ .$$

The length of a string α , denoted by $|\alpha|$, is defined by:

$$|\alpha| = \text{if } \alpha = \lambda \text{ then } 0 \text{ else } 1 + |\text{rest}(\alpha)| \ .$$

The reverse of a string α, denoted α^R, is defined by:

$$\alpha^R = \text{if } \alpha = \lambda \text{ then } \lambda \text{ else } (\text{rest}(\alpha))^R \text{first}(\alpha) \ .$$

The n-fold concatenation of a string with itself, denoted α^n, is defined by:

$$\alpha^n = \text{if } n = 0 \text{ then } \lambda \text{ else } \alpha^{n-1}\alpha \ .$$

As shown in Appendix 1:

$$|\alpha\beta| = |\alpha| + |\beta| \quad \alpha^n\alpha^p = \alpha^{n+p} \quad \alpha^{RR} = \alpha \quad (\alpha\beta)^R = \beta^R\alpha^R \ .$$

Note that, as proved by McCarthy (unpublished), any computable function on strings is representable by a system of recursive functions of conditional expressions formed with the two operators first and rest. (The proof is by showing the equivalence to the Turing machine formalism. A notational difference is that the equality does not belong to the syntax but corresponds to a predicate $\text{eq}[\alpha,\beta]$.)

SECTION 3

REGULAR EXPRESSIONS

a) Formal Definition and Interpretation

The set $\mathfrak{R}$ of regular expressions is defined as follows (Kleene [1951]):

(i) $\emptyset \in \mathfrak{R}$

$\lambda \in \mathfrak{R}$

Any symbol denoting an element of T is in $\mathfrak{R}$.

(ii) if $P \in \mathfrak{R}$ and $Q \in \mathfrak{R}$ then

$$(P) \in \mathfrak{R}, \quad P + Q \in \mathfrak{R}, \quad P \cdot Q \in \mathfrak{R}$$
$$P \& Q \in \mathfrak{R} \quad P^{*} \in \mathfrak{R}$$
$$P' \in \mathfrak{R}$$

(iii) Extremal clause: $P \in \mathfrak{R}$ only if P can be formed by a finite number of applications of rules (i) and (ii).

$\mathfrak{R}$ is context-free (Appendix 2), thus recursive.

The dot in $P \cdot Q$ is customarily omitted.

To interpret regular expressions, the structure of the Boolean ring $\mathfrak{B}$ is enriched as follows:

(1) a monoid structure is introduced by

$$P \cdot Q = \{\alpha\beta \mid (\alpha \in P) \wedge (\beta \in Q)\} .$$

Note that

$$P \cdot \{\lambda\} = \{\lambda\} \cdot P = P$$

$$P \cdot \emptyset = \emptyset \cdot P = \emptyset$$

$$P \cdot (Q \cdot R) = (P \cdot Q) \cdot R$$

but we do not have cancellation or refinement.

This operation is called <u>concatenation</u> or <u>generalized product</u>. We define P^n by

$$P^n = \text{if } n = 0 \text{ then } \{\lambda\} \text{ else } P \cdot P^{n-1} \quad .$$

(2) To a set $P \in \Theta$ we associate the free monoid generated by its elements:

$$P^* = \lim_{n \to \infty}(P^0 \cup P^1 \cup \ldots \cup P^n) \quad .$$

Note that this notation is coherent with the definition of T^* from T in Section 1.

This operation is called <u>star</u> or <u>closure</u> and sometimes denoted $cl(P)$. It can be defined externally by

$$P^* = \bigcap_{X \in \Theta} (X \mid (\lambda \in X) \wedge (P \cdot X \subseteq X)) \quad .$$

Regular expressions are interpreted recursively as sets of strings called regular sets, according to the following mapping:

$$\text{value: } \mathfrak{R} \to \Theta$$

such that

Regular expression R	value(R)
$\emptyset$	the empty set
λ	$\{\lambda\}$
a,b,...	$\{a\},\{b\},\ldots$
(P)	value(P)
P + Q	value(P) $\cup$ value(Q)
P & Q	value(P) $\cap$ value(Q)
P'	complement(value(P))
PQ	value(P) $\cdot$ value(Q)
P^*	$(\text{value}(P))^*$

Conflicts of interpretation are resolved by evaluating + , & , · , ' , and * in that order of increasing priority, parentheses being used as usual.

Regular expressions denoting a unit set are usually called by the name of the element of that set.

When the symbols & and ' are not used, we talk of <u>restricted</u> regular expressions.

Equality: $P \in \mathcal{R}$, $Q \in \mathcal{R}$

$P = Q \Leftrightarrow \text{value}(P) = \text{value}(Q)$.

The problem of recognizing the equality of regular expressions was first solved in Friedman [1957], and Moore [1956]. To devise insightful and computationally efficient algorithms for this recognition is one of the main topics of the theory of regular sets (see McNaughton [1965]). It is not an academic problem:

Regular sets are those sets of strings which can be recognized without memory, or, equivalently, with a bounded amount of memory, i.e., by a finite state automaton (Kleene [1951]). Using the black box approach and the definition of states by the Nerode equivalence relation, a simple argument (Moore [1956]) shows that any solution to the equality problem yields a solution to the practically important minimization problem.

A number of important constructs in high-level programming languages correspond to regular sets, we will discuss this in detail in Section 5.

b) Axiom System for Regular Expressions

Axiom systems have been constructed for restricted regular expressions by Aanderaa [1965] and Salomaa [1966].

We submit the system of schemata RE , for unrestricted regular expressions, and 2 rules of inference.

RE.

(b1) $\vdash A + B = B + A$

(b2) $\vdash A \,\&\, B = B \,\&\, A$

(b3) $\vdash A + B \,\&\, C = (A+B) \,\&\, (A+C)$

(b4) $\vdash A \,\&\, (B+C) = A \,\&\, B + A \,\&\, C$

(b5) $\vdash A + \emptyset = A$

(b6) $\vdash A \,\&\, (T)^* = A$

(b7) $\vdash A + A' = (T)^*$

(b8) $\vdash A \,\&\, A' = \emptyset$

(g1) $\vdash A(BC) = (AB)C$

(g2) $\vdash A\lambda = A$

(g3) $\vdash A\emptyset = \emptyset$

(s1) $\vdash A^* = \lambda + AA^*$

(s2) $\vdash A^* = (\lambda+A)^*$

(i1) $\vdash A(B\&C) = AB \,\&\, AC$

(i2) $\vdash (B\&C)A = BA \,\&\, CA$

(i3) $\vdash \lambda \,\&\, xA = \emptyset$

Rules of inference:

(R1) $$\frac{\vdash \lambda \,\&\, B = \emptyset \ , \ \vdash A = BA + C}{\vdash A = B^{*}C}$$

(R2) $$\frac{\vdash x \neq y}{\vdash xA \,\&\, yB = \emptyset}$$

Remarks: (i) Rules (b1) to (b8) define a Boolean algebra; in effect they are the Whitehead system as modified by Huntington (Section 1 in Huntington [1904]). For a discussion of it and others see Rudeanu [1963]. We shall not specify the derivation of usual Boolean relations, the derivations of associativity of + and & and of a few useful relations are given in Appendix 3.

(ii) The notation adopted is clearly redundant: as proved in Appendix 3 $\lambda = \emptyset^{*}$, $A \,\&\, B = (A' + B')'$; we are not interested in minimality.

(iii) None of these rules refers specifically to regular sets, except the non-written ones: the "zero axioms" which are the formal definition of regular expressions. The RE system specifies the operators + , & , ' , · , and * in Θ ; note that when defining them in a), we did not suppose that they were applied to regular sets. We can freely use these rules to transform systems of equations in Θ into other systems having the same solution and of a more desirable form.

(iv) At least one rule of inference is needed besides substitution of equals for equals which we consider here as a syntactic rule (Redko [1964]). Note that (R1) corresponds to the external definition of star.

(v) (R1) contains a right-recursive rule; the system obtained with a left-recursive one and corresponding modifications in rules (R2), (g2), (g3), (s1), and (i3) is equivalent. We shall use right recursion because it corresponds to left-to-right string synthesis. The results and proofs can be reformulated in terms of left-recursion.

(vi) It is interesting to compare RE to the set of formulas in McNaughton and Yamada [1960], Ghiron [1962] and the axiom systems in Aanderaa [1965] and Salomaa [1966]. All are interested in restricted regular expressions. McNaughton and Yamada have all of Salomaa's rules except the ones which deal with * , although they could derive (s2); they do not have the rule of inference (R1) and the Boolean relation A + A = A cannot be obtained from what they have. Ghiron introduces rules for * , including R1, which had been introduced independently by Arden [1961]; (s1) and (s2) are derivable from his rules. Aanderaa and Salomaa's works are not quite independent and both contain complete systems. Salomaa gives two systems, F_1 and F_2 ; F_2 corresponds to a different approach; F_1 is essentially the same as Aanderaa's, but simplified; the rules of inference are R1 and a rule of substitution of equals for equals; the Boolean algebra part is restricted to 4 rules necessary to define + . The introduction of & and ' essentially forces a complete set of Boolean relations, rule (i3) and rule of inference (R2).

An earlier paper by Salomaa is discussed in Aanderaa's paper. It contained what was proved by Aanderaa to be a complete system.

Theorem 3.1: The axiom system RE with rules of inference R1 and R2 is sound with respect to the given interpretation.

All axioms are valid and R1 and R2 preserve validity. ∎

We now want to prove the completeness of the system. The proof will follow the lines of Salomaa's proof; its by-products will be as important and useful as the final result itself. First we consider a particular type of system of equations in the Kleene algebra

$$\langle \mathfrak{B}; +, \&, ', \cdot, *; RE, R1, R2 \rangle \ .$$

c) SRL Systems. Equational Characterization.

Definition 3.2: $\delta(X) = \lambda \ \& \ X$.

Note that since $value(\lambda \ \& \ X) = \{\lambda\} \cap value(X)$,

$$\delta(X) = \text{if } \lambda \in value(X) \text{ then } \lambda \text{ else } \emptyset \ .$$

Definition 3.3: (i) An equation is standard right linear (SRL) when it is of the form $A = \sum_{x \in T} xA_x + \delta(X)$. Where the A_x 's are variables.

(ii) A system of equations is SRL when all of its equations are SRL and it has one equation per variable.

Note that in $\sum_{x \in T} xA_x$, all the x 's in T do occur. (Compare

$L(F_4)$ in Chomsky and Miller [1958]). Naturally some Ax 's can be equal to $\emptyset$.

Note also that in an SRL equation $\delta(X)$ stands for the value of $\delta(X)$, λ or $\emptyset$, and not for the function $\delta(X)$. (See example further.)

Definition 3.4: (i) The root A_1 of a system is a distinguished variable.

(ii) To solve a system is to express A_1 as an expression in the algebra of its coefficients and constant terms, such that the equations are satisfied.

Definition 3.5: A regular expression R is equationally characterized when there is an SRL system which has a solution equal to R .

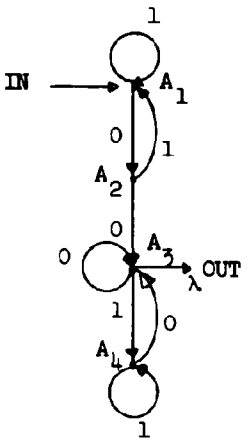
Note that the graph of an SRL system can be considered as the transition graph of a deterministic finite state automaton and conversely. (These graphs are introduced and studied in McNaughton and Yamada [1960] and Brzozowski and McCluskey [1963]).

Example: Let $T = \{0,1\}$. Consider the regular set R of all strings in T which contain two consecutive 0 's and are not terminated by a 1 :

$$R = (T^*00T^*) \,\&\, (T^*1)' \quad .$$

It can be proved, using techniques we are going to develop in this paragraph, that $R = A_1$ where A_1 is defined by the following SRL system corresponding to the following graph:

$$A_1 = 0A_2 + 1A_1$$
$$A_2 = 0A_3 + 1A_1$$
$$A_3 = 0A_3 + 1A_4 + \lambda$$
$$A_4 = 0A_3 + 1A_4$$



Here we have as usual labelled the arrows 0 or 1 rather than by the full function $0A_2 + 1A_1$ for instance.

The following lemma is due to Salomaa (Lemma 2, page 161) and is proved by induction, using (R1):

Lemma 3.6: If

$$\vdash A_i = \sum_{j=1}^{n} R_{ij}A_j + R_i \qquad i = 1,\ldots,n \qquad \text{and}$$

$$\vdash B_i = \sum_{j=1}^{n} R_{ij}B_j + R_i \qquad i = 1,\ldots,n \quad R_{ij} \text{ some regular expression}$$

where $\delta(R_{ij}) = \emptyset$ for all (i,j) then

$$\vdash A_i = B_i \qquad i = 1,\ldots,n \quad .$$

Lemma 3.7: Any SRL system has a unique solution.

Proof: Any SRL system has a solution: when an equation is not recursive one can substitute for the variables the quantities which define them;

when an equation is recursive, rule (i3) proves that rule (R1) can be applied. Note that the result will be a restricted regular expression.

The solution is unique: Let us reduce the general case to a form where Lemma 3.6 can be applied. We have a system of n equations

$$A_i = \sum_{x \in T} xA_{ix} + \delta(A_i) \qquad i = 1,\ldots,n \quad .$$

In $\sum_{x \in T} xA_{ix}$ we can group the terms corresponding to a given A_j into a term where A_j is factored out: $P_{ij}A_j$ (rule (i1), modified by de Morgan's law into a rule $A(B+C) = AB+AC$.); if in the sum of $P_{ij}A_j$ terms which we obtain, a variable A_p of the system does not occur, we can add a term $\emptyset A_p$ (in Appendix 3 we show how by (g3), (b5), and (R2) $\emptyset A_p = \emptyset$, by (b5) $A+\emptyset = A$).

We now have a system of the form

$$A_i = \sum_{j=1}^{n} P_{ij}A_j + \delta(A_i) \qquad i = 1,\ldots,n$$

and $\delta(P_{ij}) = \emptyset$ by (i3) or the Boolean rule $\lambda \& \emptyset = \emptyset$. By Lemma 3.6 the solution is unique.

This proves the lemma. Note that furthermore we can assert that if two systems have exactly the same form but the variables having different names, then not only are the roots equal, but also all the variables are equal two by two. ∎

This lemma is a direct proof in the particular case of SRL systems of a lattice-theoretical fixpoint theorem of Tarski, which can be applied to context-free grammars as shown by Ginsburg and Rice [1962] (a simpler but similar proof for context-free grammars is given in

Letichevskii [1965]). The constructive proof we can give in this simple case is not a particular case of their proof.

Corollary 3.8: (Cancellation of strings)

$$\vdash \alpha A = \alpha B \Rightarrow \vdash A = B \quad .$$

The proof is by induction on the length of α, since as we have seen at the end of the proof of Lemma 3.7,

$$\vdash xA = xB \Rightarrow \vdash A = B \quad . \qquad \blacksquare$$

Lemma 3.9: Every regular expression is equationally characterized.

This is Salomaa's Lemma 4, but regular expressions are unrestricted here. Let us briefly recall his proof and complete it for the & and ' operators.

The proof follows the recursive definition of regular expressions.

(i) $\vdash \emptyset = N$ where $N = \sum_{x \in T} xN$ by Lemma 3.7, (g3) and (b5).

$\vdash \lambda = B$ where $B = \sum_{x \in T} xN + \lambda$

$$N = \sum_{x \in T} xN$$

(the above, (b5)).

$$\vdash a = A \quad \text{where} \quad A = \sum_{\substack{x\in T \\ x\neq a}} xN + aB$$

$$B = \sum_{x\in T} xN + \lambda$$

$$N = \sum_{x\in T} xN$$

(the above, (b5))

(ii) Suppose A and B are equationally characterized by SRL systems S_A and S_B, i.e.,

$$\vdash A = A_1 \quad \text{where} \quad A_i = \sum_{x\in T} xA_{ix} + \delta(A_i) \qquad i = 1,\ldots,n \ .$$
$$A_{ix} \in I_{S_A} \ .$$

$$\vdash B = B_1 \quad \text{where} \quad B_j = \sum_{x\in T} xB_{jx} + \delta(B_j) \qquad j = 1,\ldots,m \ .$$
$$B_{jx} \in I_{S_B} \ .$$

Let us prove that $A+B$, $A\&B$, A', AB and A^* are equationally characterized.

(a) $A+B$ is equationally characterized.

Let the system

$$D_{kl} = \sum_{x\in T} xD_{klx} + \delta(D_{kl}) \qquad k = 1,\ldots,n \qquad l = 1,\ldots,m$$

be obtained as follows:

$$\vdash A + B = \sum_{x\in T} x(A_{1x} + B_{1x}) + \delta(A_1) + \delta(B_1)$$

((i1) modified by de Morgan's law, plus Boolean relations.)

Set $D_{11} = A_1 + B_1$ and generally $D_{kl} = A_k + B_l$ (or think of D_{kl} as representing the symbol "$A_k + B_l$"); we have here

$$\vdash A + B = D_{11}$$

where

$$\vdash D_{11} = \sum_{x \in T} xD_{1x1x} + \delta(D_{11})$$

since

$$\vdash \lambda \,\&\, A_1 + \lambda \,\&\, B_1 = \lambda \,\&\, (A_1 + B_1)$$

We may have here a number of D_{ij} different from D_{11} ; repeat the process with them as was done for D_{11} until no new D_{ij} appears.

Note that the method is well adapted to computer implementation, using an m by n array to keep track of the appearance of new D_{kl} 's.

(b) $A \,\&\, B$ is equationally characterized.

The proof is quite similar to the one for $A + B$,

$\vdash A \,\&\, B = A_1 \,\&\, B_1$

$$A_1 \,\&\, B_1 = \sum_{\substack{x \in T \\ y \in T}} xA_{1x} \,\&\, yB_{1y} + \sum_{x \in T} xA_{1x} \,\&\, \delta(B_1) +$$

$$\sum_{x \in T} \delta(A_1) \,\&\, xB_{1x} + \delta(A_1) \,\&\, \delta(B_1)$$

(by Boolean properties.)

By (R2), (i1), (i3) and some Boolean properties:

$$A_1 \,\&\, B_1 = \sum_{x \in T} x(A_{1x} \,\&\, B_{1x}) + \delta(A_1 \,\&\, B_1)$$

if D_{ij} denotes A_i & B_j

$$\vdash A \,\&\, B = D_{11} \quad \text{where} \quad D_{11} = \sum_{x \in T} xD_{1x1x} + \delta(D_{11}) \ .$$

The proof terminates as for A+B .

Note a simplification: if A_i or $B_j = N = \emptyset$, $D_{ij} = N$, it is not necessary to develop spurious equations having $\emptyset$ as solution.

(c) A' is equationally characterized.

Consider the system $D_i = \sum_{x \in T} xD_{ix} + \delta(D_i)$ $i = 1,\ldots,n$ obtained from S_A by replacing A_i by D_i throughout and replacing $\delta(A_i)$ by $\emptyset$ if $\delta(A_i) = \lambda$ and by λ if $\delta(A_i) = \emptyset$. This system S_D has a solution $D = D_1$.

We form A+D and A&D as just described.

A&D is equationally characterized by an SRL system in which no equation contains λ as its last term; A+D by an SRL system where all equations do contain λ .

$\emptyset$ is clearly a solution of the system characterizing A&D and T^* a solution of the system characterizing A+D .

By Lemma 3.7 these solutions are unique.

Thus $\vdash A' = D$, since it is provable that in a Boolean algebra the inverse is unique (see Appendix 3).

Note that $\vdash (A_i)' = D_i$, $i = 1,\ldots,n$.

Constructs quite similar to the one for A' can be found in Chomsky and Miller [1958] and for A&B and A' in McNaughton and Yamada [1960]; these constructs are developed on the corresponding labeled graphs.

(d) AB is equationally characterized.

We proceed in the same way by proving that $\vdash AB = D_{1\langle 1,0,\ldots,0\rangle}$ where

$$D_{jk} = AB_j + \sum_{p\epsilon k} A_p \qquad j = 1,\ldots,n$$

$$k = \langle k_1, k_2, \ldots, k_p\rangle \text{ with } k_j = 0 \text{ or } j .$$

and where $D_{1\langle 1,0,\ldots,0\rangle}$ is the solution of an SRL system.

$$\vdash AB = A_1B_1 = D_{1\langle 1,0,\ldots,0\rangle} .$$

Let us form A_1B_1 to show that $D_{1\langle 1,0,\ldots,0\rangle}$ is the solution of an SRL system:

$$A_1B_1 = \sum_{x\epsilon T} xA_{1x}B_1 + \delta(A_1)\left(\sum_{x\epsilon T} xB_{1x} + \delta(B_1)\right) .$$

Two cases:

-- if $\delta(A_1) = \emptyset$ we see that $\delta(A_1B_1) = \emptyset$ (definition of δ and (i3)) and

$$D_{1\langle 1,0,\ldots,0\rangle} = \sum_{x\epsilon T} xD_{1x\langle 1,0,\ldots,0\rangle} + \delta(D_{1\langle 1,0,\ldots,0\rangle})$$

-- if $\delta(A_1) = \lambda$

$$D_{1\langle 1,0,\ldots,0\rangle} = \sum_{x\epsilon T} xD_{1x\langle 1,0,\ldots,1x,\ldots,0\rangle} + \delta(D_{1\langle 1,0,\ldots 0\rangle})$$

where $\delta(D_{i\langle 1,0,\ldots,0\rangle}) = \delta(B_1)$, hereby.

As for A+B and A&B we can keep generating D_{jk} 's until no new term appears.

In machine implementation it is convenient to represent the subscript k by a binary number between 0 and 2^n-1 .

(e) A^* is equationally characterized.

Let k be defined as above. $\vdash A^* = D_0$ where $D_k = (\sum_{p\epsilon k} A_p)A^*$ and $D_0 = A^*$ by convention.

By (s2) or (b5) and (s1) $\vdash D_0 = \sum_{x\epsilon T} xA_{1x}A_1^* + \lambda$, thus

$$\vdash D_0 = \sum_{x\epsilon T} xD_{<0,\ldots,1x,0,\ldots,0>} + \delta(D_0)$$

since by (s1), (i3), and the Boolean relation $\lambda\&\lambda = \lambda$, $\delta(D_0) = \lambda$.

And we can proceed forming D_k terms until no new term is necessary. ∎

Corollary 3.10: Any unrestricted regular expression is equal to a restricted one.

Proof: We have seen in the proof of Lemma 3.7 that an SRL system can always be solved and that the solution is then expressed by a restricted regular expression. ∎

d) Main Property

Let Σ be the class of SRL systems, what we have done in the proof of Lemma 3.9 is to associate to each regular expression an element of Σ , to each operation in $\mathcal{R}$ a corresponding operation in Σ ; let us denote these operations in Σ by $+$, $\&$, $'$, $\cdot$ and $*$, as are denoted the operations in $\mathcal{R}$ they correspond to. Let us define equality in Σ by:

Definition 3.11: $S_A = S_B$ if and only if $A = B$.

With these conventions, if φ is the mapping of R into Σ we have defined, then the two following diagrams commute:

$$\begin{array}{ccc} (A,B) & \xrightarrow{\varphi} & (S_A,S_B) \\ \downarrow & & \downarrow \\ A \odot B & \xrightarrow{\varphi} & S_A \odot S_B \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{\varphi} & S_A \\ \downarrow & & \downarrow \\ A \otimes & \xrightarrow{\varphi} & S_A \otimes \end{array}$$

where $\odot$ stands for $=$, $\neq$, $+$, $\&$, or $\cdot$ and $\otimes$ stands for or $*$.

Let the symbol $\equiv$ denote in Σ the identity of SRL systems up to renaming of variables.

The interest of Σ as a representation of regular sets stems from the following exceptional property:

<u>Theorem</u> 3.12: $\models S_A = S_B \Leftrightarrow \vdash S_A + S_B \equiv S_A \,\&\, S_B$.

Proof: (i) $\models S_A = S_B \Rightarrow \vdash S_A + S_B \equiv S_A \,\&\, S_B$.

Hypothesis: S_A is $A_i = \sum_{x \in T} xA_{ix} + \delta(A_i)$ $\quad i = 1,\ldots,n$

S_B is $B_j = \sum_{x \in T} xB_{jx} + \delta(B_j)$ $\quad j = 1,\ldots,m$.

By 3.11:

$$\models S_A = S_B \Rightarrow \models A = B \Rightarrow \models A_1 = B_1 \quad .$$

Form A_1+B_1 and $A_1 \,\&\, B_1$:

$$A_1+B_1 = \sum_{x \in T} x(A_{1x} + B_{1x}) + \delta(A_1 + B_1)$$

$$A_1 \,\&\, B_1 = \sum_{x \in T} x(A_{1x} \,\&\, B_{1x}) + \delta(A_1 \,\&\, B_1)$$

$\models A_1 = B_1 \Rightarrow \models A_1 + B_1 = A_1 \,\&\, B_1 \Rightarrow \models \delta(A_1 + B_1) = \delta(A_1 \,\&\, B_1)$.

Furthermore, it is not possible that $A_{1x} \neq \emptyset$ while $B_{1x} = \emptyset$ since $\models A_1 = B_1 \Rightarrow \models A_1 \,\&\, xT^* = B_1 \,\&\, xT^*$. We see that if $D_{k\ell} = A_k + B_\ell$ and $C_{pq} = A_p \,\&\, B_q$, the two systems with roots D_{11} and C_{11} are going to develop in parallel, each equation having the same δ term and all variables with equal subscripts corresponding two by two:
$\models A_1 = B_1 \Rightarrow \models D_{11} = C_{11} \Rightarrow \models D_{k\ell} = C_{k\ell} \Rightarrow \models A_k = B_\ell$ for all $D_{k\ell}$ and $C_{k\ell}$ connected to D_{11} and C_{11} .

(ii) $\vdash S_A + S_B \equiv S_A \,\&\, S_B \Rightarrow \models S_A = S_B$ since obviously
$\vdash S_A + S_B \equiv S_A \,\&\, S_B \Rightarrow \vdash S_A + S_B = S_A \,\&\, S_B \Rightarrow$
$\vdash A + B = A \,\&\, B \Rightarrow \models A = B \Rightarrow \models S_A = S_B$ ∎

As a corollary we get our end result concerning the completeness of the axiom system:

Corollary 3.13: The axiom system RE with rules of inference R1 and R2 is complete.

Proof: $\models A = B \Rightarrow \vdash S_A + S_B \equiv S_A \,\&\, S_B \Rightarrow \vdash A = B$. ∎

We have

$$\vdash A = B$$

$$\models A = B \Leftrightarrow \vdash S_A + S_B \equiv S_A \,\&\, S_B$$

Theorem 3.12 calls for some remarks.

(i) The proof is essentially the proof of Theorem 2 in Salomaa's

paper. In a sense Salomaa makes a hidden use of $\&$. This becomes particularly clear as we compare his proof to the proof of the equational characterization for $+$ and $\&$.

(ii) The proof is constructive and yields an algorithm to decide the equality of regular expressions. This algorithm is fast and economical and well adapted to the computer handling of large expressions on large alphabets.

(iii) In an actual verification of $S_A + S_B \equiv S_A \,\&\, S_B$ it is not necessary to actually form $S_A + S_B$ and $S_A \,\&\, S_B$, it is sufficient to take all pairs of variables A_i and B_j which would appear in these, starting with A_1 and B_1 and verify that $\delta(A_i) = \delta(B_j)$ and, although it is not necessary, that we do not have $A_i = \emptyset$ and $B_j \neq \emptyset$.[1]

Next we want to study the SRL systems in more detail, apply Theorem 3.12 to the minimization of finite state automata, consider the recognition of regular sets and see how all this can be applied to context-free languages and high-level programming languages.

e) Simplifications and Minimization

Let us now recall the graph we have associated in Section 1 to systems of equations such as in particular SRL systems:

---

[1] Essentially the same algorithm has been independently studied by A. Ginzburg; his findings were presented at the September 1966 Asilomar conference on the algebraic theory of machines, languages and semigroups.

Definition 3.14: The substystem S_{A_i} associated with the variable A_i in a system S_A is the system of equations associated with the subgraph of root A_i .

Given an SRL system, a few simplifications (reduction of the number of variables) can often be easily performed:

(i) connection: A variable not connected to the root may have its equation discarded.

(ii) $\equiv$ -redundancy: If there are 2 variables A_i and A_j such that $S_{A_i} \equiv S_{A_j}$, one can be eliminated. This is frequent and not always obvious.

(iii) $\emptyset$ -redundancy: An SRL system S_A where there is no variable A_i such that $\delta(A_i) = \lambda$ has $\emptyset$ for solution. Let us call it a closed system. All closed subsystems can be eliminated, replaced by $N = \sum_{x \in T} xN$ and their variables are to be replaced by N .

(iv) T -redundancy: In quite a similar way it is always possible to simplify redundant representations of T_1^* , $T_1 \subseteq T$, which are not $\equiv$ -redundant; precisely, they correspond to subsystems where all the variables A_i are such that $\delta(A_i) = \lambda$ and where the coefficients of $N = \emptyset$ are the same in all the equations.

Example: $T = \{a,b,c\}$ $T_1 = \{a,b\}$

$$\begin{cases} A_1 = aA_2 + bA_3 + cN + \lambda \\ A_2 = aA_2 + bA_1 + cN + \lambda \\ A_3 = aA_1 + bA_3 + cN + \lambda \\ N = aN + bN + cN \end{cases}$$

simplifies into

$$\begin{cases} A_1 = aA_1 + bA_1 + cN + \lambda \\ N = aN + bN + cN \end{cases}$$

as

$$A_1 = A_2 = A_3 = T_1^* \quad .$$

Whenever one of these simplification rules is applied it may trigger the applicability of any of the four. They are well adapted to a fast machine implementation. However it is not difficult to find examples of SRL systems where two variables are equal and which cannot be simplified with these four simple rules.

It is a classical result of automata theory that among all finite state automata which accept the same regular set, there is one and only one up to isomorphism which has a minimum number of states (Moore [1956], Theorems 4 and 5) and thus can be taken as a canonical representation of this set.

We are going to prove this result directly in our formalism and give an algorithm to obtain this canonical representation.

The idea is that when $S_A = S_B$, if we form $S_C = S_A \,\&\, S_B$ as in Lemma 3.9, S_C has necessarily no more variables than the smallest

of S_A and S_B. Thus the closure of this operation among all SRL systems equal to S_A is bound to yield a minimal one.

Consider two distinct SRL systems S_A and S_B such that $S_A = S_B$; suppose that they are connected, that S_A has n variables and S_B has m variables.

Form

$$S_C = S_A \,\&\, S_B = S_A = S_B \quad .$$

Consider the process by which, starting with $C_{11} = A_1 \,\&\, B_1 = A_1 = B_1$, the variables in S_C are formed.

$$(\forall i)(\exists j)[A_i = B_j = A_i \,\&\, B_j = C_{ij}] \quad .$$

Clearly S_C cannot have more than (if $n \leq m$ then n else m) variables, and will have less if there are two C_{ij} with equal first or second subscript, since $(\forall k)[C_{ij} = C_{kj}]$ and $(\forall k)[C_{ij} = C_{ik}]$.

We see also that if $n = m$ then S_C has n variables if and only if $S_A \equiv S_B$.

We have proven:

Theorem 3.15: Given a regular set A there is one and only one SRL system $\underline{S}_A$ which has A as its solution and which has a minimum number of variables. This canonical system is the only system S_A in which no two variables are equal.

This yields an algorithm to obtain $\underline{S}_A$:

Given a regular expression A we have shown how to get an SRL system S_A and seen how to simplify it in some cases. We have also an

algorithm to check the equality of two SRL systems.

We now can take all pairs of subsystems in S_A and check them two by two for equality.

The algorithm can be speeded up in two ways:

(i) When an equality is recognized, simplification should be done and we should check for the four elementary simplifications. This may seem to slow the algorithm since we must then start all over again, but in fact drastic simplifications usually occur.

(ii) Given S_A and I_A, consider $I_A/\#$, the quotient of I_A by the equivalence relation $\#$ which we have defined in Section 1.

Lemma 3.16: The graph $<I_A/\#;>>$ has one and only one basis.

Proof:

one: Any finite graph is inductive, any inductive graph has a basis.

only one: $I_A/\#/\# = I_A/\#$.

Thus, there is no circuit in $<I_A/\#;>>$. Clearly a graph can have more than one basis only if it contains a circuit.

In fact, $<I_A/\#;>>$ exhibits the upper lattice property. ∎

We start by putting the subsystems of the basis in canonical form. Then we eliminate any $\equiv$-redundancy, move up one step and put in canonical form the subsystems corresponding to equivalence classes which have only for descendant classes of the basis, etc....

This algorithm, without the last strategy, has been implemented in B5500 Algol. It is well adapted to computer handling of regular

expressions on a large alphabet. The last refinement may in general be questionable because of its computational complication, but it should be a good strategy for dealing with very large systems separable into many smaller subsystems.

SECTION 4

ANALYSIS OF REGULAR SETS

a) The General Problem of Analysis

Let us describe the problem of analysis briefly and rigorously, since we are now often going to refer to it.

A production system is a generative algorithm defining a set $\underline{A}$ of strings in extension.

It is defined by a finite set T of terminal symbols, by a finite set I of variables, among which is the symbol A designating $\underline{A}$, and by a finite set of pairs from $(T + I)^* \times (T + I)^*$, called production rules, and which must be interpreted as rules permitting us to write in any string the second element of the pair in place of an occurrence of the first element.

In particular, in a context-free grammar G_A the production rules are from $I \times (T + I)^*$ and are written $X \to \alpha$, $\alpha \in (T + I)^*$. Clearly we can associate to G_A a system S_A of equations of the form $X_i = f_i(X_1,\ldots,X_n)$, $i = 1,\ldots,n$, with $A = X_1$ where for any i, $f_i(X_1,\ldots,X_n)$ is a form in the algebra of Θ with the operators $+$ and $\cdot$, i.e., a restricted regular form without any $*$.

Consider the following relation: $\alpha \Rightarrow \beta$ with $\alpha,\beta \in (T + I)^*$; it means that β is directly derivable from α by application to α of one production rule in G_A. The closure $\overset{*}{\Rightarrow}$ of $\Rightarrow$ is obviously a relation of order (derivability), thus $\Rightarrow$ defines an infinite directed graph $\langle (T + I)^*; \Rightarrow \rangle$. An interesting subgraph of this

graph is the one which contains A and all the paths starting at A (graph of all strings derivable from A). We can label each arc (α,β) in those graphs with the name of the production rule by which $\alpha \Rightarrow \beta$, with some conventional notation for specifying where in α the rule is applied in case there may be ambiguity. (See example.)

Naturally, the set of strings or context-free language $\underline{A}$ is the set $\{\alpha | \alpha \in T^* \wedge A \overset{*}{\Rightarrow} \alpha\}$.

To analyze a string α is to find all the parses of α , that is all the paths in $<(T + I)^*;\Rightarrow>$ joining A to α , each one defining a derivation from A to α by G_A . In a derivation of α certain phases may lead to some disjoint parts of α and usually such phases are then considered to be independent. Two derivations which differ only by the order of independent phases are equivalent; a convenient representation of an equivalence class of derivations of α is a tree, the well-known structural tree of α , in which independent phases are shown as developing as independent branches; another often used representation is by one of the elements of the class, a path called canonical parse which corresponds to a rule of selection in $<(T + I)^*;\Rightarrow>$.

If more than one structural tree or canonical parse can be associated to a string α , α is said ambiguous. A context-free language is ambiguous when some of its strings are; this notion is relative to the grammar. A context-free language is inherently ambiguous when it is ambiguous for all its context-free grammars; this notion is relative to the class of context-free grammars.

Example: $T = \{i,[,]\}$ $I = \{A,B\}$.

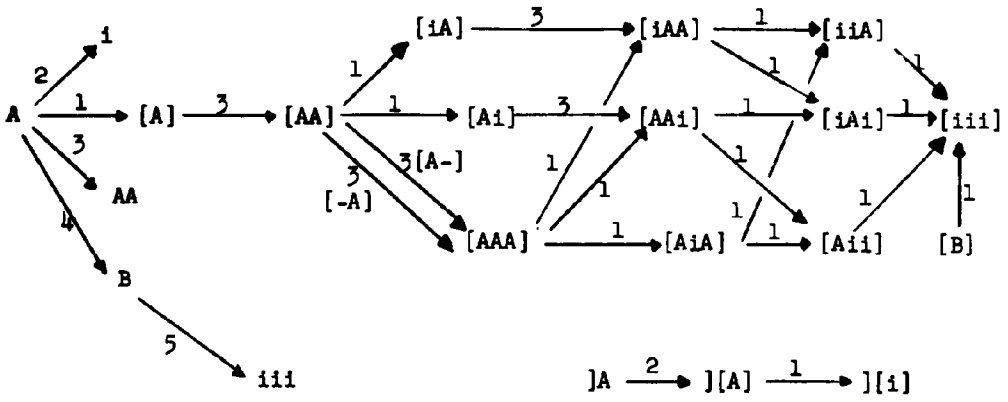
G_A contains five production rules:

1. $A \rightarrow i$
2. $A \rightarrow [A]$
3. $A \rightarrow AA$
4. $A \rightarrow B$
5. $B \rightarrow iii$

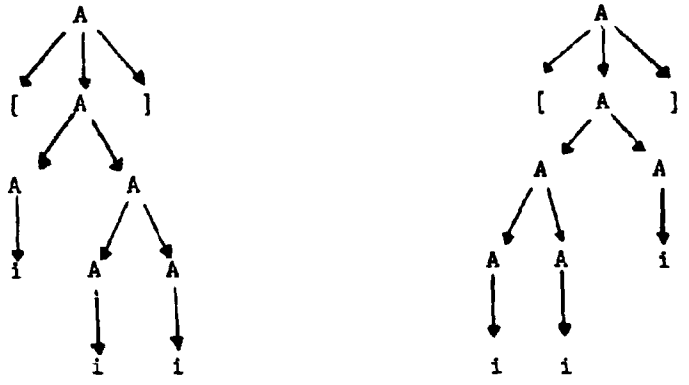
S_A contains two equations: $A = i + [A] + AA + B$

$B = iii$

Let us draw a part of $\langle (T + I)^*; \Rightarrow \rangle$.



There are 16 paths from A to [iii] , they correspond to the two following equivalence classes (represented by their structural trees):



[iii] is ambiguous.

We see that to analyze a string α is to solve constructively a combinatorial problem and thus to extract some information from α. This information is used for instance to direct a computer (interpreter), to generate some code (compiler), sometimes even to alter α at the same time it is analyzed (macro generation). These actions can be specified by factorization into elementary steps each of which is associated to one production rule, so that to a given path corresponds a succession of elementary steps driven by the analyzer (see for instance, Wirth and Weber [1965]). This association of analysis and action is mathematically a valuation; in Riguet [1962] it is shown how it corresponds to the algebraic notion of diagram defined on a directed graph with value in a category.

Let us only observe here that the notion of equivalence of two derivations and the notion of ambiguity of a string are both dependent upon valuation. For instance, the equivalence of two derivations has no operational value when valuation alters the strings as they are

analyzed; for instance also, the ambiguity of a string is unimportant if analysis is merely intended to decide whether the string belongs to the set A or not, or, more important, if analysis bypasses the ambiguity because of some systematic convention. We shall give an example of the latter in Section 5d.

The value of a string is often called its semantic; the valuation mapping together with the class of values of all strings in a language being then considered as a model of the language.

Basically, there are two ways of analyzing a string α ; we may start from A and try to reach α , following the arrows, or start from α and try to reach A , going against the direction of the arrows. The first method is called top-down analysis, the second one bottom-up. Although it is never done, there is no theoretical reason for not devising analyzers using a mixture of both.

If we have described the problem of analysis in general terms, it is because we believe that it is more general than the problem of compiling or interpreting programming languages. We will come back to this in Section 6g. Let us recall the following points we have made: there is a difference between analysis and valuation; the structural tree of a string α is not an inherent property of α , it describes a successful analysis, showing the relations and subordinations of the different phases.

b) Analysis of Regular Sets

(i) Top down analysis.

In the case of regular sets there are various ways to show that top-down analysis of the strings of a regular set is simply done by building a corresponding finite state automaton and feeding strings into it (see for instance, Brzozowski [1964]). The automaton can always be made deterministic (Rabin and Scott [1959], Theorem 11) and analysis proceeds from left to right in a time proportional to the number of symbols read.

In our formalism: Let $T = \{x_j | j = 1,\ldots,r\}$.

Let S_A be an SRL system, $A_i = \sum_{j=1}^{r} x_j A_{ij} + \delta(A_i)$ $i = 1,\ldots,n$.

We associate to each A_i a predicate In_i , such as $In_i(\alpha) = [\alpha \in A_i]$, as follows:

$$In_i(\alpha) = \text{if } \alpha = \lambda \text{ then } \delta(A_i) = \lambda \text{ else } Next_i(first(\alpha),rest(\alpha))$$

$$Next_i(x,\beta) = \text{if } x = x_1 \text{ then } In_{i1}(\beta) \text{ else } \ldots$$

$$\ldots\text{if } x = x_{r-1} \text{ then } In_{ir-1}(\beta) \text{ else } In_{ir}(\beta) \quad .$$

Since In_i may appear for instance as some In_{ij} in $Next_i$, these predicates appear as recursive. However it is clear that this recursion is computationally equivalent to an iteration: in the implementation of procedures corresponding to these predicates it is not necessary to use a pushdown store because control will only enter these procedures at their beginning.

In practice we will use an n by r array representing the transition graph; for instance, to the system given as an example following 3.5,

$$A_1 = 0A_2 + 1A_1$$
$$A_2 = 0A_3 + 1A_1$$
$$A_3 = 0A_3 + 1A_4 + \lambda$$
$$A_4 = 0A_3 + 1A_4$$

corresponds the array:

	A_1	A_2	A_3	A_4
0	A_2	A_3	A_3	A_3
1	A_1	A_1	A_4	A_4
λ			λ	

We go from state to state as we read characters one by one.

The minimal SRL system corresponds to the smallest array. The algorithm can be speeded up by grouping characters into strings corresponding to closed paths, i.e., redefining T .

Because of the speed and simplicity of this algorithm to analyze regular sets, it would be reasonable to use it systematically for analyzing regular structures in programming languages; even if the general analysis algorithm used does not reduce to this one in the particular case of regular sets. The fact that it is not recursively decidable whether a context-free language is regular (Bar Hillel, Perles and Shamir [1961], Theorem 6.3) does not cause any difficulty, one defines a grammar for a language one has in mind, not the contrary.

Note the role of the end of string marker, λ , to prevent ambiguities (see Chomsky and Miller [1958]).

Note also that we have here a case of predictive analysis in its simplest form (Kuno and Oettinger [1962]).

(ii) Bottom-up analysis.

Suppose a regular set is defined by an SRL or SLL (standard left linear) system and we are trying to find the (unique) path corresponding to the derivation of a string α.

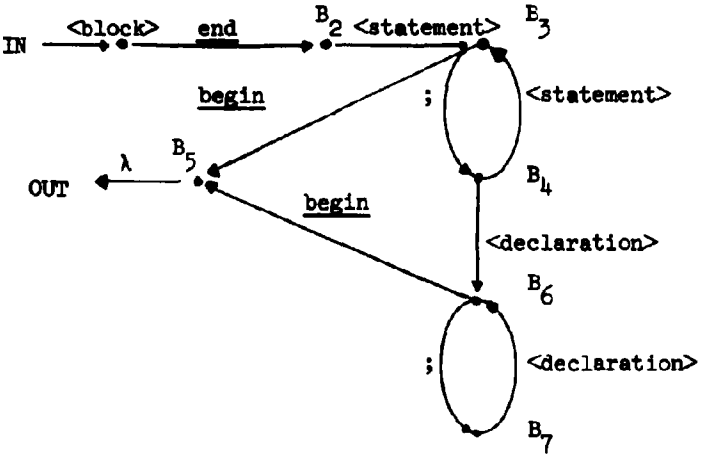
Example: Block structure in Euler.

<block> = begin (<declaration>;)*(<statement>;)*<statement> end

Because it is desirable to scan a block from left to right in order to build its declaration table first, we must consider an associated SLL system, rather than SRL: (elements equal to $\emptyset$ not written).

1. <block> = B_2 end
2. B_2 = B_3<statement>
3. B_3 = B_4; + B_5 begin
4. B_4 = B_3 <statement> + B_6 <declaration>
5. B_5 = λ
6. B_6 = B_7; + B_5 begin
7. B_7 = B_6 <declaration>

corresponding to the graph: (Labelled as traditional)



Consider the string:

begin <declaration> ; <statement> ; <statement> end

It has only one parse:

<block>
↓ 1
B_2 end
↓ 2
B_3 <statement> end
↓ 3
B_4 ; <statement> end
↓ 4
B_3 <statment> ; <statement> end
↓ 3
B_4 ; <statement> ; <statement> end
↓ 4
B_6 <declaration> ; <statement> ; <statement> end
↓ 6
B_5 begin <declaration> ; <statement> ; <statement> end
↓ 5
λ begin <declaration> ; <statement> ; <statement> end

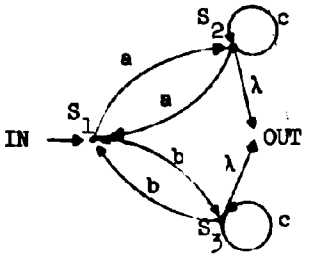
We want to reconstruct that parse as we read the string from left to right. We start at the left end, B_5 must have been applied; now we have B_5 begin ... ; B_5 begin appears in 6 and in 3, there is no way to know whether we must use 6 or 3 except to look at the ollowing symbols; since the next symbol is <declaration> we must apply 6, not 3. The situation is worse when later we get B_3 <statement>... or B_6 <declaration>... : we have to look two symbols ahead in order to make a decision since both <declaration> and <statement> must be followed by a ";" .

If we were proceeding by trial and error, we see ' . we would get into blind alleys, none of which would be longer than two analysis steps.

This important type of difficulty has been intensively studied for context-free languages (Floyd [1964], Irons [1964], Ross [1964], Wirth and Weber [1965], Knuth [1965]); in the particular case of regular sets, we recognize the notion of a k-limited automaton (2-limited, in the example) (Chomsky [1963], page 336-7). Because there are finite state automata which are not k-limited for any k (ibid.) we see that there are SRL systems for which a bottom-up analyzer will engage into blind alleys of unbounded length.

Example: $S_1 = (ac^*a + bc^*b)^* + ac^* + bc^*$

$$S_1 = aS_2 + bS_3$$
$$S_2 = cS_2 + aS_1 + \lambda$$
$$S_3 = cS_3 + bS_1 + \lambda$$



In fact, it is clear that in the notion of k-limited automaton, the finiteness of the automaton does not play any role and that the notion is generalizable to infinite automata and context-free languages.

Let us remark that to go into a blind alley and then backtrack is in practice untolerably time and space consuming and must be avoided when a decision can be made simply by a short look-ahead. Certainly a disadvantage of the context-free grammar formalism is that it implies the use of a non-deterministic analyzer, even in simple cases. Look-ahead just cannot be described in this formalism. This is one reason why in Section 6 we shall introduce conditionals in a formalism related to context-free grammars.

Before that we want to apply to context-free languages the results and the considerations of this last section and of Section 3.

SECTION 5

APPLICATION TO PROGRAMMING LANGUAGES

a) Preliminaries

Most programming languages make use of structures conveniently described by regular expressions, such as the block structure we have examined in Section 4b; since regular sets are simple to analyze, we want to take advantage of this.

We will first examine as an example the case of Euler (Wirth and Weber [1965]), in preference to Algol 60 because Euler has an unambiguous, simple and systematic syntax. Furthermore Euler is a generalization of Algol 60.

Seeing that Euler can effectively be analyzed by recursion of finite state functions without backtracking, we will formalize this approach to language recognition, define a class of sets of strings which we will call regular context-free (RCF) and study its properties.

b) Regular Structures in Programming Languages

Euler syntax (see Appendix 4) is defined in Wirth and Weber [1965] by a simple precedence context-free grammar consisting of 120 production rules in a notation similar to BNF without the vertical stroke for alternation (Boolean +); were this sign used, it would reduce the system to some 44 rules, 35 after elimination of some redundancies necessary to insure precedence.

It is clear that we can consider a context-free grammar as a system of equations in Θ homomorphic to a graph (see Section 1), the only

operators used are · and + . * could clearly be used and corresponds to terminating left or right recursive rules; as we shall see in Lemma 5.4, non-terminating left or right recursions define variables equal to $\emptyset$. Such systems have one and only one solution (Ginsburg and Rice [1962], Letichevskii [1965]).

If in the Euler system we solve left and right recursions by introducing * and then solve the system by substitution as much as this can be done, we eventually obtain no more than two large equations in one variable, one equation being recursive:

program = f(expr)

expr = g(expr)

see Appendix 4 for f and g .

This is not enough to ensure that Euler can be analyzed without trial and error by a recursive use of the finite state functions f and g , because it could happen that the analyzer would not know in some cases when to go up or down one level in recursion rather than to keep absorbing symbols on the same level, so that it would have to proceed by trial and error; in terms of programming, we say that it would backtrack, in terms of automata theory that it would simulate a non-deterministic automaton.

In the Euler case, wherever expr occurs in f or g , it is surrounded by two bracketing symbols. These symbols are used only for bracketing and there is no choice within the brackets. This clearly shows that Euler can be deterministically recognized by two finite-state automata, one of which can call itself recursively by way of a pushdown store, on which the place where a recursion must return is saved when

the recursion is entered. Such analyzer is not only extremely fast but also minimizable.

Since Euler is a generalization of Algol 60 this method seems promising. In fact the Meta series of compiler compilers (Schorre [1963], [1964], Schneider and Johnson [1964]) implicitly uses a variant of it, although in a non-systematic and informal way.

The role of regular structures in programming languages was first recognized in Čulík [1962] and rediscovered by Carr and Weiland [1966] in a misleading paper where it was wrongly argued that it is possible to express with regular expressions "the Revised Algol 1960 syntax in completely nonrecursive terms". Neither its problems nor its implications have been studied.

Note that the role of the operator * is to force us to analyze iteratively what it is not necessary to analyze recursively; in this strategy the push-down store is used as little as possible.

What we must do now is to rigourously define the strategy we have broadly described, characterize the subclass of those deterministic context-free languages which can be analyzed with it and examine their properties.

c) RCF Languages. Characterizations.

We need first to introduce some important notions.

Definition 5.1: The left derivative of A with respect to B, $\partial_B A$ is defined by

$$\partial_B A = \{\alpha \mid (\exists \beta)[\beta \in B \wedge \beta\alpha \in A]\} \quad .$$

$\delta_B A$ is the set of all strings obtained by chopping off a string in B at the head of a string in A.

Particularly when B is a unit set of one string, this notion is central to the gedanken experiment oriented theory of automata. The variables in an SRL system S_A are equal to derivatives of A. This approach is used in Stearns and Hartmanis [1963] and Brzozowski [1964] for regular sets.

Of interest to us here is the left derivative of a set with respect to itself:

$$\delta_X X = \{\alpha | (\exists\beta)[\beta \in X \wedge \beta\alpha \in X]\} \quad .$$

Definition 5.2: A predicate Π on Θ^2 is defined by:

$$\Pi(A,B) = [\text{first}(\delta_A A) \cap \text{first}(B) = \emptyset] \quad .$$

We shall say that A is separable in AB.

This definition corresponds to the difficulty we have mentioned in Section 5b. Suppose we are analyzing γ from left to right, where $\gamma \in C = AB$. Necessarily $(\exists\alpha)(\exists\beta)[\gamma = \alpha\beta \wedge \alpha \in A \wedge \beta \in B]$. The analyzer for C calls upon the analyzer for A first; when the analyzer for A comes to the end of α it should be dismissed and the analyzer for B called upon, but if there is a string $\alpha\alpha_1 \in A$ where α_1 and β have an initial non-null segment in common we are unable to recognize at the end of α whether the analyzer for A has to be dismissed or not. We see that if A is separable in AB this cannot occur.

Observe that the operation by which the analyzers for A and B are called successively to form the analyzer for C corresponds to the notion of function of function.

This notion of separability is important and will be often used. We shall write $\Pi(A,B)$ for $\Pi(A,B) =$ true .

Because $\Pi(A,B)$ expresses a property of sets, its value is preserved when we substitute for A or B expressions to which they are equal.

Let us now define the class of sets we are interested in.

Let S be a system of equations:

$$X_i = f_i(X_1,\ldots,X_n) \qquad i = 1,\ldots,n \quad ,$$

where f_i is a restricted regular expression over $T \cup I_S$. Consider the system S' obtained by developing each f_i into its canonical SRL system, introducing new variables B_{ij} :

$$X_i = B_{i1}$$

$$B_{ij} = \sum_{a\epsilon T} aB_{ija} + \sum_{X\epsilon I_S} XB_{ijx} + \delta(B_{ij}) \qquad \begin{matrix} i = 1,\ldots,n \\ j = 1,\ldots,m_i \end{matrix} .$$

Definition 5.3: A set of strings is regular context free (RCF) when it is the solution of a system S' in regular form; i.e., containing only two types of equations:

(i) $$B_{ij} = \sum_{a\epsilon T} aB_{ija} + \delta(B_{ij})$$

(ii) $$B_{kl} = XB_{klx} \quad \text{with} \quad \Pi(X,B_{klx}) \quad .$$

To say that X is separable in $B_{k\ell}$ means simply that

$$(\forall\alpha)(\forall\beta)[((\beta \neq \lambda) \wedge (\alpha \in X) \wedge (\alpha\beta \in X)) \Rightarrow \text{first}(\beta) \notin \text{first}(B_{k\ell X})] .$$

The adequacy of this definition to the algorithm we wish to use is due to the fact that the process of expansion into S' is a formal representation of the algorithm.

Π is decidable for context-free languages. Algorithms for its computation are given and discussed in Appendix 5. Usually, as in the Euler case, Π is obviously true.

Lemma 5.4: In any system corresponding to a context-free grammar, a variable defined by a non-terminating recursion is equal to $\emptyset$.

Proof: $\emptyset$ is a solution of the corresponding subsystem.

The solution is unique. (Ginsburg and Rice [1962], Letichevskii [1965]). ∎

Example: $N = aN + bN + aNb + NN$.

Since I is finite, the occurence of variables equal to $\emptyset$ can be recognized by mere testing and the system can be simplified by the rules $A\emptyset = \emptyset A = \emptyset$ and $A+\emptyset = \emptyset+A = A$.

When one has defined a class of sets of strings, it is often useful, as a tool to study its properties, to characterize it in terms of a family of automata each of which recognizes just one set of the class. Here we are clearly going to obtain a subfamily of the one-way deterministic pda (push-down store acceptors) (Schützenberger [1963], Ginsburg and Greibach [1965]).

As we shall see, RCF languages can be characterized by properties of systems of equations in more than one way. Depending on the particular characterization one uses, the class of automata can be defined by various forms of restrictions, necessarily all equivalent, but more or less natural. We are going to introduce one which we find natural.

Definition 5.5: (Greibach [1965]). A grammar rule is in standard form when it is of the type $X \to aX_1 \ldots X_n \quad n \geq 0$. A grammar is in standard form when all of its rules are.

For any context-free set L , $L-\lambda$ has a standard-form grammar (ibid.): this result is the formulation for grammars of an automaton-theoretic result: to any pda terminating its computation with an empty pushdown store, one can associate another one which defines the same set of strings, under the same condition, and has a finite state control with just one state (Ginsburg [1966], Lemma 2.5.1). Clearly the latter works on a left-right, top-down, generally non-deterministic, recognition.

Lemma 5.6: In any CF system obtained from a regular form system by substitutions of equals for equals, if X and Y are two consecutive variables in the right part of a production rule, X is separable from XY .

Proof: We have two types of rules

(i) $$B_{ij} = \sum_{a \in T} aB_{ija} + \delta(B_{ij})$$

(ii) $$B_{k\ell} = XB_{k\ell x} \quad \text{with} \quad \Pi(X, B_{k\ell x}) \ .$$

In a first substitution there can be a difficulty only when substituting for X the expression to which X is equal. But because separability is a property of sets, not of grammars, it will be preserved. The same reasoning is clearly true for other steps of substitution. (Note in particular that by definition of Π the case $\lambda \in X$ is not excluded.)

Definition 5.7: A grammar is an s'-grammar when all of its production rules are in standard form, $X \to aX_1 \ldots X_j X_{j+1} \ldots X_n$, $n \geq 0$, or of the type $X \to \lambda$, subject to the conditions that

(i) for any ordered pair (X_j , X_{j+1}) appearing on the right of a production rule, X_j is separable in $X_j X_{j+1}$, i.e., $\Pi(X_j, X_{j+1})$.

(ii) no two production rules having the same X have the same a .

Let us work out an example which we shall generalize afterward.

Consider the following definition of a simplified arithmetic expression, where the operators are $\odot$ and $\otimes$ and the parentheses are denoted by square brackets; A stands for arithmetic expression, T for term, F for factor, i for identifier (a terminal symbol).

$$A = T + A \odot T$$

$$T = F + T \otimes F$$

$$F = i + [A]$$

Solving the left recursive equations in this system, which is equivalent to the usual context-free grammar for arithmetic expressions, we get:

$$A = T(\odot T)^*$$

$$T = F(\otimes F)^*$$

$$F = i + [A]$$

All variables but A can be eliminated and we can get A as a function of A. For clarity, let us not do it now; we eliminate only T:

$$A = F(\otimes F)^*(\odot F(\otimes F)^*)^*$$

$$F = i + [A]$$

We now expand A into a regular form system: (quantities equal to ϕ not written)

$$A = A_1$$

$A_1 = FA_2$ where (rule $B^* = {}^{l}BB^* + \lambda$ applied twice):

$$\begin{aligned} A_2 &= (\otimes F)^*(\odot F(\otimes F)^*)^* \\ &= \otimes F(\otimes F)^*(\odot F(\otimes F)^*)^* + (\odot F(\otimes F)^*)^* \\ &= \otimes A_1 + \odot F(\otimes F)^*(\odot F(\otimes F)^*)^* + \lambda \\ &= \otimes A_1 + \odot A_1 + \lambda \end{aligned}$$

so that we get directly the minimal SRL system (over $T + [F]$)

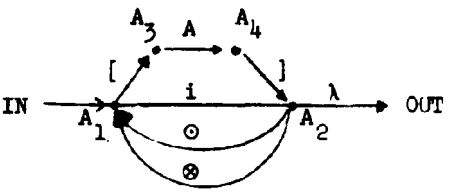
$$A = A_1$$

$$A_1 = FA_2$$

$$A_2 = \otimes A_1 + \odot A_1 + \lambda$$

Substituting now $i+[A]$ for F we get the regular form system:

$$A = A_1$$
$$A_1 = iA_2 + [A_3$$
$$A_2 = \otimes A_1 + \odot A_1 + \lambda$$
$$A_3 = AA_4$$
$$A_4 =]A_2$$



$\Pi(A,A_4)$ since $first(A_4) = \{]\}$ while $first(A_3A) = \{\odot,\otimes\}$.

Now we want an s'-grammar; the only equation to be expanded is $A_3 = AA_4$. By substitution

$$A_3 = A_1A_4 = iA_2A_4 + [A_3A_4$$

Note that by Lemma 5.6

$$\Pi(A,A_4) = \Pi(A_1,A_4) = (\Pi(A_2,A_4) \wedge \Pi(A_3,A_4))$$

and the s'-grammar is:

$$A_1 \rightarrow iA_2$$
$$A_1 \rightarrow [A_3$$
$$A_2 \rightarrow \otimes A_1$$
$$A_2 \rightarrow \odot A_1$$
$$A_2 \rightarrow \lambda$$
$$A_3 \rightarrow iA_2A_4$$
$$A_3 \rightarrow [A_3A_4$$
$$A_4 \rightarrow]A_2$$

Note that all these manipulations can be done by a computer which would easily handle much larger expressions.

Theorem 5.8: Any RCF set has an s'-grammar; any s'-grammar defines an RCF set.

Proof: (i) Any RCF set has an s'-grammar.

Starting with a regular form system, we eliminate all variables equal to $\emptyset$ (Lemma 5.4). In the resulting system we want to eliminate terms of the form $B = CD$. In all such monomials, where the leftmost symbol is a meta-variable C, we substitute for C the expression which defines it. By Lemma 5.4 the process terminates. There is a difficulty only in case in the last substitution $\delta(C) = \lambda$. We obtain then an equation of the form:

$$B = \sum_{a \in T_1} aC_{1a}C_1 \ldots C_n D + C_1 \ldots C_n D + \delta(B) \qquad T_1 \subseteq T \ .$$

For two consecutive variables C_{iaj}, $C_{(i+1)aj}$, $\Pi(C_{iaj}, C_{(i+1)aj})$, by Lemma 5.6.

$\Pi(C,D)$ implies that $T_i \cap T_j = \emptyset$ $\quad i,j = 1,\ldots,p$.

The transition from the system thus obtained to an s'-grammar is immediate.

(ii) Any s'-grammar defines an RCF set.

To a rule $X \to aX_1 \ldots X_n$ associate

- if $n > 1$ the n equations ($Y_1,\ldots,Y_{n-1}$, $n-1$ new symbols):

$$\begin{aligned} X &= aY_1 + \delta(X) \\ Y_1 &= X_1 Y_2 \\ &\cdots\cdots \\ Y_{n-1} &= X_{n-1} X_n \end{aligned}$$

- if $n = 1$ the equation $X = aX_1 + \delta(X)$

- if $n = 0$ the equations $X = aL + \delta(X)$

$$L = \lambda$$

where $\delta(X) = \lambda$ if a rule $X \to \lambda$ occurs in the s'-grammar and $\delta(X) = \emptyset$ otherwise.

We cannot have $\Pi(X_i, Y_{i+1}) =$ false in one of these equalities, otherwise we would have for some j, $i \leq j \leq n$, a pair (X_j, X_{j+1}) for which $\Pi(X_j, X_{j+1}) =$ false.

We obtain a regular form system. ∎

Note that if we were not concerned with separability and regular form systems, we would have here a simple algebraic proof of the existence of a standard-form grammar for any context-free set not containing λ (Greibach [1965]); the rules $X \to \lambda$ being eliminated by the method of Lemma 4.1 in Bar Hillel, Perles and Shamir [1961], which amounts to a substitution of $X_1 + \lambda$ for X, where $X_1 = X - \lambda$ is a new, easily defined, context-free set.

We may call a pre-standard-form grammar a grammar which is made of a standard-form grammar plus, possibly, a rule $A \to \lambda$, where A is the root. Any context-free set has such a grammar.

Corollary 5.9: Any RCF language has an s'-grammar in which no rule has a right part containing more than two variables (s'-2-grammar).

Proof: We use the construction of Corollary 3.2 in Greibach [1965]. It preserves the properties of s'-grammars.

Let n be the length of the longest right part of a rule. We show that if $n > 3$, we can construct an equivalent s'-grammar with n reduced by one.

For each pair of variables $A,B \in I$ create a new symbol $[A,B]$. For each $[A,B]$ scan all the rules $A \to aA_1 \ldots A_p$; if $p \leq n-2$, create a rule $[A,B] \to aA_1 \ldots A_p B$; if $p = n-1$, create a rule $[A,B] \to aA_1 \ldots A_{n-2}[A_{n-1},B]$; if $p = n$, create a rule $[A,B] \to aA_1 \ldots A_{n-3}[A_{n-2},A_{n-1}][A_n B]$. If $\delta(A) = \lambda$, scan all the rules $B \to bB_1 \ldots B_q$; if $q \leq n-1$, create a rule $[A,B] \to bB_1 \ldots B_q$; if $q = n$, create a rule $[A,B] \to bB_1 \ldots B_{n-2}[B_{n-1},B]$. If $\delta(B) = \lambda$ too, create a rule $[A,B] \to \lambda$. Now replace all rules $X \to xX_1 \ldots X_n$ of the old grammar by a rule $X \to xX_1 \ldots [X_{n-1},X_n]$; this connects a number of new variables $[X_{n-1},X_n]$ to the root, take the productions which correspond to them and discard the unnecessary ones.

It is clear that if the resulting grammar were not an s'-grammar the old one could not be one since only substitutions are applied. ∎

Note that, as we shall see in Section 5d, an s'-grammar or an s'-2-grammar can be ambiguous, although it is possible to derive from them deterministic parses; their form is particularly convenient to prove results about RCF languages and to characterize them automaton-theoretically.

Notation: A one-way deterministic pda M (see for instance, Ginsburg [1966], Section 2.6) with one final state, is given by

$$M = (K,T,I,\eta,\delta,q_o,\{q_o\}) \quad \text{where}$$

K is the set of states of the finite state control automaton,

$K = \{q_0,\ldots,q_n\}$.

T is the input alphabet. $\{a_1,a_2,\ldots a_r\}$

I is the finite set of pushdown symbols. $\{A,B,\ldots\}$

η a mapping from $K \times (T \cup \{\epsilon\} \cup \{\lambda\}) \times (I \cup \{\lambda\})$ into $K \times I^*$.

(transition function)

$S \in I$ (initial pushdown symbol)

q_0 is the initial state and only final state.

η is defined by rules of 4 types:

1. $(q_i,a_k,X) \rightarrow (q_j,Y_1\ldots Y_n)$ $\quad n \geq 1$ (Read and Expand)

The control in state q_i , $a_k \in T \cup \{\lambda\}$ current symbol on the input tape, or $a_k = \lambda$ meaning that the input tape is empty, X on top of the pushdown store, or $X = \lambda$ meaning that the pushdown store is empty; a_k is read in, q_j is reached and X replaced by $Y_1\ldots Y_n$ where Y_n is now on top of the pushdown store.

2. $(q_i,\epsilon,X) \rightarrow (q_j,Y_1\ldots Y_n)$ $\quad n \geq 1$ (Expand only)

same as in 1 but the operation does not depend upon the input which is not read in.

3. $(q_i,a_k,X) \rightarrow (q_j,\epsilon)$ (Read and Erase)

same as 1 but X is erased instead of expanded.

4. $(q_i,\epsilon,X) \rightarrow (q_j,\epsilon)$ (Erase only)

a combination of 2 and 3.

Definition 5.10: An s'-machine is a one-way deterministic pda with one final state, which satisfies the following 4 restrictions:

(i) $\text{card}(K) \leq \text{card}(T) + 1$.

(ii) all Read and Expand rules are transitions from q_o to q_o

— Expand Only ——————————— $q_i - q_o$ $i \neq 0$

— Read and Erase ——————————— $q_o - q_i$ $i \neq 0$

— Erase Only ——————————— $q_i — q_i$ $i \neq 0$ (same i).

Except that a Read and Erase rule where λ is read is a transition from q_o to q_o .

(iii) To each state q_i , $i \neq 0$, is associated one letter a_i in T , one-to-one so that

- For each Expand Only rule from state q_i , there is a Read and Expand rule where a_i is read-in, the remainder of those two rules being identical.

- In a Read and Erase rule, the read-in a_i corresponds to the q_i which is reached from q_o , λ to q_o .

- For each Erase Only rule in q_i there is a Read and Erase rule where a_i is read-in, the remainder of those two rules being identical.

If a pushdown symbol appears in the left part of a Read and Erase or Erase Only rule, it appears with every letter in T in the left part of some rule of type Read and Expand, Expand Only, Read and Erase, or of type Read and Expand, Expand Only, or Erase Only, respectively.

(iv) It terminates its computation with an empty pushdown store.

These rules are phrased in such a way that the states q_i different from q_o and only those states are used just to "remember" one character a_i for look-ahead purpose; so that, since the machine can remember only one letter, it cannot read when in a state q_i, $i \neq 0$: it just stays in q_i and pops out pushdown symbols until it gets one which would be expanded with a_i. It then expands it and returns to q_o. Note that it is forbidden by the last restriction in (iii) to use the pushdown store to remember from step to step the last letter read-in.

These rules certainly are complicated; it is not clear how they could be made simpler; on the other hand the functioning of the machine is intuitively quite simple.

Theorem 5.11: Any RCF set can be recognized by an A-machine and conversely any s'-machine recognizes an RCF set.

Proof: By Theorem 5.8 we can start from an s'-grammar.

To a production

$$B \to a_i X_1 \ldots X_n$$

associate the rules

$$(q_o, a_i, B) \to (q_o, X_n \ldots X_1) \quad \text{and}$$

$$(q_i, \varepsilon, B) \to (q_o, X_n \ldots X_1) \quad .$$

To a production

$$B \to \lambda$$

associate for all i such that $a_i \notin \text{first}(B)$, the rules

$$(q_o, a_i, B) \rightarrow (q_i, \varepsilon) \quad ,$$

$$(q_i, \varepsilon, B) \rightarrow (q_i, \varepsilon) \quad ,$$

and

$$(q_o, \lambda, B) \rightarrow (q_o, \varepsilon) \quad .$$

Separability and the fact that no two productions with the same B on the left can have the same a_j on the right express precisely that the pda just defined is deterministic.

If the computation starts in q_o with the pushdown store containing the root S, it will stop in q_o with an empty pushdown store and a completely read-in input string α, if and only if α belongs to the language defined by the s'-grammar.

The converse is obtained by the reverse argument. ∎

d) Relation to Other Classes of Languages

We are going to relate RCF languages to two other classes of languages which have been recently introduced.

In Korenjak and Hopcroft [1966], one-way deterministic pda are studied which

(i) must read one input symbol per pushdown symbol erased,

(ii) end their computation with an empty pushdown store and

(iii) have a finite state control of just one state: the s-machines.

They define the s-languages. If we define an s-grammar to be a

standard-form grammar in which no two rules $X \to aX_1 \ldots X_n$, $n \geq 0$ having the same X have the same a , any s-grammar defines an s-language, any s-language has an s-grammar.

For such a grammar predictive analysis cannot go into a blind alley; another important property is that no initial segment of a word derivable from a variable can be derived from it too (prefix property); this implies left and right cancellation.

Lemma 5.12: The class of s-languages is properly included in the class of RCF languages.

Proof: - Any s-grammar is an s'-grammar because the prefix property implies separability.

- The finite set $\{a,ab\}$ does not have the prefix property, it is not an s-language. It is RCF as any regular set is. ∎

We want to give a more complex example because any regular set with an end marker is an s-language (ibid.) and one may wonder whether s'-languages are but quite a mild generalization of s-languages.

Consider the set $S = \{b^{m+1}c^*c^nac^nb^{m+1} | n,m \in \Omega\}$. This is not an s-language because an s-machine cannot recognize it, since it cannot save information except on its pushdown store and must read one input symbol per pushdown symbol popped out.

An s'-2-grammar which defines it is

$$S \rightarrow bS_1B$$

$$S_1 \rightarrow bS_1B$$

$$S_1 \rightarrow cS_2K$$

$$S_2 \rightarrow cS_2K$$

$$S_1 \rightarrow aL$$

$$S_2 \rightarrow aL$$

$$B \rightarrow bL$$

$$L \rightarrow \lambda$$

$$K \rightarrow cL$$

$$K \rightarrow \lambda$$

Note that the set S@ , @ an end marker, is not an s-language either, for the same reason as S .

One must emphasize that this last grammar is ambiguous: the derivable intermediate string, bbccccaKKKKBB for instance, can yield bbccccaccbb in 6 ways, according to the K 's which are parsed into λ ; for the corresponding s'-machine, the rule $K \rightarrow \lambda$ is to be applied only when the input character read-in does not belong to first(K) . This cannot be expressed in an s'-grammar; this supplement of information is in Π and in the restrictive rules by which an s'-machine, i.e., a deterministic parse, is derived from the s'-grammar.

To make such condition explicit we could use unrestricted rewriting rules such as: $Kb \rightarrow b$.

Let us consider now another class of languages.

The Meta series of compiler compilers (Schorre [1963] [1964], Schneider and Johnson [1964]) uses restrictions of regular form systems

which have not been studied or even made precise at this writing; because, as we have seen in the Euler case, separability is often in practice trivially recognizable, it may have seemed unnecessary to give as general a condition as possible for a language to be in the realm of the method. An attempt is made though in Schorre [1965].

Definition 5.13: (ibid.) A binary grammar is a context-free grammar in which all the rules have just two symbols on the right or are of the form $A \rightarrow \lambda$.

Let us use the notation $\underline{A},\underline{B},\ldots$ for symbols which are either upper or lower case.

Definition 5.14: (ibid.) An fcr (first character recognition) grammar is a binary grammar in which:

(i) if $A \rightarrow \underline{BC}$ and $A \rightarrow \underline{DE}$ are in it, then
first($\underline{BC}$) & first($\underline{DE}$) = $\emptyset$ and $\delta(\underline{BC})$ & $\delta(\underline{DE}) = \emptyset$.

(ii) if $A \rightarrow \underline{BC}$ is a rule, $\underline{B}$ is separable from $\underline{BC}$.

Some properties are given in Schorre [1965], yielding an algorithm to determine whether a grammar is fcr. No further investigation of these grammars and of the class of languages they define has been published and the relation of fcr grammars to the formalism used in the Meta series of compiler compilers has not been made clear.

Note that because Π is preserved by substitution, we have in any derivable string $\underline{A_1A_2}\ldots\underline{A_n}$, $n > 1$, $\Pi(\underline{A_i},\underline{A_{i+1}})$, $i = 1,\ldots,n-1$; so that the restriction that the grammar be binary is unnecessary.

Lemma 5.15: There are no terminating left recursions in an fcr grammar.

Proof: We certainly cannot have a direct terminating left recursion: if $B \to \underline{BC}$ with $B \to \underline{EF}$ or $B \to \lambda$, then $\underline{EFC}^* \subseteq B$ or $C^* \subseteq B$ which is not compatible with respectively $\text{first}(BC) \cap \text{first}(EF) = \emptyset$ or $\Pi(B,C)$. If a left recursion is not direct, it can be reduced to a direct one by successive substitutions, without altering separability. ∎

Note that is is as usual possible to get rid of nonterminating recursions by eliminating any variable equal to $\emptyset$. We will suppose that this has been done.

Theorem 5.16: Any RCF set has an fcr grammar, any fcr grammar defines an RCF set.

Proof: (i) Since a regular form system is clearly equivalent to a very restricted fcr grammar any RCF set has an fcr grammar. We want to show that the restrictions are not effective.

(ii) In an fcr grammar there are rules of 5 types:

1. $A \to \lambda$
2. $A \to bc$
3. $A \to bC$
4. $A \to Bc$
5. $A \to BC$

Let us show that there exist an s'-grammar which defines the same set as any fcr grammar. And to do that we reduce an fcr grammar to an s'-grammar without modifying the language, just as we did for regular form systems. First define an equivalent system of equations:

- Determine all variables A such that $\delta(A) = \lambda$.

In the equation of A :

- To each rule $A \to bc$ corresponds a term bK , K a new variable, where $K = cL$, $L = \lambda$.

- To each rule $A \to bC$ corresponds a term bC .

- To each rule $A \to Bc$ corresponds a term BK .

- To each rule $A \to BC$ corresponds a term BC .

We have a system of equations of the form

$$A = \sum aB_a + \sum CD + \delta(A) .$$

In any monomial CK we substitute for the leftmost variable the expression which defines it: $C = \sum aC_a + \sum EF + \delta(C)$.

We do the same operation in any monomial EFD or D we may have obtained.

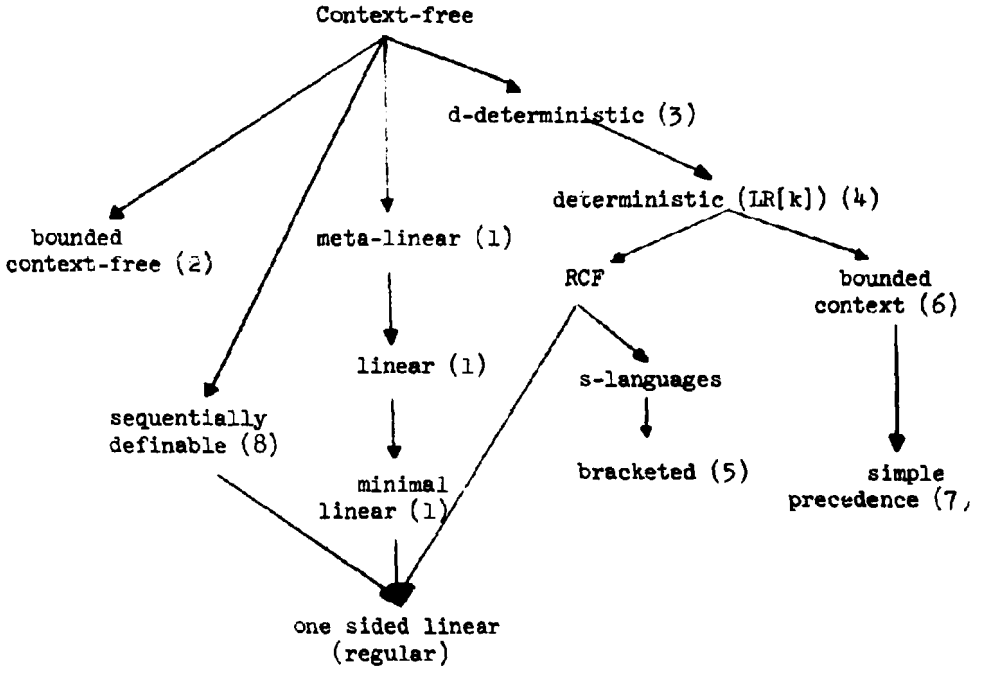
We keep doing this as long as we have in the equation of A some terms which do not begin by a terminal letter.

By Lemma 5.15, the process comes to an end since the number of variables is finite and we have eliminated non terminating recursions using Lemma 5.4.

We have not introduced new variables in this calculation and it is clear that separability is preserved: $\Pi(C,D) \wedge \Pi(E,F) \wedge (EF \subseteq C) \Rightarrow \Pi(F,D)$; thus, the resulting system is equivalent to an s'-grammar

because of the condition $(A \to BC) \wedge (A \to DE) \Rightarrow (\text{first}(BC) \cap \text{first}(DE) = \emptyset)$ for fcr grammars. ∎

The situation of RCF languages among other classes of context-free languages is depicted by the following graph:



Inclusion is proper along an arrow. (We shall see in the next section that the class of RCF languages is properly included in the class of deterministic languages.) The references are:

1. Chomsky [1963].

2. Ginsburg and Spanier [1964].

3. Hibbard [1966].

4. Ginsburg and Greibach [1965], Knuth [1965].

5. Ginsburg and Harrison [1966].

6. Floyd [1964].

7. Floyd [1963], Wirth and Weber [1965].

8. Ginsburg and Rice [1962].

We are now going to study the properties of RCF languages and, to begin with, we shall delimit the field by some negative results.

e) Negative Properties of RCF Languages

Theorem 5.16: It is undecidable whether a context-free language is RCF.

Proof: Following the method of the proof of Theorem 6.1 in Bar-Hillel, Perles and Shamir [1961], we construct a class of context-free languages by which we map the set of solutions to the Post correspondence problem (Post [1946]) onto an RCF language, T^*, and the non-solution onto non RCF context-free languages.

Given $n \in \Omega$, two finite sets $A = \{\alpha_1,\ldots,\alpha_n\}$, $B = \{\beta_1,\ldots,\beta_n\}$, a symbol $\circledast \notin T$ and a finite set $S = \{b_1,\ldots,b_n\}$, $b_i \notin T\ (\forall i)$; consider the three following languages, functions of A, B and n.

$$L_1 = \{b_{i_1} \ldots b_{i_p}\ \alpha_{i_p}^R \ldots \alpha_{i_1}^R \circledast \beta_{j_1} \ldots \beta_{j_q} b_{j_q} \ldots b_{j_1} \mid p,q \in \Omega\}$$

$$L_2 = \{\alpha^R \circledast \alpha \mid \alpha \in (T+S)^*\}$$

$$L_3 = L_1 \,\&\, L_2 .$$

L_1 and L_2 are context-free and more precisely minimal linear (Chomsky [1963]), their complements, L_1' and L_2' are context-free

too (more precisely linear and unambiguous). (Chomsky and Schützenberger [1963], Theorem 3, page 141.)

If there is no solution to Post correspondence problem for A , B and N , $L_3 = \emptyset$; otherwise L_3 is not context-free (Bar-Hillel et al.).

Consider its complement $L_3' = L_1' + L_2'$.

L_3' is context-free as the union of two context-free languages (ibid.). When $L_3 = \emptyset$, $L_3' = T^*$ and is RCF. When $L_3 \neq \emptyset$, L_3' cannot be RCF because then it would be deterministic as any RCF language is and its complement L_3 would be deterministic too (Schützenberger [1963], Ginsburg and Greibach [1965]) thus, a fortiori, context-free.

Since the Post correspondence problem is unsolvable for $\text{card}(T) \geq 2$ there is no algorithm to decide whether a set L_3' , a function of A , B and n is RCF; this implies the theorem. ∎

Corollary 5.17: It is undecidable whether a context-free language is equal to a given RCF language.

Proof: Since it is undecidable whether $L_3' = T^*$. ∎

Theorem 5.18: It is undecidable whether the intersection of two RCF languages is empty.

Proof: The proof is very similar to the proof of Theorem 18 in Rabin and Scott [1959] or Theorem 5 in Landweber [1964].

Given as before $n \in \Omega$, $A = \{\alpha_1,\ldots,\alpha_n\}$, $B = \{\beta_1,\ldots,\beta_n\}$, $S = \{b_1,\ldots,b_n\}$, $b_i \notin T$ $(\forall i)$;

Let $\begin{cases} L_1 \to b_i L_i A_i \\ L_1 \to \lambda \\ A_i \to \alpha_i \end{cases}$ $i = 1,\ldots,n$ $\begin{cases} L_2 \to b_j L_2 A_j \\ L_2 \to \lambda \\ A_j \to \alpha_j \end{cases}$ $j = 1,\ldots,n$

These are two s'-grammars, as it is easily verified.

Their intersection is empty if and only if the Post correspondence problem for this particular A , B and n has no solution. This proves the theorem. ∎

Corollary 5.19: It is undecidable whether the intersection of two RCF languages is RCF or not, regular or not.

Proof: $\emptyset$ is an RCF language and a regular set, and it is decidable, as we have seen, whether an RCF language (more generally any context-free language) is empty. Thus Theorem 5.18 implies the corollary. ∎

This leads one naturally to ask whether the class of RCF languages is closed under intersection.

Lemma 5.20: The class of RCF languages is not closed under intersection, union, concatenation or closure.

Proof: Consider the following RCF languages defined by regular form systems. (Quantities equal to $\emptyset$ not written.)

$L_1 = AC$ $\qquad$ $L_2 = AB$

$A = aB + \lambda$ $\qquad$ $A = aA + \lambda$

$B = AB_1$ $\qquad$ $B = bB_1 + \lambda$

$B_1 = bL$ $\qquad$ $B_1 = BC$

$L = \lambda$ $\qquad$ $C = cL$

$C = cC + \lambda$ $\qquad$ $L = \lambda$

$L_1 = \{a^n b^n c^* | n \in \Omega\}$ $\qquad$ $L_2 = \{a^* b^n c^n | n \in \Omega\}$

L_1 & $L_2 = \{a^n b^n c^n | n \in \Omega\}$ is not context free (Bar-Hillel, Perles and Shamir [1961], Scheinberg [1960]) a fortiori not RCF.

$$L_1 + L_2 = \{a^n b^p c^q | n,p,q \in \Omega \wedge (p = n \vee p = q)\}$$

is inherently ambiguous (Parikh [1961]), thus it cannot be RCF since RCF languages have grammars yielding deterministic parses, as we have seen in Sections 5b and c.

Let $L_3 = dL_1 + L_2$, L_3 is RCF, a regular form system defining it is immediately obtained from those defining L_1 and L_2.

Let $L_4 = d + \lambda$, L_4 is finite thus RCF.

$L_4 \cdot L_3 = ddL_1 + d(L_1 + L_2) + (L_1 + L_2)$ is clearly inherently ambiguous, thus cannot be RCF.

Let $L_5 = \{da^n b^n a^* | n \in \Omega\} + \{a^* b^n a^n | n \in \Omega\}$. L_5^* is not deterministic (Ginsburg and Greibach [1965], page 33) thus not RCF, while L_5 clearly is. ∎

As we shall see next, the language $\{b^n(a^n + c^n) | n \in \Omega\}$ is another example of a non RCF set which is the union of two RCF sets.

We leave unresolved the question of closure under complementation. As one could expect from the role of a left to right parse in the definition of RCF languages:

Lemma 5.21: The class of RCF languages is not closed under reversal.

Proof: $L_1 = \{(a^n + c^n)b^n | n \in \Omega\}$ is RCF:

$$L_1 = aB + cB + \lambda$$
$$B = L_1B_1$$
$$B_1 = bL$$
$$L = \lambda$$

is a regular form system which defines it.

$L_1^R = \{b^n(a^n + c^n) | n \in \Omega\}$ is not RCF because it cannot be recognized by an s'-machine.

The machine cannot predict whether it will meet a string of a 's or c 's when counting the b 's; so that when counting the a 's or c 's, it must remember somehow either what the first non- b letter was or what the last read-in letter is. It cannot use its finite state part to do this and if it used the pushdown store it would have to destroy that information before it could use it. ∎

Note that $L_1^R = \{b^n(a^n + c^n) | n \in \Omega\}$ is obviously a deterministic language so that the inclusion of RCF languages into deterministic languages is proper.

gsm-mappings (Ginsburg and Rose [1963]) are often useful in proving results about context-free languages.

Lemma 5.22: The class of RCF languages is not closed under gsm-mapping.

Proof: The set $L_1 = \{\alpha \otimes \alpha^R | \otimes \notin T\}$ is RCF as generated by the s'-grammar

$$\left.\begin{array}{l} L_1 \rightarrow aL_1A_a \\ A_a \rightarrow aL \end{array}\right\} \quad \text{for all } a \in T$$

$$L_1 \rightarrow \otimes L$$

$$L \rightarrow \lambda$$

The set $L_2 = \{\alpha\alpha^R\}$ is not RCF because it is not deterministic.

L_2 is obtained from L_1 by the gsm-mapping which erases $\otimes$ and maps all other symbols onto themselves. ∎

This implies that the class of RCF languages is not closed under sequential transduction, of which gsm-mappings are the deterministic case.

This implies also that the class of RCF languages is not closed under substitution of an RCF set for a given letter or substring; here substitution of $\{\lambda\}$ for $\{\otimes\}$. A more interesting example is the following highly pathological non-deterministic context-free language: $\{a^m a^n b a^n a^p b a^p a^m | m,n,p \in \Omega\}$ due to R. McNaughton. It can be obtained from $\{a^m c d a^m | m \in \Omega\}$ by obvious substitutions. It cannot be parsed by the classical methods from left to right, right to left nor even both ends inward and it is unambiguous.

Another negative property comes up naturally, as we shall see in Section 5g, in the study of the application of RCF languages to Algol 60; as the preceding lemma, it disallows the application to RCF languages

of techniques frequently useful to prove results on context-free languages such as precisely the machine mapping theorem (Ginsburg and Rose [1963]).

Lemma 5.23: The class of RCF languages is not closed under intersection with a regular set.

Proof: Let $L_1 = \{[^n a]^n a + [^n b]^n b \mid n \in \Omega\}$

$L_2 = [^* a]^* a + [^* b]^* b$

$L_3 = L_1 + \{[^n a]^n b + [^n b]^n c \mid n \in \Omega\}$

L_2 is regular.

L_3 is RCF as defined by the following s-grammar:

$L_3 \rightarrow aC$

$L_3 \rightarrow bC$

$C \rightarrow a$

$C \rightarrow b$

$L_3 \rightarrow [L_4 BC$

$B \rightarrow]$

$L_4 \rightarrow [L_4 B$

$L_4 \rightarrow a$

$L_4 \rightarrow b$

$L_1 = L_3 \,\&\, L_2$.

L_1 is not RCF because it cannot be recognized by an s'-machine: there is no way by which the machine could save the information that the bracketed character is for instance an "a" for matching against

the terminal character, because it cannot save it in its finite state controller and if it puts it on the pushdown store it will have to erase it to verify the bracket matching. ∎

Note that the class of deterministic languages is closed under intersection with a regular set. (Ginsburg and Greibach [1965], Theorem 3.1.)

The positive results we are going to give in Section 5g are unclassical, let us give here a classical one:

Lemma 5.24: The class of RCF languages is closed under the operation of derivation with respect to a string.

Proof: It is enough to show that it is closed under derivation with respect to a letter.

Let S be the root, "a" the given letter.

Consider the rules having S as a left part; among them erase those the right part of which does not begin by "a" and erase all the rules connected only to erased rules.

If no rule remain, $D_aA = \emptyset$. If one rule with S on the left remains, it is of the form

$$S \to aB_1 \ldots B_n$$

We replace it by the rules obtained by erasing a and substituting for B_1 any expression $cC_1 \ldots C_p$ where we had $B_1 \to cC_1 \ldots C_p$ in the old grammar:

$$S \to cC_1 \ldots C_p B_2 \ldots B_n \quad .$$

We simplify any possible occurence of two identical production rules.

We still have an s'-grammar since substitution preserves separability. ∎

Noting that $\alpha \in X \Leftrightarrow \delta(D_\alpha(X)) = \lambda$, we see that the proof of this Lemma is exactly patterned after the recognition by an s'-machine, as can be understood best if one thinks of the definition of an automaton by Nerode equivalence classes.

f) Axiomatic of Context-free Grammars

We are now in a position to prove that no complete axiom system for the equality of context-free grammars can exist.

The idea behind the proof is simple: context-free languages are recursive; T^* is enumerable; thus given two context-free grammars G_1 and G_2 defining the sets L_1 and L_2, we can enumerate the strings in T^* one by one, verifying each time whether they do or do not belong to both L_1 and L_2. We stop at the first string which belongs to one and not to the other one. Thus we have a trivial semi-decision procedure $\mathfrak{f}$ for the inequality of context-free grammars. On the other hand there is no decision procedure for the equality of context-free grammars (Bar-Hillel, et al. Theorem 6.3); at best there could be a semi-decision procedure $\mathfrak{f}'$. But then $\mathfrak{f}$ and $\mathfrak{f}'$ taken together would form a decision procedure for the equality of context-free grammars; thus $\mathfrak{f}'$ cannot exist; this implies that no complete axiom system for the equality of context-free grammars can exist either.

We see that to make this proof formal we must exhibit an algebra in which $\mathfrak{L}$ can be described; in other words an algebra with a complete axiom system for the inequality of context-free grammars. This is just what we have been doing so far.

Recall that after having proved that any RCF set has an s'-grammar (Theorem 5.8) we noted that with minor modifications, the proof could be turned into a constructive algebraic proof of the existence of a pre-standard-form grammar for any context-free language; this being done using only relations in $\langle RE,R1,R2\rangle$ and substitutions of equals for equals.

Let us write

(d1) $\quad \mathfrak{L}_a(X) = \mathrm{rest}(aT^* \,\&\, X)$

(d2) $\quad \mathfrak{L}_\alpha(X) = \text{if } \alpha = \lambda \text{ then } X \text{ else } \mathfrak{L}_{\mathrm{first}(\alpha)}(X)$

(d3) $\quad \mathrm{rest}(\sum_{i=1}^{n} a_i Y_i) = \sum_{i=1}^{n} Y_i \quad .$

Since the inclusion symbol does not belong to our algebra we replace $\alpha \in X$ by $\delta(\mathfrak{L}_\alpha(X)) = \lambda$.

Recall that in $\langle RE,R1,R2\rangle$ we have the rules

(i3) $\quad \lambda \,\&\, xA = \emptyset$

(R2) $\quad \dfrac{x \neq y}{xA \,\&\, yB = \emptyset}$

with the Boolean relations

(b6) $\quad A \,\&\, (T^*) = A$

(b5) $\quad A + \emptyset = A \quad .$

We see now that with these rules and with (d1), (d2), and (d3), we can formalize the process described in the proof of Lemma 5.24 to the point where it defines algebraically an algorithm to verify whether $\delta(\$_\alpha(X)) = \lambda$ and we see that we can do this for any pre-standard-form grammar.

Thus <RE,R1,R2> together with (d1), (d2), and (d3) form a complete axiom system for the inequality of context-free grammars.

Although, as we have just proven, no complete axiom system for the equality of context-free grammars can exist, it is reasonable to look for incomplete but practically sufficient ones. Let us consider the following relations:

(d4) $\quad \text{first}(\sum_{i=1}^{n} a_i Y_i) = \sum_{i=1}^{n} a_i$

(d5) $\quad X \subseteq \text{first}(X)\text{rest}(X) \quad$ for any $X \neq \lambda$.

We believe that <RE,R1,R2> together with (d1), (d2), (d3), (d4), and (d5) are such a system.

We would like to acknowledge the help of J. Friedman in establishing the non-existence of a complete axiom system.

g) Cancellation, Regularity and Equality

The results which follow shed much light on the algebraic nature of separability. The first corollary is a generalization of Lemma 12 in Korenjak and Hopcroft [1966].

Theorem 5.25:

$$AX = BX \wedge X \neq \emptyset \wedge \Pi(A,X) \Rightarrow A \subseteq B \quad .$$

Proof: We show that $A \not\subseteq B \wedge AX = BX \wedge X \neq \emptyset \wedge \Pi(A,X)$ cannot hold true.

Let ξ be a shortest (possibly null) string in X. Suppose there is at least one $\alpha \in A$, $\alpha \notin B$; we can write

$\alpha \in A \Rightarrow \alpha\xi \in AX \Rightarrow \alpha\xi \in BX \Rightarrow$ (refinement rule and string cancellation)

$$(\exists\beta_1,\xi_1)[\alpha = \beta_1\xi_1 \wedge \xi_1 \neq \lambda \wedge \beta_1 \in B \wedge \xi_1\xi \in X]$$

$\beta_1 \in B \Rightarrow \beta_1\xi \in BX \Rightarrow \beta_1\xi \in AX \Rightarrow$

(as before) $(\exists\alpha_1,\xi_2)[\beta_1 = \alpha_1\xi_2 \wedge \alpha_1 \in A \wedge \xi_2\xi \in X]$.

Thus we have $\alpha = \alpha_1\xi_2\xi_1$, with $\alpha \in A$ and $\alpha_1 \in A$ while $\xi_2\xi \in X$ and $\xi_1\xi \in X$, $\xi_1 \neq \lambda$.

This is not compatible with $\Pi(A,X)$: ξ_2 is an initial substring of X (or if $\xi_2 = \lambda$, ξ_1 is); $\alpha_1\xi_2$ an initial substring of A ($\alpha_1\xi_1$ if $\xi_2 = \lambda$) and α_1 a string in A. ∎

Corollary 5.26: (right cancellation).

$$AX = BX \wedge X \neq \emptyset \wedge \Pi(A,X) \wedge \Pi(B,X) \Rightarrow A = B \quad .$$

The proof is immediate.

Theorem 5.27:

$$XA = XB \wedge X \neq \emptyset \wedge \Pi(X,A) \Rightarrow A \subseteq B \quad .$$

Proof: We show that $A \not\subseteq B \wedge XA = XB \wedge X \neq \emptyset \wedge \Pi(X,A)$ cannot hold true as before, except that this case is simpler.

Let ξ be a shortest non null string in X.

Let $\alpha \in A$. $\alpha \notin B$.

$\xi\alpha \in XA \Rightarrow \xi\alpha \in XB \Rightarrow$ (refinement rule and string cancellation)

$(\exists \xi_1, \beta_1)[\alpha = \xi_1\beta_1 \wedge \xi_1 \neq \lambda \wedge \beta_1 \in B \wedge \xi\xi_1 \in X]$.

Thus we have $\xi_1\beta_1 \in A$ and $\xi\xi_1 \in X$ with $\xi \in X$, which implies $\Pi(X,A) =$ <u>false</u> . ∎

<u>Corollary</u> 5.28: (left cancellation)

$$XA = XB \wedge X \neq \emptyset \wedge \Pi(X,A) \wedge \Pi(X,B) \Rightarrow A = B \quad .$$

The proof is immediate.

<u>Corollary</u> 5.29: The equation $A = XB$ (resp. $A = BX$) subject to $\Pi(X,B)$ (resp. $\Pi(B,X)$) cannot have more than one solution.

The proof is immediate.

An obvious particular case of 5.28 is often used in the form $\alpha A = \alpha B \Rightarrow A = B$, which we have derived in Corollary 3.8.

The following corollary of Corollary 5.26 is a generalization of what Korenjak and Hopcroft call a type B replacement.

Suppose that from two systems of equations having the same solution we have a derivable equality $X_1 \ldots X_m = Y_1 \ldots Y_n$, $m,n \geq 2$, where $Y_1, \ldots, Y_n$ are all different from $\emptyset$ and such that $\Pi(Y_i, Y_{i+1})$,

$i = 1,\ldots,n-1$. Let α be some non-null string in X_1 . α is certainly an initial substring of a string derivable from $Y_1\ldots Y_p$: $\alpha X_2\ldots X_m = \alpha Z_\alpha Y_k\ldots Y_n$ for some set Z_α and some $k > 1$.

Corollary 5.30: With the above hypothesis and notations,

$$X_1\ldots X_m = Y_1\ldots Y_n \Leftrightarrow \begin{cases} X_2\ldots X_m = Z_\alpha Y_k\ldots Y_n & k > 1 \\ X_1 Z_\alpha = Y_1\ldots Y_{k-1} \end{cases} .$$

Proof: (i) $\Rightarrow$ part.

$$X_1\ldots X_m = Y_1\ldots Y_n \Rightarrow \alpha X_2\ldots X_m = \alpha Z_\alpha Y_k\ldots Y_n \Rightarrow X_2\ldots X_m = Z_\alpha Y_k\ldots Y_n$$

first equality.

$$\begin{cases} X_2\ldots X_m = Z_\alpha Y_k\ldots Y_n \\ X_1\ldots X_m = Y_1\ldots Y_n \end{cases} \Rightarrow X_1\ldots X_m = X_1 Z_\alpha Y_k\ldots Y_n = Y_1\ldots Y_{k-1}\ldots Y_n \Rightarrow$$

(corollary 5.24) $X_1 Z_\alpha = Y_1\ldots Y_{k-1}$

second equality.

(ii) $\Leftarrow$ part.

$$\begin{cases} X_2\ldots X_m = Z_\alpha Y_k\ldots Y_n \\ X_1 Z_\alpha = Y_1\ldots Y_{k-1} \end{cases} \Rightarrow X_1 X_2\ldots X_m = X_1 Z_\alpha Y_k\ldots Y_n \Rightarrow$$

$$X_1\ldots X_m = Y_1\ldots Y_{k-1} Y_k\ldots Y_n \quad . \qquad \blacksquare$$

In the particular case where $X_1\ldots X_m = Y_1\ldots Y_n$ is derivable from two s'-2-grammars, we can write $Z_\alpha = Z_1\ldots Z_j$, $j \leq |\alpha|+1$ and $Z_1,\ldots,Z_j$ variables of the second s'-2-grammar; given an equivalnece of

this type, we can always reduce it to a system of two equivalences where the left part contains at most a certain number of variables which depends upon the minimal length of the strings in X_1 .

We now consider equality and regularity questions.

Theorem 5.31: A connected s'-2-grammar in which no variable is equal to $\emptyset$ defines an s-language if and only if it reduces to an s-grammar in standard-2-form by elimination of all variables equal to λ .

Proof: The only formal difference between s'-2-grammars and s-grammars in standard-2-form is in the occurence of rules $A \to \lambda$ in the former.

Consider an s-language A_1 and an RCF language D_1 given by two connected systems of equations corresponding to such grammars:

$$A_j = \sum_{a \in T} a B_{ja} C_{ja} \qquad j = 1,\ldots,m$$

where possibly B_{ja} or $C_{ja} = \lambda$ or $\emptyset$

$$D_i = \sum_{a \in T} a E_{ia} F_{ia} + \delta(D_i) \qquad i = 1,\ldots,n$$

same remark as above.

We want to show that $A_1 = D_1 \Rightarrow (\forall i)[\delta(D_i) = \emptyset \vee D_i = \lambda]$;

$A_1 = D_1 \Rightarrow \delta(A_1) = \delta(D_1) = \emptyset$. It is true for $i = 1$.

$$A_1 = D_1 \Rightarrow (\forall a)[aT^* \,\&\, A_1 = aT^* \,\&\, D_1] \Rightarrow$$
$$(\forall a)[B_{ja} C_{ja} = E_{ia} F_{ia}] \Rightarrow$$
$$(\forall a)[\delta(B_{ja} C_{ja}) = \delta(B_{ja})\delta(C_{ja}) = \delta(E_{ia} F_{ia}) = \delta(E_{ia})\delta(F_{ia})] \Rightarrow$$

$\delta(D_i) = \emptyset$ for all D_i implied here unless some $B_{ja}C_{ja} = \lambda$,

in which case we must have the corresponding $E_{ia}F_{ia} = \lambda$.

By substituting in the same way for all B_{ja} and E_{ia} the expressions which define them we introduce new equalities, new variables, and get $\delta(D_i) = \emptyset$ unless $D_i = \lambda$, for new values of i .

Suppose that a variable D_q is not reached in this process. Because we suppose the grammar connected and because an s'-grammar cannot be ambiguous, $(\forall\gamma)(\exists\alpha)(\exists\beta)[\gamma \in D_q \Rightarrow \alpha\gamma\beta \in D_1]$ and D_q has to be used in the derivation of $\alpha\gamma\beta$.

At step n of the substitution process we have outlined, all strings of length smaller than n and all initial segments of length n of longer strings in D_1 will have been produced.

Thus at step $n = |\alpha\gamma|$, D_q will have been reached, unless $D_q = \emptyset$; contrarily to the hypothesis. ∎

Corollary 5.32: It is decidable whether an RCF set is an s-language.

Proof: By Theorem 5.8, Corollary 5.9, Theorem 5.16, we know how to construct an s'-2-grammar for an RCF set defined in another way. ∎

Corollary 5.33: The equality problem between RCF sets and s-languages is solvable.

Proof: Since Korenjak and Hopcroft [1966] have shown the equality problem for s-languages to be solvable and since we have just seen that, given an RCF set, it can be decided whether it is an s-language and at the same time an s-grammar can be constructed for it, if it actually is one. ∎

Corollary 5.34: The equality problem between RCF sets and regular sets is solvable.

Proof: Given an RCF set A, the set $A@$ with $@$ a symbol not in T, is RCF. (Tag $@$ to the first rules of an s'-grammar.) This is the corresponding set with end-marker $@$.

Given a regular set R, $R@$ is regular on $T \cup \{@\}$ and it can be proved that it is always an s-language (ibid. Lemma 4).

$$A@ = R@ \Leftrightarrow A = R \quad .$$

The corollary follows by Corollary 5.31. ∎

Note that this result yields another proof of Corollary 5.17 since it is undecidable whether a context-free language is a regular set (Bar-Hillel, Perles and Shamir [1961]).

Corollary 5.35: An s'-grammar in which no variable is equal to $\emptyset$ and only one to λ, defines a regular set if and only if it is non self-embedding.

Proof: (i) If an s'-grammar defines a regular set R, the s'-grammar obtained by tagging to the rules of the root a new variable $A = @$, end marker, defines $R@$. By Corollary 5.9 it can be reduced to an s'-2-grammar without alteration of self-embedding. By Theorem 5.31 the elimination of all variables equal to λ reduces it to an s-grammar in standard-2-form which defines the regular set $R@$. By Corollary 2.1 in Korenjak and Hopcroft this s-grammar cannot be self-embedding. This implies that the original one cannot either.

(ii) Any non self-embedding context-free grammar defines a regular set (Chomsky [1959a, b]). ∎

Note that Corollary 5.35 implies that we can eliminate all variables in an s'-grammar defining a regular set and effectively compute a regular expression representing that set. This is obviously true also of regular form systems since we go from them to s'-grammars by chains of substitutions.

Note also that this corollary yields a direct proof of Theorem 5.16, since it is undecidable whether a context-free language is regular.

We have not been able to solve the equality nor the containment problems for s'-languages, nor to show that they are not solvable. At this writing, the class of RCF languages is the most general class of languages for which such solvability results as Corollaries 5.32, 3, and 5 have been obtained. Theorem 5.1 of Ginsburg and Greibach [1965] implies Corollary 5.34 but the proof of 5.34 is constructive and quite simple.

It is interesting to note that the solution to the equality problem of s-languages reduces precisely to the Salomaa's algorithm which we have derived from Theorem 3.12 for the equality problem of regular sets. Because of Lemma 5.17 though, it seems that the same approach which yields the simple formulation of Theorem 3.12 could not be used for s-languages without difficulties.

As for RCF languages, corollaries 5.26, 28, and 29 show us that in practical cases the equality of two RCF languages will be verifiable on the minimal regular form systems which define them.

h) Application of RCF Languages

We have four characterizations of RCF languages, as sets

(i) recognizable by regular expression techniques used recursively, (i.e., definable by a regular form system),

(ii) having an s'-grammar,

(iii) recognizable by an s'-machine,

(iv) having an fcr grammar.

The two first definitions correspond to two analysis techniques which in most cases are radically different; these differences illustrate a trade-off between speed and space, in the form of a trade-off between the use of the finite-state control of a PDA and the use of its pushdown store: the first one of these techniques makes as little use of the pushdown store as possible and is extremely fast, the other one, which corresponds to an automaton-theoretic characterization, uses as small a finite state control as possible and uses the pushdown store constantly. For an actual implementation, the first one is faster and corresponds to a more convenient notation. This is an example of the fact that automaton-theoretic characterizations of sets of strings must be understood as models of their computational complexity and not as programming strategies, even when the automaton is deterministic.

We have seen that a generalization of Algol 60 is within the scope of the method; is Algol 60 an RCF language?

There is an Algol 60 context-free construct which bars it from being RCF: <conditional expression> .

The intermediary strings containing any number n of matching parentheses:

...then $(^n$ <boolean expression>$)^n$ else <boolean expression>...

or

...then $(^n$ <arithmetic expression>$)^n$ else <arithmetic expression>...

are well formed; but

...then $(^n$<boolean expression>$)^n$ else <arithmetic expression>...

and

...then $(^n$<arithmetic expression>$)^n$ else <boolean expression>...

are not.

As we have seen in the proof of Lemma 5.23, <conditional expression> cannot be an RCF set, although it is the intersection of an RCF set and a regular set.

This seems to be the sole difficulty as far as the formal syntactical definition of Algol 60 is concerned. Otherwise RCF languages seem to be an insufficient model of Algol 60 for the same reasons that context-free languages are insufficient too.

If we cannot give a more precise answer to the question, it is in part because the definition of Algol 60 is not fully formalized, even as far as its context-free grammar in BNF (Naur [1963]) is concerned, and also because this grammar is ambiguous, sometimes deliberately.

e.g.: (2.6.1 in Naur [1963])

<proper string> ::= <any sequence of basic symbols not containing ' or '>|<empty>

<open string> ::= <proper string>|'<open string>'|<open string> <open string>

<string> ::= '<open string>'

This definition of <string> is ambiguous (not deliberately) and the variable <any sequence ...> is defined only by the English meaning of its name (somewhat ambiguous itself), which is naturally intended to denote the regular set ('+')' .

In this case the difficulties are easily lifted; for discussions of the intricacies and the ambiguities, deliberate or not, of the definition of Algol 60, see Knuth and Merner [1961], Knuth [1965 a, b page 624], Medema [1965].

We are going to study the ways by which we can enrich the formalism of RCF sets.

SECTION 6

EXTENSIONS OF RCF LANGUAGES

a) Direction of Extension, Syntax and Semantics

It may seem natural to try to augment the class of RCF languages within the class of context-free languages; for instance we could consider the class of intersections of RCF languages and regular sets; this broader class would correspond to an extension of s'-machines with a larger finite state control where only certain states could interact with the pushdown; we might also want to study left-right top-down deterministic analysis with more than one character look-ahead and in effect we might try to parallel for left-right top-down analysis Knuth's work for bottom-up analysis.

Although such research topics would certainly be of high theoretical interest and seem quite feasible, we want to leave them as proposals and we want to argue that in practice other avenues for extension must be sought. Our argument will apply as well, mutatis mutandis, in the case of bottom-up analysis, to simple precedence languages as opposed to higher order precedence languages or LR(k) languages.

If one wants to use an RCF language analyzer to recognize the <conditional expression> of Algol 60, all one should do is to set a flag when going through the first <boolean expression> or <arithmetic expression> and refer to it on encounter of the second one.

In the same way it is easy to recognize such a set as $\{b^n(a^n + c^n) | n \in \Omega\}$. Furthermore, nothing can prevent a programmer

from using here a counter or two. In this vein, the non context-free set $\{a^n b^n c^n | n \in \Omega\}$ is trivial to recognize.

As we use flags and counters we can as well use lists or tables. In effect, this is precisely what has to be done to check some constraints of programming languages which just cannot be expressed in a context-free grammar:

e.g.: - that an identifier is declared and just once in a block.
- that use end declaration agree (identifier types, array dimensions and bounds).
- that a label occurs only once in a block.
- that a go to statement refers to a label which occurs in its scope.

Because such a practice is simple and efficient there is no reason why it should be used only for non context-free constraints. In the case of RCF languages we believe that to try to extend the formalism within the class of context-free languages is not worth the extra effort, complication and corresponding loss in parsing speed, because these few features of programming languages which are context-free and not RCF can be analyzed by using the methods we have just mentioned and because most of the non-RCF features of programming languages are not context-free anyway.

So far we have carefully talked imprecisely about non context-free features or constraints; let us state now that these features are for us to be called syntactic and not semantic, contrary to what seems to be the spreading usage. This is not a point of negligible importance

because it is intimately connected with the way one thinks about the definition of artificial languages:

The tendency has been lately to call semantic whatever peculiarity of a programming language could not be described in the phrase structure system part of its definition. The belief that context-free languages offer a close model of the syntax of programming languages is not foreign to this. This tendency is encouraged by the observation that the verification of a non context-free syntactical constraint, such as existence of the label mentioned in a go to statement, is conveniently described in the same way as is described the action coupled with the analysis of that statement, such as code generation. Both are specified by associating one-to-one the production rules of the grammar and some procedures, which must be executed when the corresponding rule is applied. In the computing community the confusion has gone to the point where people would talk about the semantic of a language for the semantic of a compiler or conversely; true, they are related, since a compiler must be a semantic preserving operator, but not identical, as is quite clear when one notes that compilers do have bugs. Semantics can be precisely stated, for instance as proposed by Riguet [1962].

Note that this improper usage we are discussing puts the people who adopt it in an untenable position if they change formalism to describe the same language: what is for them semantic in the weaker formalism can be syntactic in the stronger one.

A programming language is not an object independent of its definition, or rather it should not be: it does not have existence

and we do not know it by anything else. If the formalism adopted for this definition is not subtle enough to describe it with all the desired fineness, a set of computable conditions on strings is necessarily added and described in some other metasyntactical language. In the case of Algol 60, BNF was insufficient and conditions were described in English in order to refine the BNF definition; these conditions were put under the heading "semantics", together with some broad descriptions of the action of an interpreter. This certainly was misleading.

One must distinguish between the relations among symbols defining the well-formedness of a string of the language and between the actions this string may induce when analyzed. The latter is ultimately a mapping of the language into some domain, the former belongs to the definition of a set. Because there are formalisms in which all constraints defining any recursive set can a priori be expressed, namely, Post's formal systems, a Turing-machine programming language, McCarthy's recursive functions of conditional expressions, that distinction is quite meaningful. Furthermore, it is coherent with the mathematical usage as fixed in the simple case of predicate calculus: well-formedness is not a semantic matter, as definitively discussed in the introduction of Church [1956], Section 09.

What appears, when we use flags, counters or tables in the course of analysis, is precisely computable conditions on strings, necessary to refine a formal definition made in a too weak metasyntactical language.

We want to formalize this approach; beforehand we will study another avenue for extension which is very general too, but has never been used or mentioned, we believe, in spite of its simplicity.

b) Boolean Closure of Recursive Classes of Languages

Lemma 6.1: The Boolean closure of a class of recursive languages is a class of recursive languages.

Proof: The proof is straightforward. Let us use the original definition of recursivity as in Post [1944]; although Post is writing about sets of integers, his work is relevant here: we could either arithmetize the problem, as is often done, or use Davis's reformulation of it for strings and restrict this formulation to recursivity rather than A-recursivity by taking $A = \emptyset$. (Davis [1958], Chapter 4).

(i) A set is recursive if and only if both it and its complement are recursively enumerable, (Post [1944], page 290); thus if a set is recursive, its complement is recursive.

(ii) If two sets are recursively enumerable, so is their union by the very definition of recursive enumerability.

(iii) The complement of the union is equal to the intersection of the complements.

This corresponds to the following intuitively obvious fact: suppose we have a recognition program, for our general purpose computers, which can recognize any language of a class of sets; for instance suppose we have a general analyzer for the class of all context-free languages such as the Harvard analyzer (Kuno and Oettinger [1963]); then we can easily build with it a recognition program for the Boolean closure of this class, since:

$$\alpha \in A \,\&\, B \;\Leftrightarrow\; (\alpha \in A) \wedge (\alpha \in B)$$

$$\alpha \in A + B \;\Leftrightarrow\; (\alpha \in A) \vee (\alpha \in B)$$

$$\alpha \in A' \;\Leftrightarrow\; \alpha \notin A \quad .$$

Note that the Boolean closure of context-free languages is a larger class in Θ than the class of context-free languages; as proved by Kuroda [1964] it is included in the class of deterministic context-sensitive languages (sets recognizable by a deterministic linear bounded automaton). In fact, the classical example of a useful context-sensitive language which is not context-free is the intersection of two RCF sets: (so-called "respectively construct") cf. proof of Lemma 5.20

$$\{a^n b^n c^n | n \in \Omega\} = \{a^n b^n c^p | n,p \in \Omega\} \,\&\, \{a^r b^q c^q | r,q \in \Omega\} \quad .$$

This calls for a few remarks which form Appendix 6.

In practice, this means that if someone finds it natural to define a language as the intersection of context-free sets, or complement of some context-free set, or any Boolean function, such as $A-B = A \,\&\, B'$, he should not refrain from doing so. As long as the sets, which he takes the complement or the intersection of, are independently defined, there is no difficulty. In fact we shall see in Section 6g that people actually do so, implicitly; this alone would justify the explicit introduction of & and ' .

However it is important that there are no two variables X_i and X_j in the system such that $X_i \# X_j$ while one is defined from the other one by use of & or ' . In such a case it can happen that the system has no solution.

e.g.: $S = S'$.

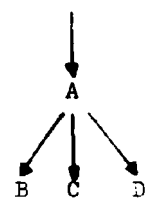
As remarked by M. Arbib who noted this difficulty, the situation is analogous to the one in which we obtain unstable circuits by assembling well-behaved components without timing constraints.

It is an interesting open problem to know when such systems have a solution; it is a generalization of the already difficult problem of knowing when a system of equations in a Boolean algebra of sets has a solution.

The main shortcoming of the use of + or & in the safe case, is a loss of speed, since the recognizer must be applied twice on the same string; on the other hand these applications are independent and can be made in parallel.

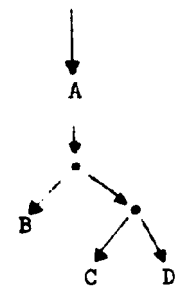
Another problem with the use of & and ' is in defining the structure of the analyzed string. This difficulty is removed if we consider that the structural tree of a string, as defined usually, does not reflect an inherent property of the string but rather describes the course of analysis, independent analysis phases being described by parallel branches in the tree. (cf. Section 4a.)

When the application of a rule $A \rightarrow BCD$ corresponds usually to 3 nodes in the structural tree:

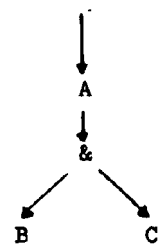


We associate to it 5 nodes in which the operators involved, here two concatenations, do appear. Let us adopt the convention that BCD

is read from left to right, so that BCD is understood as B(CD) rather than (BC)D :

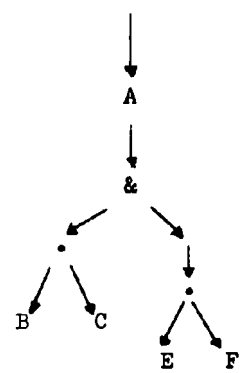


In the same way if a rule A → B & C is used, the substructure:



will appear in the development of the sentence tree structure.

e.g.: To A → (BC) & (EF)' corresponds



c) Conditicnal Regular Expressions

The methods we are going to use have been introduced in McCarthy [1963].

Let $\underline{b}_1,\underline{b}_2,\ldots,\underline{b}_n$ be a set of variables taking values in {true,false} when defined.

Their algebra $\mathcal{B}$ is defined in a slightly different manner than a Boolean algebra is. We introduce a ternary operator:

$$\text{if } \underline{b}_1 \text{ then } \underline{b}_2 \text{ else } \underline{b}_3$$

with the following valuation rule

$\underline{b}_1$	if $\underline{b}_1$ then $\underline{b}_2$ else $\underline{b}_3$
true	value ($\underline{b}_2$)
false	value ($\underline{b}_3$)
undefined	undefined

Such operators as $\wedge$, $\vee$, $\sim$ are redefined by

$$\underline{b}_1 \wedge \underline{b}_2 = \text{if } \underline{b}_1 \text{ then } \underline{b}_2 \text{ else } \underline{\text{false}}$$
$$\underline{b}_1 \vee \underline{b}_2 = \text{if } \underline{b}_1 \text{ then } \underline{\text{true}} \text{ else } \underline{b}_2$$
$$\sim \underline{b}_1 = \text{if } \underline{b}_1 \text{ then } \underline{\text{false}} \text{ else } \underline{\text{true}}$$

so that $\wedge$ and $\vee$ are no longer commutative since for instance we can have $\underline{b}_1 \wedge \underline{b}_2 = \underline{\text{false}}$ and $\underline{b}_2 \wedge \underline{b}_1$ undefined. This corresponds to a left to right evaluation scheme.

The family of all functions built with this ternary operator is studied in McCarthy [1963], a complete set of relations to manipulate

them is given and two canonical forms are derived in Section 7.

Note that we have two types of equivalence, weak and strong, $\underline{b}_1 =_w \underline{b}_2$, $\underline{b}_1 =_s \underline{b}_2$, according to whether $\underline{b}_1$ and $\underline{b}_2$ are equal only when both defined or furthermore have the same domain of definition.

Definition 6.2: $C(\mathcal{R})$, class of functions computable in terms of $\mathcal{R}$, is defined recursively from $\mathcal{R}$ by

(i) $E \in \mathcal{R} \Rightarrow E \in C(\mathcal{R})$

(ii) if $E_1 \in C(\mathcal{R})$, $E_2 \in C(\mathcal{R})$ and $\underline{b}_1$ is a variable in $\mathcal{B}$, then

$(E_1) \in C(\mathcal{R})$ $\quad E_1 + E_2 \in C(\mathcal{R})$ $\quad E_1^* \in C(\mathcal{R})$

$E_1 \,\&\, E_2 \in C(\mathcal{R})$ $\quad E_1' \in C(\mathcal{R})$

$E_1 \cdot E_2 \in C(\mathcal{R})$

if $\underline{b}_1$ then E_1 else $E_2 \in C(\mathcal{R})$.

(iii) Extremal clause: $E \in C(\mathcal{R})$ only if E can be formed by a finite number of applications of rules (i) and (ii).

The interpretation of expressions of $C(\mathcal{R})$ is defined in the obvious way to be coherent with the interpretation of $\mathcal{R}$ (3a);

value (if $\underline{b}_1$ then A else B) = if $\underline{b}_1$ then value(A) else value(B) .

Let us adopt the convention that "if" and "else" have the lowest priority above parentheses in evaluation, so that for instance:

$(A + \text{if } \underline{b}_1 \text{ then } B \text{ else } C + D) = (A + (\text{if } \underline{b}_1 \text{ then } B \text{ else } (C+D)))$

$\text{if } \underline{b}_1 \text{ then } B \text{ else } C^* = \text{if } \underline{b}_1 \text{ then } B \text{ else } (C^*)$

$\text{if } \underline{b}_1 \text{ then } \underline{b}_2 \text{ else } C = D \bullet \text{if } \underline{b}_1 \text{ then } \underline{b}_2 \text{ else } (C = D)$

"if" and "then" , "then" and "else" , are used as brackets.

We call these expressions conditional regular expressions and denote them by upper case letters since they take values in Θ .

When they do not contain & or ' we call them restricted.

Note that while a regular expression designates a set, a conditional one varies over different sets according to the values taken by the $\underline{b}_i$ variables in them; precisely, it varies over the set of vertices of a hypercube of regular sets.

There are two types of equalities in $C\{R\}$, strong ones and weak ones, as in $\mathcal{B}$, according to whether the domains of definition of two expressions coincide or not. We write $E_1 =_s E_2$ or $E_1 =_w E_2$.

At this point it is clear that we can recognize the equalities of conditional regular expressions, derive canonical forms for them and get a complete set of axioms for their algebra. Let us do it briefly, before considering systems of equations in $C\{R\}$.

d) Foundations of the Algebra of Conditional Regular Expressions

We seek a complete set of axioms for the algebra $\langle\Theta;+,\&,',\cdot,*,\text{if-then-else}\rangle$. The following 12 rules are clearly valid according to the valuation mapping we have just defined:

(1) $(\text{if } \underline{b}_1 \text{ then } A \text{ else } A) =_w A$

(2) $(\text{if } \underline{\text{true}} \text{ then } A \text{ else } B) =_s A$

(3) (if <u>false</u> then A else B) $=_s$ B

(4) (if $\underline{b}_1$ then
if $\underline{b}_1$ then A else B
else C) $=_s$ (if $\underline{b}_1$ then A
else C)

(5) (if $\underline{b}_1$ then A else
if $\underline{b}_1$ then B else C) $=_s$ (if $\underline{b}_1$ then A
else C)

(6) (if $\underline{b}_1$ then $\underline{b}_2$ else $\underline{b}_3$ then A else B) $=_s$
(if $\underline{b}_1$ then
if $\underline{b}_2$ then A else B
else if $\underline{b}_3$ then A else B)

(7) (if $\underline{b}_1$ then
if $\underline{b}_2$ then A else B
else if $\underline{b}_2$ then C else D) $=_s$
(if $\underline{b}_2$ then
if $\underline{b}_1$ then A else C
else if $\underline{b}_1$ then B else D)

These first 7 rules permit us to handle the nesting of if-then-else's and to do a few simplifications. We now need 5 rules to permit us to handle the nesting of an if-then-else within the scope of a + , & , ' , · , or * .

To avoid repeating the same long rule 3 times, let ⊙ denote a binary operator; consider the folowing predicate, function of ⊙ :

Distrib(⊙) = [((if $\underline{b}_1$ then A else B) ⊙ (if $\underline{b}_2$ then C else D)) $=_s$

(if $\underline{b}_1 \wedge \underline{b}_2$ then A⊙C else

if $\sim \underline{b}_1 \wedge \underline{b}_2$ then B⊙C else

if $\underline{b}_1 \wedge \sim \underline{b}_2$ then A⊙D else B⊙D)]

(8) Distrib(+)

(9) Distrib(&)

(10) Distrib(·)

(11) (if $\underline{b}_1$ then A else B)' $=_s$ if $\underline{b}_1$ then A' else B'

(12) (if $\underline{b}_1$ then A else B)* $=_s$ if $\underline{b}_1$ then A^* else B^*

To these 12 basic rules we want to add another one, which must be considered as syntactic since we consider here that the equality is part of the syntax language.

(13) (if $\underline{b}_1$ then A else B) $=_w$ C •

(if $\underline{b}_1$ then A $=_w$ C else B $=_w$ C) .

Our 12 first rules come from McCarthy [1963], Section 7, somewhat indirectly. McCarthy studies $\mathcal{B}$; this is the reason why we do not have any rule corresponding to his rule 4: if $\underline{b}_1$ then $\underline{\text{true}}$ else $\underline{\text{false}} =_s \underline{b}_1$; but in fact he is killing two birds with one stone and defining rules which are valid, mutatis mutandis, for classes of functions computable in terms of any base algebra. This is quite clear in the notation of his rules and in his remark that the relation of functions to conditional forms is given by a distributive law, which we write here:

$$f(x_1,\ldots,x_{i-1}\ ,\ \text{if}\ \underline{b}_1\ \text{then}\ g\ \text{else}\ h\ ,\ x_{i+1},\ldots,x_n\ ) =_s$$
$$\text{if}\ \underline{b}_1\ \text{then}\ f(x_1,\ldots,x_{i-1},g,x_{i+1},\ldots,x_n)$$
$$\text{else}\ f(x_1,\ldots,x_{i-1},h,x_{i+1},\ldots,x_n)$$

where g and h are some expressions possibly conditional.

Our relations (1) to (7) come from his relations (1) to (8), with (4) omitted, and our relations (8) to (12) from this last one.

By using rules (8) to (12) we can put any conditional regular expression in a strongly equivalent form with the property that no if-then-else is within the scope of another operator; we can, loosely speaking, move all the + , & , · , ' and * 's within the if-then-else 's.

McCarthy's developments are applicable to any such constructs and our reader is now referred to McCarthy [1963], Section 7, to see how, by application of rules (1) to (7), we can now get two canonical forms, a weakly and a strongly equivalent one: Essentially this amounts to the classical disjunctive normal form of Boolean algebra; the expression is represented as the disjunction of the values it takes on the vertices of the hypercube where it varies. When strong equivalence is concerned some precautions must be taken, due to the non-commutativity.

Because these values are defined by regular expressions, and because we can recognize the equality of regular expressions, this solves the weak and strong equality problems of conditional regular expressions. We have proved the following theorem:

Theorem 6.3: Rules (1) to (12) for conditional regular expressions, together with system <RE,R1,R2> for regular expressions, form a

complete axiom system for conditional regular expressions.

We cannot directly use $\langle RE, R1, R2\rangle$ for conditional regular expressions without putting restrictions on the nature of the conditions: for instance R1 is no longer valid; suppose B = if $\underline{b}_1$ then D else F, we cannot write

$$\frac{\models \lambda \,\&\, B = \emptyset \;,\; \models A = BA + C}{\models A = B^{*}C}$$

because one could devise a $\underline{b}_1$ equal to *true* if B belongs to a monomial terminated by A and to *false* otherwise.

In the same way we cannot always write

$$A^{*} = \lambda + A + A^{2} + \ldots + A^{n}A^{*}$$

where A is a conditional regular expression, the condition could be on whether A has an exponent odd of even or is starred.

This difficulty corresponds to the fact that within an expression a condition $\underline{b}$ can refer to the form or the context of the expression.

Note that it is natural to associate to a conditional regular expression the regular expression which is the sum of the values it takes for all possible values of the conditionals and also to associate the regular expression which is the intersection of these values. We may call them respectively the *envelope* and the *center*. We will not use these notions here.

e) Recursive Functions of Regular Expressions

In $C(R)$, as we did in R , we consider systems S of recursive equations of the form

$$X_i = f_i(X_1,\ldots,X_n) \qquad i = 1,\ldots,n$$

f_i a conditional regular expression over $T+I$,

i.e., a recursive function of regular expressions.

Here also we first focus our attention on restricted f_i 's.

As we have just seen any f_i can equivalently be written as

$$\text{if } \underline{b}_1 \text{ then } g_1 \text{ else if } \underline{b}_2 \text{ then } g_2 \text{ else } \ldots g_p$$

where $g_1,\ldots,g_p$ are restricted regular expressions over $T+I$.

Replacing each f_i by its envelope we can associate to a set defined in that way its envelope which is a context-free language.

Related systems have already been used with different notations by some authors.

In Chomsky [1965] the use of "features" is advocated to solve certain vexing problems in the description of natural languages, or rather of the native speaker behaviour, by transformational grammars: To each terminal symbol is attached an array of Boolean variables which specify binary features of that symbol or of its syntactical usage; for instance "boy" is a name, designating something human and animate, "to laugh" can have such things for subject; just as for instance in Slagle's DEDUCOM (Slagle [1965]) to each object is attached its property list. The base grammar derivations are to be made dependent

upon these features, so that sentences such as "The harvest was clever to agree" can be avoided, by not letting any noun be the subject of any verb in the base grammar. The resulting system is essentially a conditional production system. It seems that this method permits a considerable simplification of the transformational rules by putting much of the burden on this conditional base grammar.

In Gilbert [1966] a class of languages called "analytic languages" is defined as given by a context-free grammar coupled with a "scan function". The latter is a function which computes at each step of a bottom-up analysis which productions of the grammar are applicable. In other terms we have a conditional context-free grammar with rules of the form:

$$\text{if applicable then } A \to BC \text{ else undefined.}$$

The link between conditional context-free systems used for top-down and bottom-up analysis is in relation (13).

It is quite obvious that if we allow the Boolean variables to be equal to any recursive predicate, any recursive set L of strings can be defined by a system of two conditional regular expressions:

$$S = \text{if given string} \in L \text{ then } T^* \text{ else undefined.}$$

$$T^* = \sum_{a \in T} aT^* + \lambda \quad .$$

This amounts to defining a language by its analyzer written in whatever formal language is used to specify $\underline{b}$ = given string $\in L$.

Since we are concerned with computers, the ability to define any recursive set is just what we need.

We see also that, as is the case for regular sets, the Boolean closure of this class is identical to the class itself (Lemma 6.1); nonetheless if it is more natural to define a set with & and ' , there is no reason not to do it.

It is very likely a difficult, maybe unsolvable, problem to determine when such systems have a solution. This does not detract from the usefulness of the formalism: it is not decidable whether an Algol 60 program will halt and any recursive set can be defined by a transformational grammar too, this does not bar Algol 60 from being a useful tool and transformational analysis a promising one.

f) Use of Recursive Functions of Regular Expressions

We suggest defining the syntax of languages by way of systems of recursive functions of regular expressions as follows:

First a number of arrays, auxilliary variables, counters, list structures, ad libitum, are declared.

These quantities can be manipulated as they can be in Algol 60 or Lisp.

The manipulation on these quantities are coupled to the execution of the recognizer steps; that is, they are defined by procedures, each one of which is associated to an equation of the regular form system, just like semantic procedures are.

The conditions are conditions upon the state of these quantities.

The role of these manipulations of lists, flags or arrays coupled with the analyzer steps is in fact to gather information in advance of the time it may be called for in a conditional. For instance, when we parse declarations in Algol 60 we build an identifier table, so that later on, when we parse a procedure call, we will know whether an actual parameter is an identifier as an array identifier or as a switch identifier (this is a point where the Algol 60 syntax is deliberately ambiguous); we can now formalize this. But note that even if an identifier table was not built, we could write a lengthy Boolean function, say in Lisp, which would examine the program and report whether the considered identifier has been declared as a switch or as an array. This is why we do not want to be formal or even precise about the form of these procedures coupled to the recognizer steps. They just represent a practical way of implementing Boolean functions by forseeing the questions which may be asked.

As for the conditions themselves, we see them as insuring the determinicity of analysis.

We suggest applying this method starting from an RCF or a simple precedence language embedded into the set to be defined; in other words to enhance well-behaved, fast analysis techniques.

It is true that this formalism is no more powerful than unrestricted rewriting systems and even perhaps context-sensitive ones, but we submit that it is incomparably more convenient just as Algol 60 is more convenient than Turing machine programming when it comes to numerical analysis problems. In this respect, we believe that a number of the problems of language definition have been self-inflicted. Because we

are entering the era of compiler compilers and querry systems there is a need for such a formalism.

g) Hints Toward Further Research

Let us hint toward further research and first let us remark that most often the possibilities of syntactical analysis are at present underexploited and its nature misunderstood. It is nearly always considered only as describing the recognition of well formed sentences in non-redundant precisely specified languages.

In Wirth [1966] it is shown how a syntactical analyzer can recover from errors and keep analyzing in certain cases by deliberately using production rules corresponding to not well formed sentences. This promising idea calls for some reflections on the nature of syntactical analysis: what an actual compiler analyzes is always T^* as partitioned into a language L plus its complement L'. The handling of L' can be done by classical methods.

In a multi-pass compiler, the first pass is usually devoted to a finite state transduction of the terminal symbols, plus a construction of the identifier tables by blocks, through a declaration scan. We have never seen this process explained but informally. It can and should be understood as directed by the syntactical analysis of a language L_1 in which the given language L is embedded, a language which would have the same block and declaration structure as L has, but would admit any string where L has simple statements. So that L_1 could effectively be described to a compiler compiler and analyzed by the same algorithm by which L is analyzed in subsequent passes. This is

not only for clarity's sake; systematic methods are invariably more efficient. In the particular case of a language organized as Euler is, we see that this first pass analysis can be accomplished by just one finite state automaton calling itself recursively at each block entry. On the other hand, subsequent passes should not have to analyze the declarations or check the block structure. What is done is to represent L as the intersection L_1 & L_2 of two languages where L_2 is obtained by replacing in L the declarations by any strings. L is accepted when L_1 and L_2 are consecutively accepted.

In the field of "natural language" interaction with computers, we have either ad hoc specialized systems which perform remarkably well, such as Weizenbaum's Eliza (Weizenbaum [1966]), Colby's on line belief system analyzer (Colby [1966]), Abelson and Caroll's simulator (Abelson and Caroll [1965]), or on the other hand general and theoretically grounded systems, fairly rigid and hardly field usable, such as the Mitre system (Zwicky, Friedman, Hall and Walker [1965]). All of the ad hoc programs do not care much about grammatical correctness and do not extract all the possible information of a sentence. We believe that this is necessary to natural language handling and can be formalized by the methods of syntax description we have outlined.

Last, some recent papers have shown how the theory or the techniques of syntax analysis can be applied in such apparently unrelated fields as number theory (Schützenberger [1966]) and combinatorial problems of geometry (Gross [1966]); there may be other elements in diverse disciplines where such elegant generalizations can be made, in return

we might expect from them some more insights into the mathematical nature of syntax analysis, possibly in the form of a theory of the constructive solutions of combinatorial problems; the first steps in that direction are perhaps to be found in Riguet [1962].

APPENDIX 1

Axiom System and Rules of Inference for T^*

($\vdash$ is understood.)

The following system is obtained by minor modifications of an axiom system for the expressions of predicate calculus in prefixed form, published in Tarski [1956], VIII, Section 2, page 173. It was communicated to us by D. Scott.

(0) $\lambda \in T^*$

$T \subseteq T^*$

$$\frac{\alpha \in T^*; \beta \in T^*}{\alpha\beta \in T^*}$$

(1) $\lambda \neq a\alpha$

(2) $$\frac{a\alpha = b\beta}{a = b, \alpha = \beta}$$

(3) $\lambda\alpha = \alpha$

(4) $a(\beta\gamma) = (a\beta)\gamma$

(5) $$\frac{\lambda \in A, TA \subset A}{A = T^*}$$

Note the resemblance to the Peano's axioms for natural numbers. Note also that (5) is a particular case of R1, first rule of inference for RE .

Axioms for first and rest are: (after McCarthy [1963]).

(6) $first(a\alpha) = a$

(7) $rest(a\alpha) = \alpha$

(8) α = if $\alpha = \lambda$ then λ else first(α)rest(α) .

It is possible to take first and rest as primitive, define concatenation of strings from concatenation of a letter to a string, taken as primitive, by:

(9) $\alpha\beta$ = if $\alpha = \lambda$ then β else first(α)(rest(α)β)

and replace (5) by the recursion induction principle (McCarthy [1963], page 58, 59) in which case (3) and (4) are no longer needed.

Then we need as in Section 6d the McCarthy's rules to manipulate the nesting of if-then-else's and 3 rules expressing the distributivity of if-then-else over first, rest and equality (see Section 6d, or McCarthy [1963] page 55 and 58).

We have chosen this last approach, since we have defined in Section 2 $|\alpha|$, α^n and α^R by recursive conditional expressions. Let us first give some examples of the first one, after what we shall derive some relations by the second approach. In particular, we shall prove (3) and (4), thus showing the equivalence of the two approaches.

(i)

$\alpha\lambda = \alpha$

Proof: Let $A = \{\alpha | \alpha\lambda = \alpha\}$, $\lambda \in A$ by (3), $TA \subseteq A$ by (4), thus $A = T^*$ by (5).

Associativity: $\alpha(\beta\gamma) = (\alpha\beta)\gamma$

Proof: Let $A = \{\alpha | \alpha(\beta\gamma) = (\alpha\beta)\gamma\}$, $\lambda \in A$ by (3), $TA \subseteq A$ by (4), whence $A = T^*$ by (5).

Left cancellation: $\alpha\beta = \alpha\gamma \Rightarrow \beta = \gamma$

Proof: Let $A = \{\alpha | \alpha\beta = \alpha\gamma \Rightarrow \beta = \gamma\}$, $\lambda \in A$ by (3), $TA \subseteq A$ by (4) and (2), whence $A = T^*$ by (5).

Refinement: $\alpha\beta = \gamma\delta \Rightarrow (\exists\xi)[\alpha\xi = \gamma \text{ or } \xi\beta = \delta]$

Proof: Let $A = \{\alpha | \alpha\beta = \gamma\delta \Rightarrow (\exists\xi)[\alpha\xi = \gamma \text{ or } \xi\beta = \delta]$ for any $\beta,\gamma,\delta \in T^*\}$, $\lambda \in A$ by (3), with $\xi = \gamma$; $TA \subseteq A$ by (2), whence $A = T^*$ by (5).

(ii) (The methodology and 12, 14, and 13 below are in McCarthy [1963b])

(10) $\quad first(\alpha\beta) = \text{if } \alpha = \lambda \text{ then } first(\beta) \text{ else } first($

$\quad rest(\alpha\beta) = \text{if } \alpha = \lambda \text{ then } rest(\beta) \text{ esle } rest(\alpha)\beta$.

Proof:

$first(\alpha\beta) = first(\text{if } \alpha = \lambda \text{ then } \beta \text{ else } first(\alpha)(rest(\alpha)\beta))$ by (9)

$= \text{if } \alpha = \lambda \text{ then } first(\beta) \text{ else } first(\alpha)$ by (6)

and by distributivity of if-then-else over functions.

Same proof for $rest(\alpha\beta)$.

(3) $\quad \lambda\alpha = \alpha$ by (9).

(11) $\quad \alpha\beta = \lambda \Rightarrow \alpha = \beta = \lambda$.

Proof:

$\alpha\beta = \lambda \Rightarrow (\text{if } \alpha = \lambda \text{ then } \beta \text{ else } first(\alpha)(rest(\alpha)\beta)) = \lambda$

$\Rightarrow \text{if } \alpha = \lambda \text{ then } \beta = \lambda \text{ else } first(\alpha)(rest(\alpha)\beta) = \lambda$

$\Rightarrow \text{if } \alpha = \lambda \text{ then } \beta = \lambda \text{ else } \underline{\text{false}}$ by (1)

$\Rightarrow A = \lambda \wedge \beta = \lambda$.

(12) $\alpha\lambda = \alpha$

Proof: $\alpha\lambda$ = if $\alpha = \lambda$ then λ else first(α)(rest(α)λ)

= if $\alpha = \lambda$ then λ else first($\alpha\lambda$)(rest($\alpha\lambda$))

by (10). This has the form of (8). Whence $\alpha\lambda = \alpha$.

(4) $(a\beta)\gamma$ = if $a\beta = \lambda$ then γ else first($a\beta$)(rest($a\beta$)γ)

= if false then γ else $a(\beta\gamma)$

= $a(\beta\gamma)$.

(13) Associativity:

$\alpha(\beta\gamma)$ = if $\alpha = \lambda$ then $\beta\gamma$ else first(α)(rest(α)($\beta\gamma$))

$(\alpha\beta)\gamma$ = (if $\alpha = \lambda$ then β else first(α)(rest(α)β))γ

= if $\alpha = \lambda$ then $\beta\gamma$ else (first(α)(rest(α)β))γ

= if $\alpha = \lambda$ then $\beta\gamma$ else first(α)((rest(α)β)γ)

by (4).

Whence $\alpha(\beta\gamma) = (\alpha\beta)\gamma$ as they both satisfy equations of the form

$Q(\alpha,\beta,\gamma)$ = if $\alpha = \lambda$ then $\beta\gamma$ else first(α)(Q(rest(α),β,γ))

(14) Left cancellation.

$\alpha\beta = \alpha\gamma \Rightarrow$ (if $\alpha = \lambda$ then β else first(α)(rest(α)β)) =

(if $\alpha = \lambda$ then γ else first(α)(rest(α)γ))

$\Rightarrow$ if $\alpha = \lambda$ then $\beta = \gamma$ else first(α)(rest(α)β) =

first(α)(rest(α)γ)

$\Rightarrow$ if $\alpha = \lambda$ then $\beta = \gamma$ else rest(α)β = rest(α)γ

by (2), of the form

$$P(\alpha,\beta,\gamma) = \text{if } \alpha = \lambda \text{ then true else } P(\text{rest}(\alpha),\beta,\gamma)$$

whence $\beta = \gamma$.

(15) Refinement.

$$\begin{aligned}
\alpha\beta = \gamma\delta &\Leftrightarrow (\text{if } \alpha = \lambda \text{ then } \beta \text{ else } \text{first}(\alpha)(\text{rest}(\alpha)\beta)) = \gamma\delta \\
&\Leftrightarrow \text{if } \alpha = \lambda \text{ then } \beta = \gamma\delta \text{ else } \text{first}(\alpha)(\text{rest}(\alpha)\beta) = \gamma\delta \\
&\Rightarrow \text{if } \alpha = \lambda \text{ then } \alpha\gamma = \gamma \text{ else } \gamma\delta = \text{first}(\alpha)(\text{rest}(\alpha)\beta) \\
&\Rightarrow \text{if } \alpha = \lambda \text{ then } (\exists\xi)[\alpha\xi = \gamma] \text{ else} \\
&\quad \text{if } \gamma = \lambda \text{ then } \alpha\beta = \delta \text{ else } \text{first}(\alpha)(\text{rest}(\alpha)\beta) = \\
&\qquad\qquad \text{first}(\gamma)(\text{rest}(\gamma)\delta) \\
&\Rightarrow \text{if } \alpha = \lambda \text{ then } (\exists\xi)[\alpha\xi = \gamma] \text{ else if } \gamma = \lambda \text{ then } (\exists\xi)[\xi\beta = \delta] \\
&\qquad \text{else } \text{first}(\alpha)(\text{rest}(\alpha)\beta) = \text{first}(\gamma)(\text{rest}(\gamma)\delta) \\
&\Rightarrow \text{if } \alpha = \lambda \text{ then } (\exists\xi)[\alpha\xi = \gamma] \text{ else if } \gamma = \lambda \text{ then } (\exists\xi)[\xi\beta = \delta] \\
&\qquad \text{else } \text{rest}(\alpha)\beta = \text{rest}(\gamma)\delta \qquad \text{by (2).}
\end{aligned}$$

Of the form:

$$P(\alpha,\beta,\gamma,\delta) = \text{if } \alpha = \lambda \vee \gamma = \lambda \text{ then } \underline{\text{true}} \text{ else } P(\text{rest}(\alpha),\beta,\text{rest}(\gamma),\delta)$$

whence $\alpha\beta = \gamma\delta \Rightarrow (\exists\xi)[\alpha\xi = \gamma \vee \xi\beta = \delta]$.

(16) $(\alpha\beta)^R = \beta^R\alpha^R$

Proof:

$$\begin{aligned}
(\alpha\beta)^R &= (\text{if } \alpha = \lambda \text{ then } \beta \text{ else } \text{first}(\alpha)(\text{rest}(\alpha)\beta))^R \\
&= \text{if } \alpha = \lambda \text{ then } \beta^R \text{ else } (\text{first}(\alpha)(\text{rest}(\alpha)\beta))^R
\end{aligned}$$

$$= \text{if } \alpha = \lambda \text{ then } \beta^R \text{ else if } \text{first}(\alpha)(\text{rest}(\alpha)\beta) = \lambda \text{ then}$$
$$\lambda \text{ else } \text{rest}(\text{first}(\alpha)(\text{rest}(\alpha)\beta))^R\text{first}(\text{first}(\alpha)(\text{rest}(\alpha)\beta))$$
$$= \text{if } \alpha = \lambda \text{ then } \beta^R \text{ else } (\text{rest}(\alpha)\beta)^R\text{first}(\alpha)$$

while

$$\beta^R\alpha^R = \beta^R(\text{if } \alpha = \lambda \text{ then } \lambda \text{ else } \text{rest}(\alpha)^R\text{first}(\alpha))$$
$$= \text{if } \alpha = \lambda \text{ then } \beta^R \text{ else } \beta^R(\text{rest}(\alpha)^R\text{first}(\alpha))$$
$$= \text{if } \alpha = \lambda \text{ then } \beta^R \text{ else } (\beta^R\text{rest}(\alpha)^R)\text{first}(\alpha)$$

both equations of the form

$$f(\alpha,\beta) = \text{if } \alpha = \lambda \text{ then } \beta^R \text{ else } f(\text{rest}(\alpha),\beta)\text{first}(\alpha)$$

whence $(\alpha\beta)^R = \beta^R\alpha^R$.

(17) $\alpha^{RR} = \alpha$

Proof: $\alpha^{RR} = (\text{if } \alpha = \lambda \text{ then } \lambda \text{ else } \text{rest}(\alpha)^R\text{first}(\alpha))^R$
$= \text{if } \alpha = \lambda \text{ then } \lambda \text{ else } (\text{rest}(\alpha)^R\text{first}(\alpha))^R$
$= \text{if } \alpha = \lambda \text{ then } \lambda \text{ else } \text{first}(\alpha)\text{rest}(\alpha)^{RR}$.

On the other hand

$$\alpha = \text{if } \alpha = \lambda \text{ then } \lambda \text{ else } \text{first}(\alpha)\text{rest}(\alpha)$$

whence $\alpha^{RR} = \alpha$.

(18) Right cancellation. $\beta\alpha = \gamma\alpha \Rightarrow \beta = \gamma$

by (16), (17), and (14).

For the remaining relations we need the following definition, for $n,p \in \Omega$.

$$n + p = \text{if } n = 0 \text{ then } p \text{ else } n^- + p'$$

n' successor of n, p^- predecessor of p. (See McCarthy [1963].)

$$|\alpha| = \text{if } \alpha = \lambda \text{ then } 0 \text{ else } 1 + |\text{rest}(\alpha)| =$$
$$\text{if } \alpha = \lambda \text{ then } 0 \text{ else } |\text{rest}(\alpha)|' \quad .$$

(19) $|\alpha\beta| = |\alpha| + |\beta|$

Proof:

$$|\alpha\beta| = |\text{if } \alpha = \lambda \text{ then } \beta \text{ else } \text{first}(\alpha)(\text{rest}(\alpha)\beta)|$$
$$= \text{if } |\alpha| = 0 \text{ then } |\beta| \text{ else } |\text{first}(\alpha)(\text{rest}(\alpha)\beta)|$$
$$= \text{if } |\alpha| = 0 \text{ then } |\beta| \text{ else } |\text{rest}(\alpha)\beta|' \quad .$$

It is possible to prove (ibid.) that $n^- + p' = (n^- + p)'$ whence $|\alpha\beta| = |\alpha| + |\beta|$ since $|\text{rest}(\alpha)| = |\alpha|^-$ by definition.

(20) $\alpha^n\alpha^p = \alpha^{n+p}$

Proof:

$$\alpha^n\alpha^p = (\text{if } n = 0 \text{ then } \lambda \text{ else } \alpha^{n-1}\alpha)\alpha^p$$
$$= \text{if } n = 0 \text{ then } \alpha^p \text{ else } \alpha^{n^-}\alpha\alpha^p$$
$$= \text{if } n = 0 \text{ then } \alpha^p \text{ else } \alpha^{n^-}\alpha^{p'}$$

whence $\alpha^n\alpha^p = \alpha^{n+p}$.

APPENDIX 2

A Context-free Grammar for $\mathcal{R}$

We give a context-free grammar for $\mathcal{R}$ in BNF; the symbol stands for the metalinguistic + ; we take $T = \{a,b,c,d\}$ for instance:

$$
\begin{aligned}
\mathcal{R} ::= \ & \emptyset \mid \lambda \mid \\
& a \mid b \mid c \mid d \mid \\
& (\mathcal{R}) \mid \\
& \mathcal{R} + \mathcal{R} \mid \mathcal{R} \,\&\, \mathcal{R} \mid \mathcal{R} \cdot \mathcal{R} \mid \\
& \mathcal{R}' \mid \mathcal{R}^*
\end{aligned}
$$

Note that this grammar, which follows exactly the formal definition of $\mathcal{R}$, is ambiguous. This corresponds to the necessity of priority rules for interpretation.

APPENDIX 3

Some Relations Derivable from <RE,R1,R2>

(i) Boolean relations: (Huntington [1904] except 6th and 7th).

(b1) to (b8) are globally invariant under an exchange of + and & , $(T)^*$ and $\emptyset$. Each derivable relation has its dual obtained by this permutation.

-- There is no $X \neq \emptyset$ such $\vdash A + X = A$ for all A . Otherwise we would have $\vdash X + \emptyset = X = \emptyset$. Dually the maximal element $(T)^*$ of the lattice Θ is unique.

-- $\vdash A + (T)^* = (A + (T)^*) \& (T)^* = (T)^* \& (A + (T)^*) =$

$$(A + A') \& (A + (T)^*) = A + A' \& (T)^* = A + A' = (T)^*$$

thus $\vdash A + (T)^* = (T)^*$ and dually $\vdash A \& \emptyset = \emptyset$.

-- Absorption Law.

$$\vdash A + (A \& B) = A \& (T)^* + A \& B = A \& ((T)^* + B) =$$

$$A \& (B + (T)^*) = A \& (T)^* = A$$

thus $\vdash A + (A \& B) = A$ and dually $\vdash A \& (A + B) = A$.

-- The inverse is unique. Suppose A has two inverses A_1' and A_2' .

$$\vdash A_2' = (T)^* \& A_2' = (A + A_1') \& A_2' = A \& A_2' + A_1' \& A_2' =$$

$$\emptyset + A_1' \& A_2' = A_1' \& A + A_1' \& A_2' = A_1' \& (A + A_2')$$

$$= A_1' \& (T)^* = A_1' \quad .$$

-- (De Morgan's law)

$$\vdash A + B = (A' \,\&\, B')$$

Proof: first $\vdash A + (A' + C) = (T)^*$ and dually $\vdash A \,\&\, (A' \,\&\, C) = \emptyset$

since

$$\vdash A + (A' + C) = (T)^* \,\&\, (A + (A' + C)) =$$

$$(A + A') \,\&\, (A + (A' + C)) = A + (A' \,\&\, (A' + C)) =$$

$$A + A' = (T)^*$$

then

$$\vdash (A + B) + (A' \,\&\, B') = ((A + B) + A') \,\&\, ((A + B) + B') =$$

$$(T)^* \,\&\, (T)^* = (T)^*$$

and

$$\vdash (A + B) \,\&\, (A' \,\&\, B') = (A(A' \,\&\, B')) + (B(A' \,\&\, B')) =$$

$$\emptyset + \emptyset = \emptyset$$

whence

$$\vdash A + B = (A' \,\&\, B')'$$

and dually

$$\vdash A \,\&\, B = (A' + B')' \quad .$$

-- $\vdash \emptyset' = (T)^*$

Proof: $\vdash \emptyset + (T)^* = (T)^*$ and $\vdash \emptyset \,\&\, (T)^* = \emptyset$.

-- $\vdash (A')' = A$

Proof:

$$\vdash A = A + \emptyset = (A' \,\&\, \emptyset')' = (A' \,\&\, (T)^*)' = (A')' \quad .$$

-- Associativity.

Let $(A + B) + C = X$ and $A + (B + C) = Y$.

$$(A' \,\&\, B') \,\&\, C' = X' \quad .$$

We have $\vdash Y + A' = Y + B' = Y + C' = (T)^*$.

Proof: $\vdash Y + A' = A' + (A + (B + C)) = (T)^*$.

$$\vdash Y + B' = (T)^* \,\&\, (B' + Y) = (B' + B) \,\&\, (B' + Y) =$$
$$B' + (B \,\&\, Y) = B' + (B \,\&\, (A + (B + C))) =$$
$$B' + (B \,\&\, A + (B \,\&\, (B + C))) =$$
$$B' + (B \,\&\, A + B) = B' + B = (T)^*$$

similarly for $Y + C'$.

Dually $\vdash X' \,\&\, A = X' \,\&\, B = X' \,\&\, C = \emptyset$.

Now

$$\vdash Y + X' = Y + ((A' \,\&\, B') \,\&\, C') =$$
$$((Y + A') \,\&\, (Y + B')) \,\&\, (Y + C') = ((T)^* \,\&\, (T)^*) \,\&\, (T)^* = (T)^* .$$

$$\vdash X' \,\&\, Y = X'(A + (B + C)) =$$
$$((X' \,\&\, A) + (X' \,\&\, B)) + (X' \,\&\, C) = (\emptyset + \emptyset) + \emptyset = \emptyset$$

hence

$$\vdash X = Y \quad \text{and} \quad \vdash A + (B + C) = (A + B) + C \quad .$$

Dually $\vdash A \,\&\, (B \,\&\, C) = A \,\&\, (B \,\&\, C)$.

(ii) Other relations (first 3 in Aanderaa [1965], Salomaa [1966]).

-- $\vdash \emptyset A = \emptyset$.

Proof:

$$\vdash \emptyset\emptyset = \emptyset \Rightarrow \vdash \emptyset(\emptyset A) = \emptyset A$$

$$\Rightarrow \vdash \emptyset(\emptyset A) + \emptyset = \emptyset A \Rightarrow \vdash \emptyset A = \emptyset^* \emptyset = \emptyset \quad .$$

-- $\vdash \emptyset^* = \lambda$ by (s1) and above.

-- $\vdash \lambda A = A$

Proof:

$$\vdash A = A + \emptyset = A + \emptyset A = \emptyset A = \emptyset A + A \Rightarrow$$

$$\vdash A = \emptyset^* A = \lambda A \quad .$$

-- $\vdash \lambda^* = \lambda$

Proof:

$$\vdash \lambda^* = (\lambda + \emptyset)^* = \emptyset^* = \lambda \quad .$$

-- $\vdash A^* = \lambda + A + A^2 + \ldots + A^n A^* \quad (\forall n)$

by (s1) and the principle of complete induction for integers.

-- $\vdash A^* = (A - \lambda)^*$ where, as usual, $A - B = A \,\&\, B'$.

Proof:

$$\vdash A^* = (A \,\&\, \lambda' + A \,\&\, \lambda)^* = (A - \lambda)^*$$

in all cases.

-- $\vdash A^* = A^*A^*$

Proof:

$$\vdash A^* = \lambda + AA^* = \lambda + (A \,\&\, \lambda' + A \,\&\, \lambda)A^*$$
$$= \lambda + (A - \lambda)A^* + A^*$$

in all cases since $\vdash A^* = A^* + A^*$.

Whence $\vdash A^* = (A - \lambda)A^* + A^*$ since $\vdash A^* = \lambda + A^*$ by (s1), whence, by R1

$$\vdash A^* = (A - \lambda)^*A^* = A^*A^* \quad .$$

-- $\vdash \lambda \,\&\, A = \lambda \Rightarrow \vdash AA^* = A^*$.

Proof:

$$\vdash \lambda \,\&\, A = \lambda \Rightarrow \vdash \lambda \,\&\, AA^* = \lambda \,\&\, (A + \lambda)(A^* + \lambda)$$

whence

$$\vdash \lambda \,\&\, AA^* = \lambda$$

whence

$$\vdash AA^* = \lambda + AA^* = \lambda^* \quad .$$

-- $\vdash AA^* = A^*A$.

Proof:

$$\vdash AA^* = A(\lambda + AA^*) = A + AAA^* = A + (A \,\&\, \lambda')AA^* + (A \,\&\, \lambda)AA^*$$
$$\Rightarrow \vdash AA^* = (A \,\&\, \lambda')^*(A + (A \,\&\, \lambda)AA^*) = A^*A + A^*(A \,\&\, \lambda)AA^*$$

if $\vdash A \,\&\, \lambda = \cdots$ we obtain $\vdash AA^* = A^*A$

if $\vdash A \,\&\, \lambda = \lambda$ we obtain

$$\vdash AA^* = A^*A + A^*AA^* = A^*A + A^*AA^* = A^*A + A^*A^* = A^*A + A^*$$

whence $\vdash AA^* = A^* = A^*A + A^*$.

On the other hand $\vdash A \;\&\; \lambda = \lambda \Rightarrow \vdash A = A + \lambda$ whence

$$\vdash A^*A = A^*(\lambda + A) = A^* + A^*A = A^*A + A^*$$

$$\begin{cases} \vdash A^* = A^*A + A^* \\ \vdash A^*A = A^*A + A^* \end{cases} \Rightarrow \vdash A^* = A^*A$$

since $\vdash A^* = AA^*$ we obtain the desired relation.

-- $\vdash A^{**} = A^*$

Proof:

$$\vdash A^* = A^*A^* = (A^* \;\&\; \lambda')A^* + (A^* \;\&\; \lambda)A^* = (A^* - \lambda)A^* + A^*$$

$$\Rightarrow \vdash A^* = (A^* - \lambda)^*A^* = A^{**}A^* = A^*A^{**}$$

but

$$\vdash \lambda \;\&\; A^* = \lambda \;\&\; (A^* + \lambda) = \lambda \Rightarrow \vdash A^*A^{**} = A^{**}$$

whence

$$\vdash A^{**} = A^* \quad .$$

-- Note on rule R1.

In

$$\frac{\vdash \delta(B) = \emptyset, \vdash A = BA + C}{\vdash A = B^*C}$$

we do need $\vdash \delta(B) = \emptyset$.

e.g.: $\vdash A^* = \lambda A^* + \lambda$ since $\vdash B = \lambda B$ and $\vdash A^* + \lambda = A^*$ by (s1).

This would imply $\vdash A^* = \lambda$ if we had not $\vdash \delta(B) = \emptyset$ in R1. (Generally, we could derive $\vdash A = \lambda$ or $\vdash A = \emptyset$ for any A, just as one can derive that any number is equal to any number by division by zero.)

Note that we have shown that $\vdash A^* = (A \cdot \lambda)^*$.

APPENDIX 4

Euler System

Euler is defined by the following context-free grammar (Wirth and Weber [1964]) in which Euler parentheses are denoted ⌈ and ⌉, ↑ by ⊙ and λ standing for identifier, by i, in order to avoid confusions:

1: vardecl → **new** i
2: fordecl → **formal** i
3: labdecl → **label** i
4: var- → i
5: var → var- [expr]
6: var- → var- .
7: var → var-
8: logval → **true**
9: logval → **false**
10: digit → 0
. . . . . .
19: digit → 9
20: integer- → digit
21: integer- → integer- digit
22: integer' → integer-
23: real' → integer'.integer'
24: real' → integer'
25: number → real'
26: number → real'$_{10}$ integer'
27: number → real'$_{10}\dot{-}$ integer'
28: number → $_{10}$ integer'
29: number → $_{10}\dot{-}$ integer'
30: reference → ⊙var
31: listhead → listhead expr ,
32: listhead → ⌈
33: list' → listhead expr⌉
34: list' → listhead⌉
35: prochead → prochead fordecl;
36: prochead → '
37: procdef → prochead expr '
38: primary → var
39: primary → var list'
40: primary → logval
41: primary → number
42: primary → σ
43: primary → reference
44: primary → list'

45: primary → tail primary

46: primary → procdef

47: primary → Ω

48: primary → [expr]

49: primary → in

50: primary → isb var

51: primary → isn var

52: primary → isr var

53: primary → isl var

54: primary → isli var

55: primary → isy var

56: primary → isp var

57: primary → isu var

58: primary → abs primary

59: primary → length var

60: primary → integer primary

61: primary → real primary

62: primary → logical primary

63: primary → list primary

64: factor- → primary

65: factor- → factor- ↑ primary

66: factor → factor-

67: term- → factor

68: term- → term- × factor

69: term- → term- / factor

70: term- → term- ÷ factor

71: term- → term- mod factor

72: term → term-

73: sum- → term

74: sum- → ⊙ term

75: sum- → - term

76: sum- → sum- ⊙ term

77: sum- → sum- - term

78: sum → sum-

79: choice- → sum

80: choice- → choice- min sum

81: choice- → choice- max sum

82: choice → choice-

83: relation → choice

84: relation → choice = choice

85: relation → choice ≠ choice

86: relation → choice < choice

87: relation → choice ≤ choice

88: relation → choice ≥ choice

89: relation → choice > choice

90: negation → relation

91: negation → ¬ relation

92: conjhead → negation ∧

93: conj- → conjhead conj

94: conj- → negation

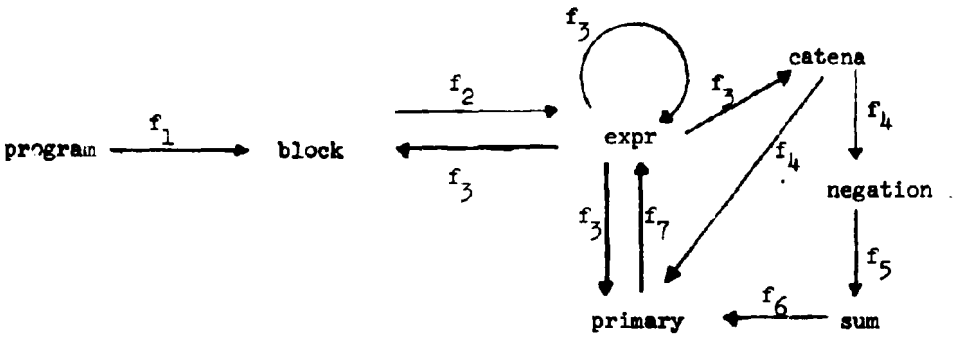
95: conj → conj-

96: disjhead → conj ∨

97:	disj	→ disjhead disj	98:	disj	→ conj
99:	catena	→ catena & primary	100:	catena	→ disj
101:	truepart	→ expr else	102:	ifclause	→ if expr then
103:	expr-	→ block	104:	expr-	→ ifclause truepart expr-
105:	expr-	→ var ← expr-	106:	expr-	→ goto primary
107:	expr-	→ out expr-	108:	expr-	→ catena
109:	expr	→ expr-	110:	stat-	→ labdef stat-
111:	stat-	→ expr	112:	stat	→ stat-
113:	labdef	→ i :	114:	blokhead	→ begin
115:	blokhead	→ blokhead vardecl;	116:	blokhead	→ blokhead labdecl;
117:	blokbody	→ blokhead	118:	blokbody	→ blokbody stat ;
119:	block	→ blokbody stat end	120:	program	→ ⊥ block ⊥

In order to make the system of equations obtained by regular expressions manipulation techniques more readable, we do not fully eliminate all the variables which can be eliminated; i.e., all but expr , but leave it in the form:

$$\begin{aligned}
\text{program} &= f_1(\text{block}) \\
\text{block} &= f_2(\text{expr}) \\
\text{expr} &= f_3(\text{expr},\text{primary},\text{block},\text{catena}) \\
\text{catena} &= f_4(\text{negation},\text{primary}) \\
\text{negation} &= f_5(\text{sum}) \\
\text{sum} &= f_6(\text{primary}) \\
\text{primary} &= f_7(\text{expr})
\end{aligned}$$



expr is common to all circuits.

f_1 corresponds to rule	120
f_2	119-110, 1, 3
f_3	109-101, 7-4
f_4, f_5, f_6	100-64
f_7	63-4, 2

program = ⊥ block ⊥

block = begin ((new i + label i);)*(i:)*expr(;(i:)*expr)* end

expr = (out + if expr then expr else + i([expr] + ·)*←)*

(goto primary + block + catena)

catena = ((negation ∧)*negation ∨)*(negation ∧)*negation(& primary)*

negation = (¬ + λ)sum((max + min)sum)*

(λ + (> + ≥ + ≤ + < + ≠ + =)sum((max + min)sum)*)

sum = (λ + - + ⊙)primary(↑ primary)*

((mod + ÷ + / + ⊛)primary(↑ primary)*)*

((- + ⊙)primary(↑ primary)*

((mod + ÷ + / + ⊛)primary(↑ primary)*)*)*

$$
\begin{aligned}
\text{primary} = {} & (\underline{\text{tail}} + \underline{\text{list}} + \underline{\text{logical}} + \underline{\text{real}} + \underline{\text{integer}} + \underline{\text{length}} + \underline{\text{abs}})^{*} \\
& ((\underline{\text{isu}} + \underline{\text{isp}} + \underline{\text{isy}} + \underline{\text{isli}} + \underline{\text{isl}} + \underline{\text{isr}} + \underline{\text{isn}} + \underline{\text{isb}}) \\
& \text{i}([\text{expr}] + \cdot)^{*} + \\
& \underline{\text{in}} + [\text{expr}] + \Omega + \sigma + \text{`}(\underline{\text{formal}}\ \text{i};)^{*}\text{expr'} + \\
& \{(\text{expr},)^{*}\text{expr}] + \bullet\ \text{i}([\text{expr}] + \cdot)^{*} + \\
& (0 + \ldots + 9)^{*}(0 + \ldots + 9) \\
& (\lambda + \cdot(0 + \ldots + 9)^{*}(0 + \ldots + 9))) \\
& (\lambda + {}_{10}(\lambda + \div)(0 + \ldots + 9)^{*}(0 + \ldots + 9)) + \\
& {}_{10}(\lambda + \div)(0 + \ldots + 9)^{*}(0 + \ldots + 9) + \\
& \text{i}([\text{expr}] + \cdot)^{*}(\lambda + \{(\text{expr};)^{*}\text{expr}\}) + \underline{\text{true}} + \underline{\text{false}} \quad .
\end{aligned}
$$

We have here, in less than one page the context-free syntax of a systematic generalization of Algol 60. At the same time this syntax specifies a high speed analyzer for the language.

If such expressions are not easy to handle for a human being, they are well adapted to machines.

Note that by not completely eliminating the variables which can be eliminated, one can reduce the total size of the tables; for instance, it would be unwise to eliminate primary in sum while primary is repeated 8 times in sum. In the same way the construct $(0 + \ldots + 9)^{*}(0 + \ldots + 9)$ occurs 4 times in primary, it should be replaced by a variable, unless somebody is interested only in speed since naturally the introduction of spurious variables results in more pushdown manipulations and a speed loss. On the other hand, introducing extra variables can result in considerable space savings when the corresponding extra automata use but a small fraction of the alphabet.

APPENDIX 5

Computation of Π

By Definition 5.2:

$$\Pi(X_1,X_2) = [\text{first}(\Phi_{X_1} X_1) \cap \text{first}(X_2) = \emptyset] \quad .$$

There are well-known techniques for the computation of $\text{first}(X_2)$ when X_2 is context-free.

We are going to give an algorithm to compute $\text{first}(\Phi_{X_1} X_1)$. An example is given at the end of this Appendix.

(i) For regular form systems.

Let us set $\text{ft}(X) = \text{first}(\Phi_X X)$.

Consider the subsystem of X .

The equation of X has two possible forms:

a) $X = \sum_{a \in \text{first}(X)} aB_a + \delta(X)$

b) $X = YZ$

By definition, we see that, respectively:

a) $\text{ft}(X) = \delta(X) \cdot \text{first}(X) + \sum_{a \in \text{first}(X)} \text{ft}(B_a)$

b) $\text{ft}(X) = \delta(Y) \cdot \delta(Z) \cdot \text{first}(X) + \text{ft}(Z) + \delta(Z) \cdot \text{ft}(Y)$.

So that to compute $\text{ft}(X)$ we need compute the $\text{ft}(B_a)$ terms or $\text{ft}(Z)$ and possibly $\text{ft}(Y)$.

Note that $\delta(A)$, where A is a context-free set, is easy to compute.

We do the same manipulation on $ft(B_a)$ or $ft(Z)$ and $ft(Y)$ and we expand $ft(X)$ in that way.

I is finite. When in expanding $ft(P)$ we meet $ft(P)$ we naturally do not start computing it recursively, it would amount to write $A = B + A + A + A + \ldots$; instead, we just leave it in the expression, so that we will end up with something of the form:

$$ft(X) = \sum_{I_1} first(A) + \sum_{I_2} ft(Z) \qquad I_1, I_2 \subseteq I \quad .$$

This implies $\sum_{I_1} first(A) \subseteq ft(X)$.

Let us show that $ft(X) \subseteq \sum_{I_1} first(A)$.

No variable Z in the $\sum_{I_2} ft(Z)$ term depends upon a variable not in $I_1 + I_2$, by hypothesis.

If $I_1 = \emptyset$ no variable contains λ , this implies $X = \emptyset$; the initial system should be simplified; let us suppose then that $I_1 \neq \emptyset$.

Suppose $(\exists b)[b \in ft(X) \wedge b \notin \sum_{I_1} first(A)]$.

b necessarily occurs in a type a) rule, these are the only rules containing letters:

$$(\exists b)[B = \sum_{a \neq b} aB_a + bB_b + \delta(B)]$$

and

$$b \in ft(X) \Rightarrow X >^* B \wedge \delta(B) = \lambda \Rightarrow B \in I_1 \Rightarrow$$

$$b \in first(B) \subseteq \sum_{I_1} first(A) \qquad \text{a contradiction.}$$

Thus $ft(X) \subseteq \sum_{I_1} first(A)$ so that $ft(X) = \sum_{I_1} first(A)$.

In graph terms what we do is very simple: We consider all the nodes which are exit points (X_i such that $\delta(X_i) = \lambda$), the labels of the vertices going from these nodes yield the elements of $ft(X_1)$, unless they correspond to an equation of type b, in which case a little more work is required.

(ii) for s'-grammars.

The same algorithm, with obvious variations will work for s'-grammars or we can reduce an s'-grammar to a regular form system as follows:

-- reduce to an s'-2-grammar (Corollary 5.9).

-- to a rule $X \to aX_1X_2$ associate a term aX_3 , X_3 a new symbol, in the equation of X , and add the equation $X_3 = X_1X_2$. Take $\delta(X) = \lambda$ in the equation of X if and only if $X \to \lambda$ occurs in the s'-2-grammar.

(iii) for fcr grammars.

An algorithm is discussed in Schorre [1965]. It works on the same principle that the one in (i), which we could clearly adapt to fcr grammars too, but is somewhat more complicated.

Example.

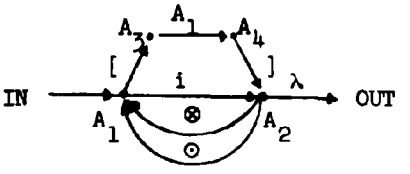
Let us take the regular form system which we use as an example for Theorem 5.8:

$A_1 = iA_2 + [A_3$

$A_2 = \otimes A_1 + \odot A_1 + \lambda$

$A_3 = A_1A_4$

$A_4 =]A_2$



$$
\begin{aligned}
ft(A_1) &= ft(A_2) + ft(A_3) && \text{expanding } ft(A_2)\\
&= \otimes + \odot + ft(A_1) + ft(A_3) && \text{expanding } ft(A_3)\\
&= \otimes + \odot + ft(A_1) + ft(A_4) + \delta(A_4)ft(A_1) + \delta(A_1)\delta(A_4)first(A_1)\\
&= \otimes + \odot + ft(A_1) + ft(A_2) && \text{all the } ft \text{ terms have been met.}\\
&= \otimes + \odot
\end{aligned}
$$

as we have just proved, and as is obvious from an examination of the graph.

APPENDIX 6

Two Conjectures on the Boolean Closure of Context-free Languages

(i) Ambiguity and inherent ambiguity.

Since a set belonging to the Boolean closure of context-free languages is deterministic context-sensitive, it has a non-ambiguous context-sensitive grammar (see in Kuroda [1964] the one-to-one correspondence between a linear bounded automaton computation and a context-sensitive grammar derivation).

This is true even of such a context-free language as:

$$\{a^n b^n c^p \mid n,p \in \Omega\} + \{a^q b^r c^r \mid q,r \in \Omega\}$$

which is inherently ambiguous for any string of the non-context-free intersection of its two components (Parikh [1961]). This fact evokes the Boolean equality

$$A + B = A \,\&\, B' + B$$

which shows how the set could be defined without overlapping, providing an intuitive but possibly wrong explanation for the existence of a non-ambiguous context-sensitive grammar defining it.

This yields another question: Is every inherently ambiguous language the union of two context-free languages such that their intersection is not context-free?

There are very few inherently ambiguous languages known (see Ginsburg and Ullian [1966], Hibbard and Ullian [1966]) and it is the case for all of them.

Let P be the context-free set of all palindromes without central marker. (Even palindromes.)

We conjecture that the context-free sets $P \cdot T^*$ and $P \cdot P$ are inherently ambiguous. Unfortunately the Parikh and Ginsburg techniques are not applicable here. We have been able to obtain only partial results for PT^* by studying the ways in which an even palindrome can be embedded into another one.

(ii) Characterization.

Another interesting research topic is to try to characterize the Boolean closure of context-free languages by a property similar to the important Bar-Hillel, Perles and Shamir theorem 4.1: A is context-free and infinite ⇔

$$(\exists \alpha,\beta,\gamma,\delta,\varepsilon)[\alpha\beta\gamma\delta\varepsilon \in A \Rightarrow \alpha\beta^n\gamma\delta^n\varepsilon \in A, \forall n \in \Omega]$$

This theorem is a non-commutative restriction of the Parikh mapping theorem; the results of Ginsburg and Spanier [1964], [1966], on the Boolean closure of semi-linear sets make it a reasonable conjecture that a similar commutative mapping theorem can be obtained for the Boolean closure of context-free languages.

On the other hand a number of results on this class can be gathered from results on context-free languages, such as the undecidability of the emptiness problem (see Theorem 5.18). A family of endomorphisms of this class has also been studied by Schützenberger [1964].

REFERENCES

Aanderaa, S. [1965]. "On the algebra of regular expressions". Harvard University. Mimeograph.

Abelson, R. P. and Carrol, J. D. [1965]. "Computer simulation of individual belief systems." American Behavioral Scientist. Vol. VIII. No. 9. p. 24-30.

Arden, D. N. [1961]. "Delayed logic and finite state machines." Proceedings of the second annual symposium on switching theory and logical design. (AIEE). p. 133-151.

Bar-Hillel, Y., Perles, M. and Shamir, E. [1961]. "On formal properties of simple phrase structure grammars." Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung. Vol. 14. p. 143-172. Also as Chapter 9 in Bar-Hillel, Y. [1964]. "Language and Information", p. 116-150, Addison-Wesley, Reading.

Berge, C. [1958]. "La théorie des graphes et ses applications." (in French.) Dunod, Paris. (English translation: "The theory of graphs and its applications", Wiley (1962), New York.)

Braffort, P. and Hirschberg, D. (Eds.) [1963]. "Computer programming and formal systems." (Studies in logic and the foundations of mathematics.) North-Holland Publishing Company, Amsterdam.

Brzozowski, J. A. [1964]. "Derivatives of regular expressions." Journal of the ACM. Vol. 11. p. 481-494.

Brzozowski, J. A. and McCluskey, E. J. Jr. [1963]. "Signal flow graph techniques for sequential circuit states diagrams." IRE Trans. on EC. Vol. 12. p. 67-76.

Carr, John W. III, Weiland, J. [1966]. "A non-recursive method of syntax specification." Comm. of the ACM. Vol. 9. p. 267-269.

Chomsky, N. [1959a]. "On certain formal properties of grammars." Information and Control. Vol. 2. p. 137-167.

Chomsky, N. [1959b]. "A note on phrase structure grammars." Information and Control. Vol. 2. p. 393-395.

Chomsky, N. [1963]. "Formal properties of grammars." in Handbook of mathematical psychology. Vol. 2, Chapter 12. p. 323-418. Luce, Bush and Galanter (Eds.). J. Wiley, New York.

Chomsky, N. [1965]. "Aspects of the theory of syntax." The M.I.T. Press, Boston.

Chomsky, N. and Miller, G. [1958]. "Finite state languages." Information and Control. Vol. 1. p. 91-112.

Chomsky, N. and Schützenberger, M. P. [1963]. "The algebraic theory of context-free languages." in Braffort and Hirschberg (Eds.) [1963]. p. 118-161.

Church, A. [1956]. "Introduction to Mathematical Logic." Vol. I. Princeton University Press. Princeton.

Colby, K. M. [1965]. "Computer simulation of change in personal belief systems." Paper delivered in Section L_2, the Psychiatric Sciences, General System Research, AAAS Berkeley Meeting, December 29, 1965.

Čulík, K. [1962]. "Formal structure of Algol and simplification of its description." in Symbolic languages in data processing. p. 75-82. Gordon and Breach, New York.

Davis, M. [1958]. "Computability and unsolvability." McGraw-Hill, New York.

Floyd, R. W. [1963]. "Syntactic analysis and operator precedence." Journal of the ACM. Vol. 10. p. 316-333.

Floyd, R. W. [1964]. "Bounded context syntactic analysis." Comm. of the ACM. Vol. 7. p. 62-65.

Friedman, J. [1957]. "Some results in Church's restricted recursive arithmetic." Journal of symbolic logic. Vol. 22. No. 4. p. 337-342.

Ghiron, H. [1962]. "Rules to manipulate regular expressions of finite automata." IRE Trans. on EC. Vol. 11. p. 574-575.

Gilbert, P. [1966]. "On the syntax of algorithmic languages." Journal of the ACM. Vol. 13. p. 90-107.

Ginsburg, S. [1966]. "The mathematical theory of context-free languages." McGraw-Hill, New York.

Ginsburg, S. and Greibach S. [1965]. "Deterministic context-free languages." SDC report TM-738/014/00. May 7, 1965. Also, in Information and Control. Vol. 8. (1966). p. 620-648.

Ginsburg, S. and Harrison, M. A. [1966]. "Bracketed context-free languages." SDC report TM-738/023/00. Jan. 4, 1966.

Ginsburg, S. and Rice, H. G. [1962]. "Two families of languages related to Algol." Journal of the ACM. Vol. 9. p. 350-371.

Ginsburg, S. and Rose, G. F. [1963]. "Operations which preserve definability in languages." Journal of the ACM. Vol. 10. p. 175-195.

Ginsburg, S. and Spanier, E. H. [1964]. "Bounded Algol-like languages." Transactions of the American Math. Soc. Vol. 113. p. 333-368.

Ginsburg, S. and Spanier, E. H. [1966]. "Semigroups, Presburger formulas and languages." Pacific Journal of Mathematics. Vol. 16. p. 285-296.

Ginsburg, S. and Ullian, J. [1966]. "Ambiguity in context-free languages." Journal of the ACM. Vol. 13. p. 62-89.

Greibach, S. A. [1965]. "A new normal form theorem for context-free phrase-structure grammars." Journal of the ACM. Vol. 12. p. 42-52.

Gross, M. [1966]. "Applications géométriques des langages formels." (in French.) ICC Bulletin. Vol. 5. No. 3. p. 141-167.

Hibbard, T. N. [1966]. "Scan limited automata and context limited grammars." To appear.

Hibbard, T. N. and Ullian, J. [1966]. "The independence of inherent ambiguity from complementedness among context-free languages." Journal of the ACM. Vol. 13. p. 588-593.

Huntington, E. V. [1904]. "Sets of independent postulates for the algebra of logic." Trans. Amer. Math. Soc. Vol. 5. p. 288-309.

Irons, E. T. [1964]. ""Structural connections" in formal languages." Comm. of the ACM. Vol. 7. p. 67-71.

Kleene, S. C. [1951]. "Representation of events in nerve nets and finite automata." RAND research memorandum RM-704 (12/15/1951) and in Shannon, C. E. and McCarthy, J. (Eds.) [1956]. p. 3-41.

Knuth, D. E. [1965a]. "A list of the remaining trouble spots in Algol 60." AB19.3.7 Algol Bulletin No. 19. p. 29-38.

Knuth, D. E. [1965b]. "On the translation of languages from left to right." Information and Control. Vol. 8. p. 607-639.

Knuth, D. E. and Merner, J. N. [1961]. "Algol 60 Confidential." Comm. of the ACM. Vol. 4. p. 268-272.

Korenjak, A. J. and Hopcroft, J. E. [1966]. "Simple deterministic languages." Technical report No. 51, August 1966. Princeton University. Also in the proceedings of the 7th annual symposium on switching and automata theory. (IEEE). p. 36-46.

Kuno, S. and Oettinger, A. G. [1962]. "Multiple-path syntactic analyzer." In Information Processing 62 (IFIP congress). p. 306-311. Popplewell (Ed.), North-Holland, Amsterdam.

Kuroda, S.-Y. [1964]. "Classes of languages and linear bounded automata." Information and Control. Vol. 7. p. 207-223.

Landweber, P. S. [1964]. "Decision problems of phrase structure grammars." IEEE Trans. on EC. Vol. 13. p. 354-362.

Letichevskii, A. A. [1965]. "The representation of context-free languages in automata with a push-down type store." Cybernetics (Kibernetika). Vol. 1. No. 2. p. 81-86. The Faraday Press, New York.

Medema, P. [1965]. "Another trouble spot in Algol 60." AB 20.3.7. Algol Bulletin. No. 20. p. 47-8.

McCarthy, J. [1960]. "Recursive functions of symbolic expressions and their computation by machine. Part I." Comm. of the ACM. Vol. 3. p. 184-195.

McCarthy, J. [1963]. "A basis for a mathematical theory of computation." in Braffort. P. and Hirschberg D. (Eds.). [1963]. p. 33-70.

McNaughton, R. [1965]. "Techniques for manipulating regular expressions." M.I.T. Project M.A.C. Machine structure group memo No. 10.

McNaughton, R. and Yamada, N. [1960]. "Regular expressions and state graph for automata." IRE Trans. on EC. Vol. 9. p. 39-47.

Moore, E. F. [1956]. "Gedanken-experiments on sequential machines." in Shannon and McCarthy (Eds.) [1956]. p. 129-153.

Naur, P. (Ed.) [1963]. "Revised report on the algorithmic language ALGOL 60." Comm. of the ACM. Vol. 6. p. 1-17.

Parikh, R. J. [1961]. "Language generating devices." Quarterly progress report No. 60. Research Laboratory of Electronics, M.I.T. January 1961. p. 199-212. Reprinted with minor editorial revisions under the title: "On context-free languages." Journal of the ACM. Vol. 13. p. 570-581.

Post, E. [1944]. "Recursively enumerable sets of positive integers and their decision problems." Bulletin of the American Math. Soc. Vol. 50. p. 284-316.

Post, E. [1946]. "A variant of a recursively unsolvable problem." Bulletin of the American Math. Soc. Vol. 52. p. 264-268.

Rabin, O. and Scott, D. [1959]. "Finite automata and their decision problems." IBM Journal of Res. and Dev.. Vol. 3. p. 114-125.

Redko, V. N. [1964]. "On defining relations for the algebra of events." (in Russian.) Ukrain. Mat. Ž.. Vol. 16. p. 120-126.

Riguet, J. [1962]. "Programmation et théories des catégories." (in French.) in Symbolic languages in data processing. p. 83-98. Gordon and Breach, New York.

Ross, D. T. [1964]. "On context and ambiguity in parsing." Comm. of the ACM. Vol. 7. p. 131-133.

Rudeanu, S. [1963]. "Axiomele laticior şi algebrelor Boolene." (in Rumanian.) Edition of the Rumanian Popular Republic Academy.

Salomaa, A. [1966]. "Two complete axiom systems for the algebra of regular events." Journal of the ACM. Vol. 13. p. 158-169.

Scheinberg, S. [1960]. "Note on the Boolean properties of context-free languages." Information and Control. Vol. 3. p. 372-375.

Schneider, F. W. and Johnson, G. D. [1964]. "Meta-3, a syntax directed compiler writing compiler to generate efficient code." Proceedings of the 19th national conference of the ACM. D1.5.

Schorre, D. V. [1963]. "A syntax-directed Smalgol for the 1401." 1963 ACM National Conference.

Schorre, D. V. [1964]. "Meta II. A syntax oriented compiler writing language." 1964 ACM National Conference. D1.3.

Schorre, D. V. [1965]. "A necessary and sufficient condition for a context-free grammar to be unambiguous." SDC report SP-2153.

Schützenberger, M. P. [1963]. "Context-free languages and push-down automata." Information and Control. Vol. 6. p. 246-264.

Schützenberger, M. P. [1964]. "Classification of Chomsky languages." in T. B. Steel (Ed.) [1966]. p. 100-102.

Schützenberger, M. P. [1966]. "Some remarks on acceptable sets of numbers." Paper presented at the August 1966 conference on the algebraic theory of machines, languages and semigroups.

Shannon, C. E. and McCarthy, J. (Eds.) [1956]. "Automata studies." Princeton University Press. Princeton.

Slagle, J. R. [1965]. "Experiments with a deductive question answering program." Comm. of the ACM. Vol. 8. p. 792-798.

Stearns, R. E. and Hartmanis, J. [1963]. "Regularity preserving modifications of regular expressions." Information and Control. Vol. 6. p. 55-69.

Steel, T. B. (Ed.) [1966]. "Formal language description languages for computer programming." North-Holland (1966). (Proceedings of the Baden IFIP conference of September 1964.)

Tarski, A. [1956]. "Logic, semantics, metamathematics." Clarendon Press, Oxford.

Weizenbaum, J. [1966]. "ELIZA - A computer program for the study of natural language communication between man and machine." Comm. of the ACM. Vol. 9. p. 36-45.

Wirth, N. and Weber, H. [1965]. "Euler, a generalization of Algol and its formal definition." Report CS20, Stanford University, April 27, 1965, and Comm. of the ACM. Vol. 9. p. 13-25 and 89-99.

Wirth, N. [1966]. "A programming language for the 360 computers." Report CS53, Stanford University, December 20, 1966.

Zwicky, A. M., Friedman, J., Hall, B. C., Walker, D. E. [1965]. "The Mitre syntactic analysis procedure for transformational grammars." Proceedings of the Fall Joint Computer Conference 1965. p. 317-326. Spartan Books, Baltimore.