CS 69

TRANSLATOR WRITING SYSTEMS

BY

JEROME A. FELDMAN

DAVID GRIES

TECHNICAL REPORT NO. CS 69
JUNE 9, 1967

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

TRANSLATOR WRITING SYSTEMS*

by

Jerome A. Feldman

David Gries

June 9, 1967

*Research supported in part by the U. S. Atomic Energy Commission

Preface

"...for all of it is contained in a long
poem which neither I, nor anyone else, has
ever succeeded in wading through."

The Devil
in Man and Superman,
George Bernard Shaw

This paper arose from a number of years of ambivalence towards efforts to produce automated translator writing systems. While much had been accomplished, the research seemed marred by xenophobia and loose scientific standards. The immediate impetus was an unsuccessful working conference in April 1967, which indicated that the situation was every bit as serious as we had imagined.

Many people have aided in the preparation of this report. The original draft of Section II.B3 (COGENT) was done by Fred Hansen and the draft of Section II.B4 (META systems) was done by Jeff Rulifson. David Gries prepared Sections I.B, II.A and III.B2 and made important contributions elsewhere. In spite of this help and more, this is in many ways a personal paper. The selection and arrangement of material and the tone of the entire paper are entirely Feldman's responsibility. Any resemblance of this work to a carefully prepared paper is due to the efforts of the typist, Miss Elaine Callahan.

This work was supported in part by the U. S. Atomic Energy Commission.

TABLE OF CONTENTS

I. Preliminaries

I.A. Introduction

Compiler writing has long been a glamour field within programming and has a well developed folklore [Knu 62, Ros 64b]. More recently, the attention of researchers has been directed toward various schemes for automating different parts of the compiler writer's task. This paper contains neither a history of nor an introduction to these developments; the references at the end of this section provide what introductory material there is in the literature. Although we will make comparisons between individual systems and between various techniques, this is certainly not a consumer's guide to translator writing systems. Our intended purpose is to carefully consider the existing work in an attempt to form a unified scientific basis for future research.

Compiler writing is a large programming task with many aspects and it is not surprising that many different techniques have been proposed as aids to compiler writers. In a very real sense, any system feature (e.g. trace, edit) which helps one produce large programs is a compiler-writing tool. This remark will become relevant as we examine various systems for their specificity to compiler writing. Since there has been no general agreement on terminology, we will define a term Translator Writing System (TWS) to denote the programs and proposed programs considered here. A translator written in a TWS might be an interpreter, a compiler, an incremental compiler, or an assembler.

Relatively few existing translators have been implemented with the aid of a TWS; the most common techniques involve the use of (macro-) assemblers or conventional algebraic or list-processing languages. There have been claims that this proves that TWS research is therefore a failure, but we find this argument unconvincing. For one thing, there is little in the work of commercial compiler writers which would lead one to believe in their infinite wisdom. Further, commercial translators often involve the informal use of TWS concepts and the division between conventional and TWS systems is not always sharp.

It is even more difficult to classify the various TWS developments in a meaningful way. We have chosen to divide the work into four categories: those efforts concerned only with syntax, syntax-directed symbol manipulating systems, macro processors of various kinds, and compiler-compilers. The emphasis throughout is on recent work; a fairly complete (though abominably edited) survey of earlier work may be found in Burkhardt [Burk 65].

Unfortunately, one cannot understand the development of TWS research without some knowledge of its sociology. This is doubly unfortunate because neither the intercommunication nor the publication behavior has been inspiring. One might be able to attribute this to the great financial potential of a successful (i.e. accepted) TWS. In any event, one must use care in reading much of the literature on TWS proposals.

One common way to begin a TWS paper is with a statement like "Most of the existing TWS systems lack property X, which is essential." The author of such a statement rarely describes which systems have property X, how they compare to his work, or even why property X is essential. This kind of oversight occurs in other contexts and may simply be the result of not reading the literature. In any event, there is a tremendous amount of rediscovery and very little cross-referencing within the field.

Another statement often found runs something like "Our system has been used to implement N compilers on M different computers." This rarely means that the TWS presented in that paper was used just as presented and was completely adequate to the task. For example, essentially no existing language can be adequately handled by any of the syntax mechanisms mentioned in the TWS literature (cf. Section III.B2, Floyd [Flo 62b]). One could make a much more significant contribution by carefully describing both the strengths and weaknesses of one's work. To some extent this is due to referees and reviewers who seem to judge a paper on what it claims to have done.

Another flaw has been the prevalence of a more-mathematical-than-thou attitude. The worst form of this attitude seems to come from confusing mathematical notation with mathematics. However, even the serious work on mathematical models (Section III.B) seems more concerned with applying known results than with developing new ones. Many basic concepts in programming (e.g. the storage location, transfer of control) have not been adequately formalized.

These criticisms should not be construed as a complete rejection of the field of TWS research. It has been and continues to be one of the most active and fruitful areas of Computer Science. Many of the outstanding workers in Computer Science have contributed to the TWS development, and even the bad work seems well-intentioned. The problem is that a lack of communication and a tendency towards over-enthusiastic reporting has marred the record.

Before describing the particular systems in the next section, we should say a few things about the general problem of translator writing. We will concentrate on compilers, because these contain all the essential problems found in assemblers and interpreters. Considering the amount of effort that has gone into compiler writing, there has been relatively little published on the subject. The history [Knu 62, Ros 64b] and syntax methodology [Flo 64b] have been fairly well covered, but very little has been said about code generation or interactions with the operating system. This lack of literature has forced TWS designers to try to formalize systems which were largely intuitive and had never been described carefully. A further difficulty is that there are no accepted standards of performance for translators, except such shibboleths as efficiency. The efficiency of a compiler depends on its ability to conserve both time and space, while translating and during execution of the object program. The error detection and recovery facilities, the editing facilities and the speed of recompilation have important effects on efficiency. Since all these goals are not mutually compatible, one

can expect no absolute measure of efficiency for compilers. The designers of the TWS considered here have varied considerably in their preferred choice of compromises.

We have divided the review of TWS (Chapter II) into four major parts. The first describes the efforts which are primarily aimed at automatic syntax techniques. The second section deals with systems where the syntax processing is augmented by a symbol manipulation language for producing output. The third section treats the related topics of extendible compilers and meta-assemblers. The final section describes systems which attempt to provide specific techniques for many of the post-syntactic problems of translator writing.

The related topics discussed in Chapter III have been chosen to complement the review sections and are treated in much less detail. The treatments of the other uses of syntax-directed techniques and related mathematical studies are aimed at elucidating their relationships with TWS efforts. Finally, we sketch a number of potentially fruitful research topics related to the future development of translator writing systems. The bibliography is arranged alphabetically with references pertinent to a particular section listed at the end of that section.

References for I.A

The Communications of the ACM, and to a lesser extent
The Computer Journal of the British Computer Society are the major journals for publications on translator writing.

See especially

Comm. ACM 4 (Jan. 61)
Comm. ACM 7 (Feb. 64)
Comm. ACM 9 (Mar. 66)

Other general references:

Che 64a, Flo 64b, Hals 62, Knu 62, Ran 64, Ros 64b, Weg 62, Wil 64b.

- Formal descriptions of various programming languages:

Bac 59, Ber 62, Brook 61, Bro 63,
EvA 64, Gor 61, IBM 66, Naur 60, 63b,
Rab 62, Samm 61, Shaw 63, Tay 61,
Wir 66b, 66c.

I.B. Terminology

One of the minor irritants in the TWS literature is the lack of uniform notation. In order to make this paper more readable, we have taken the liberty to change the symbols and sometimes the syntax used by various authors. For the discussions on syntax we have decided on the notation used by Ginsburg ([Gin 66a], pages 8,9). However, as an (non-conflicting) alternative, the notation of the ALGOL report [Naur 63b] and of the syntactic meta-language Backus-Naur Form (BNF) is used where it is more readable.

Many terms will be used in both a formal and an informal sense; the default sense is the informal except in Sections II.A and III.B. The formal definitions of such terms as "syntax" and "semantics" are not generally agreed upon and we will discuss them further in Section III.B. Informally, we consider syntax to be the specification of well-formed statements in a language and semantics to be essentially anything else.

In general, a language, L , will be some subset of the set of all strings of symbols from an alphabet $\mathcal{A}$. The specification of which strings are in the language L (syntax of L) will be described in a syntactic meta-language. The syntactic meta-language will be procedural and will describe either an algorithm for generating strings of L (synthetic syntax) or for recognizing if a string over $\mathcal{A}$ is in L (analytic syntax). Any process utilizing a non-trivial analytic syntax will be called syntax-directed.

An individual statement in a syntactic meta-language will be called a production. We have found no way to overcome the unfortunate

use of the word "production" in the TWS literature. The term was originally used in mathematical logic to describe string transformations which are more general than any considered here and which can be considered both analytic and synthetic. In going through a series of applications in Computer Science the term "productions" began to be applied to a set of rules for recognizing (reducing) a program (cf. Section II.A5). This analytic meta-language is widely known as "production language" even though its statements are reductions and will be so described here.

A syntactic meta-language may include symbols not in $\mathcal{A}$ (non-terminal symbols) which are used in defining a grammar. These will normally be enclosed in angular brackets '<' and '>' as in the Algol report, and will appear informally in the text as well as in formal syntax rules. In the sections dealing more formally with syntax (II.A and III.B1) we will bow to clarity and convention and omit the brackets. These sections will also require a fairly extensive technical vocabulary used less formally in the other sections.

For the formal discussions, characters or symbols are represented by Latin capitals S,T,U,..., strings of symbols by lower case Latin letters u, v, w, x, y, z,... . The set of all strings of finite length (including the empty string ϵ) over a set of symbols $\mathcal{V}$ is denoted by $\mathcal{V}^*$. If z = xy is a string, x is a head and y a tail of z. A production $U \rightarrow u$ is an ordered pair consisting of a symbol U and a nonempty string u. U is the left and u the right part of the production. A set of productions is called a (synthetic) grammar.

Given a grammar, we say that $w \Rightarrow v$ if there is a production $U \rightarrow u$ and strings x and y (possibly empty - the empty string is represented by ϵ) such that $w = xUy$ and $v = xuy$. "$\overset{*}{\Rightarrow}$" is the transitive closure of "$\Rightarrow$"; $w \overset{*}{\Rightarrow} v$ if $w = w_o$, $w_o \rightarrow w_1, \ldots, w_{i-1} \rightarrow w_i$ $(i \geq 1)$ and $w_i = v$. If $w \overset{*}{\Rightarrow} v$, v is called a <u>derivative</u> of w. A set of productions P is called a phrase structure grammar if P contains exactly one symbol $\overline{U}$ which appears only on the left of "$\rightarrow$" and a nonempty set $\mathcal{A}$ (the alphabet) of symbols which occur only to the right of "$\rightarrow$", called <u>terminal symbols</u> and always denoted by T, T_1, T_2, ... The symbols which occur on the left of "$\rightarrow$" are called <u>nonterminal symbols</u> and are denoted by $\overline{U}$, U, U_1, U_2, ... The derivatives of U are called <u>sentential forms</u> and the sentential forms consisting only of terminal symbols are called <u>sentences</u> of the <u>language</u> L_P determined by P. If the grammar represents a programming language, the sentences are just the <u>programs</u> of that language.

In order to be able to recognize the beginning and end of a sentence x, one usually puts a special marker $\perp$ at the beginning and end of it. Formally we add the production <Program> $\rightarrow \perp \overline{U} \perp$ to the grammar.

Figure 1 contains, as an example, a grammar which will be used throughout the rest of the paper. The sentences of this grammar are the set of all arithmetic expressions (enclosed by $\perp$ and $\perp$) consisting of the operand I, the binary operations * and + (* takes precedence over +), and parentheses.

<Program> → ⊥ E ⊥

E → T

E → E + T

T → P

T → T * P

P → (E)

P → I

Nonterminal symbols: <Program> E T P.

Terminal symbols: I ( ) + * ⊥ .

Fig. 1. Example of a grammar

The sentential form ⊥P + T * P⊥ has at least two derivations (according to the grammar of Fig. 1):

(a) <Program>⇒⊥E⊥⇒⊥E + T⊥⇒⊥T + T⊥⇒⊥P + T⊥⇒⊥P + T * P⊥

(b) <Program>⇒⊥E⊥⇒⊥E + T⊥⇒⊥E + T * P⊥⇒⊥T + T * P⊥⇒

⊥P + T * P⊥

Both have the same syntax tree:

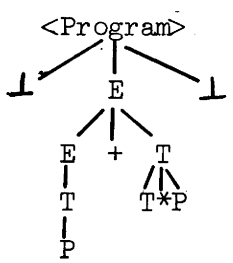


Fig. 2. A syntax tree

A sentence which has two or more derivations with different syntax trees is called ambiguous. A grammar (and also the corresponding language) is called ambiguous if it contains an ambiguous sentence.

Let us suppose for the moment that a grammar is unambiguous (as is the grammar of Fig. 1). One can generate sentences of a language by deriving them from the symbol <Program>. When given a probable sentence, though, one must work backwards and produce the opposite of a derivation. A parse of a sentential form of a language is a sequence of productions used to reduce the sentential form to <Program>. Two parses of ⊥P + T * P⊥ corresponding to the above two derivations are:

(a) $T \rightarrow T * P$, $T \rightarrow P$, $E \rightarrow T$, $E \rightarrow E + T$, <Program> $\rightarrow \perp E \perp$;

(b) $T \rightarrow P$, $E \rightarrow T$, $T \rightarrow T * P$, $E \rightarrow E + T$, <Program> $\rightarrow \perp E \perp$.

When parsing a sentential form, reductions are made by replacing a substring which is the right part of a production by the corresponding left side. In other words, given the syntax tree, a reduction consists of cutting off (pruning) a set of adjacent leaves forming a complete branch. Thus, in Figure 2, we could "prune" the branches "P" and "T * P" (make reductions $T \rightarrow P$ and $T \rightarrow T * P$).

In order to avoid the unimportant differences between parses which are the same except for the order in which the reductions are executed, we designate one as the canonical parse. Given a sentential form and its syntax tree, the canonical parse is the one which always prunes the leftmost branch first. Such a leftmost branch we call the handle ([Knu 65]). Thus for the trees (a), (b), (c) in Figure 3, the handles

are T, T * P and E + T respectively. (b) is the result of pruning the handle "T" of (a), while (c) arises by pruning the handle "T * P" of (b).

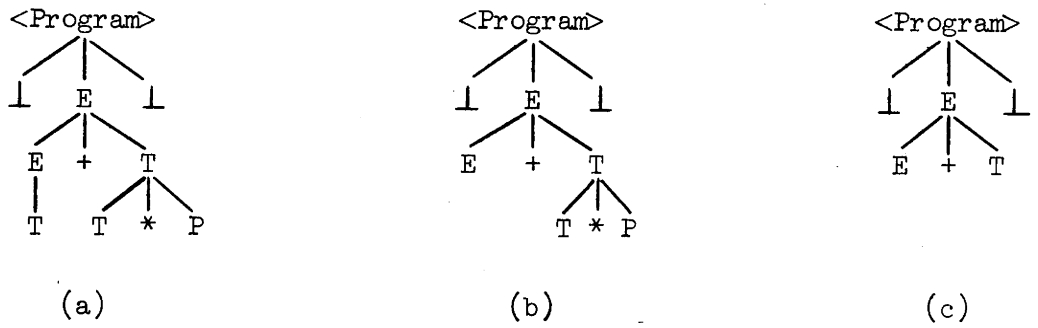


Figure 3

A left-right recognizer, or parsing algorithm, always finds a canonical parse. Of course, if a sentence is ambiguous, it has more than one canonical parse -- one for each syntax tree. A left-right recognizer will find only one of these. In Section II.A certain recognizers will be discussed which can be constructed automatically from the grammar if the grammar satisfies certain restrictions. Part of the duty of the construction algorithm will be to check the definition of the programming language by verifying that the grammar is indeed unambiguous.

When given just a string it is sometimes difficult to detect a handle. For instance, with the string ⊥E + T * P⊥ , according to the grammar of Figure 1, E + T is not the handle. Reducing E + T to E yields ⊥E * P⊥, which is no longer a sentential form. The handle in this case is T * P. Most of the recognizers to be discussed will have means for detecting the handle, so that wrong reductions will not occur.

Some confusion has arisen over the terms "top-down" and "bottom-up". These refer to two different methods of recognizing or parsing a sentence of a language. Part of the confusion has arisen because people draw their syntax trees differently - for example, the tree for the string ⊥T + T ⊥ can be written as in (a) or (b) of Figure 4.

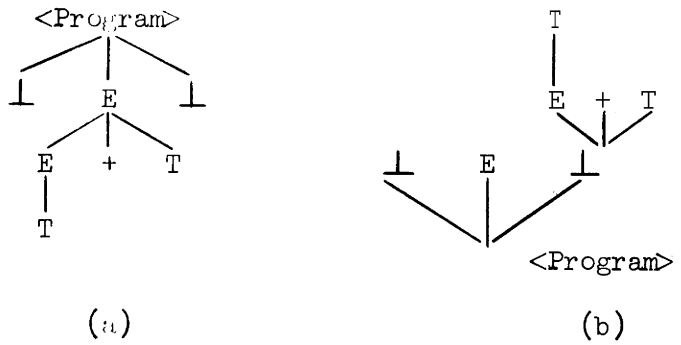


Figure 4

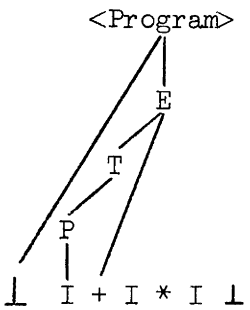
We will use (a) throughout the paper.

The other part of the confusion has arisen because the two concepts have actually merged as recognizers have become more sophisticated. We will try to clear up this confusion here.

A pure top-down recognizer is entirely goal-oriented. The main goal is of course the distinguished nonterminal symbol <Program> -- a prediction is made that the string to be recognized is actually a program. The next step is to see whether the string can be reduced to the left part $S_1S_2...S_n$ of some production <Program> → $S_1S_2...S_n$. Thus, if S_1 is a terminal symbol, the string must begin with the same terminal symbol. If S_1 is nonterminal, our first subgoal is to see whether some head of the string may be reduced to S_1. At any step, if

some subgoal is not met, the failure is reported to the next higher level, which must try another alternative.

This type of recognizer gets its name from the way the syntax tree is being constructed. At any point of the parse, certain connections have been made (perhaps wrongly) by constructing the tree from the top node and reading down to the string (Fig. 5).

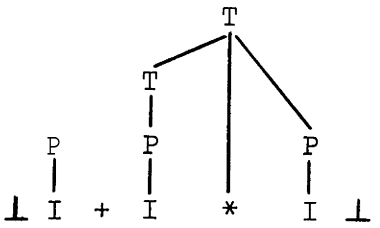


Partial Top-Down Parse

Figure 5

If some of these connections are wrong -- a subgoal cannot be met -- some of the connections must be erased and other alternatives tried (backtracking or backup). A top-down recognizer may of course be programmed in many different ways -- as recursive subroutines, as a single routine working with a stack, etc. The significant feature is that it is goal-oriented.

In contrast, a pure bottom-up recognizer has essentially no goals (except of course the implicit goal <Program>). The string is searched for substrings which are right parts of productions. These are then replaced -- perhaps wrongly if they are not really handles -- by the corresponding left side. This may be illustrated by Fig. 6.



Partial Bottom-Up Parse

Figure 6

In order to reduce backup, modified top-down recognizers have been introduced. For instance, before starting out on a new subgoal, a modified top-down recognizer may see whether some derivative of the subgoal may actually start with the initial symbol of the substring in question (look ahead) or whether the subgoal could occur with the partial tree (memory). Examples of modified top-down recognizers are those in [Ir 61] and [War 64]. Most of the syntax directed symbol processors (II.B) used modified top-down recognizers.

Similarly, as will be seen in II.A, bottom-up recognizers can be constructed (for suitable grammars) which eliminate backup entirely. Such modified bottom-up recognizers generally look to the left and right of a possible handle to see if it really is a handle or not.

It is these modifications which have led to the (con)fusion of the two concepts. It is sometimes very difficult to tell whether a particular recognizer is bottom-up or top-down. For instance, a production language recognizer as generated by Earley's algorithm (cf. Section II.A5) has some of the properties of both. If a recognizer has any explicit goals and subgoals to meet, we tend to call it

(modified) top-down, Since it is essentially goal-oriented.

Most of the remaining terminology should be familiar to anyone with general knowledge of Computer Science. We will use a few data-structure terms which require definition. The term list structure will be used generically to describe any programming system making significant use of pointers (links) and dynamic storage allocation. A list structure which does not allow more than one path between any two nodes is a tree. A list structure which explicitly allows general connectivity is called a plex. The term plex also loosely implies that each element is a block of storage containing several (often two-way) links, We will also use the terms LIFO (last-in-first-out) and FIFO (first-in-first-out) as general rules for handling sequential information. For those who worry about such things, the symbol TWS will be used as the singular, plural, possessive and adjectival forms of "Translator Writing Systems."

II. Review of Current Translator Writing Systems

II.A. Recognizers which are constructed mechanically

In this section, several practical techniques for parsing, or recognizing, sentences of languages defined by grammars will be described and evaluated. A "practical" technique is one that has been or is being used to write a compiler. Each of these recognizers has a second important property -- there is an algorithm for constructing, or generating, it from a suitable grammar of the language, either in the form of tables to be used by a set of basic routines or in the form of a program. We will call such an algorithm a constructor.

This property of automatic generation is very important to the compiler writer. Most of the constructors check the grammar for unambiguity before actually constructing the recognizer -- a decided advantage. Automatic construction of parts of a compiler also means less work, leaving more time for considerations such as code optimization. Moreover, the automatic construction will guarantee that the recognizer follows the formal syntax.

Unfortunately, these recognizers and their constructors do not solve all problems.- First of all, much of the syntax of a language can not be defined by existing grammars. Secondly, semantics form a much larger and more difficult part of a programming language -- often either the grammar or the generated recognizer must be changed in order to fit in semantics properly. Thirdly, while a technique may be theoretically very nice, it may not be practical. The usual programming language grammar may

for some reason not be accepted by the constructor of some technique. If not, the grammar must be altered substantially or another technique used.

We note in passing that the "efficiency" of several recognizers have been compared by Griffiths and Petrick [Grif 65]. While theoretically interesting, this comparison is of no practical value, since it is based mainly on the efficiency of Turing machines corresponding to each of the recognizers. We are interested in the practical problems of actual space used and time consumed, as well as the problems of adequacy mentioned in the last paragraph,

Some of the recognizers discussed here have been used in many compilers by many people; we cannot list references to all of them. For each recognizer we have given one reference to a paper where not only the recognizer, but also its constructor, is discussed. Some theoretically interesting recognizers which can be mechanically constructed, as well as formal properties of systems described here, are discussed briefly in section III.B1.

Top-down methods will not be discussed here, although they are used in some compilers. They are in general less efficient than the recognizers to be discussed, since some amount of back-up is almost always necessary. See [War 61] and [Ir 61] for details of compilers which use modified top-down recognizers. [Che 64c] is a good tutorial paper on the use of top-down

recognizers in compiling, while [Flo 64b] also contains a good description of the technique.

The grammar in Figure 1 (page 10) will be used throughout this section as an example. At this point it may be advisable to briefly review section I.B for definitions and notations.

1. Operator Precedence (Floyd [Flo 63])

The grammar is restricted to an operator grammar; no production may be of the form $U \rightarrow xU_1U_2y$ for some strings x and y and nonterminals U_1, U_2. This means that no sentential form contains two adjacent nonterminal symbols. This is not a serious restriction; many programming language grammars are already in this form. Most programming languages grammars which are not, can be made into operator grammars without essentially disturbing the structure of a sentence.

During the parse of a sentence $T_1 \ldots T_m$, a LIFO stack will contain symbols $S_0S_1 \ldots S_i$ of the partially reduced string $S_0S_1 \ldots S_iT_iT_{ij+1} \ldots T_m$. At any step, it is necessary to be able to tell solely from the symbols S_{i-1}, S_i and T_j whether

1) S_i is the tail of a handle (the leftmost substring for which a reduction may be made) in the stack; or whether
2) S_i is not the tail of a handle and T_j must be pushed into the stack.

In order to do this, the following three relations are defined between terminal symbols T_1 and T_2 of an operator grammar.

1) $T_1 \doteq T_2$ if there is a production $U \to xT_1T_2y$ or $U \to xT_1U_1T_2y$ where U_1 is nonterminal.

2) $T_1 \gtrdot T_2$ if there is a production $U \to xU_1T_2y$ and a derivation $U_1 \overset{*}{\Longrightarrow} zT_1$ or $U_1 \overset{*}{\Longrightarrow} zT_1U_2$ for some z and U_2.

3) $T_1 \lessdot T_2$ if there is a production $U \to xT_1U_1y$ and a derivation $U_1 \overset{*}{\Longrightarrow} T_2z$ or $U_1 \overset{*}{\Longrightarrow} U_2T_2z$ for some z and U_2.

If at most one relation holds between any ordered pair T_1, T_2 of terminal symbols, then the grammar is called an operator precedence grammar and the language an operator precedence language.

In an operator precedence language, these unique relations may be used quite simply for detecting a handle (or any right part of a production which may be reduced). Suppose T_oxT is a substring of a sentential form, and suppose that the following relations hold between T_o, the terminal symbols $T_1, T_2, \ldots, T_n (n \geq 1)$ of x, and T:

$$T_o \lessdot T_1 \doteq T_2 \doteq \ldots \doteq T_n \gtrdot T .$$

(Note that nonterminals of x play no role here). Then x is what Floyd calls a prime phrase; it is either the right part of a production $U \to x$, or there is a production

$$U \to x'$$

where $x' \overset{*}{\Longrightarrow} x$ and the only productions in the derivation $x' \overset{*}{\Longrightarrow} x$

are of the form $U_i \rightarrow U..$ The substring x may therefore be replaced by the nonterminal U, yielding T_oUT.

The parse of a sentence (or program) is quite straightforward. Symbols are pushed into the stack until the relation $T_n \gtrdot T$ holds between the top terminal stack symbol T_n and the next incoming symbol T. If the program is indeed a sentence of the language, the top stack elements then hold a string T_ox as described above. One searches back in the stack, using the relations, to find T_o and the beginning of x. x is then a handle and can then be reduced to some U, yielding T_oU in the stack. The process is then repeated by comparing T_o with T.

The relations $\gtrdot$, $\doteq$ and $\lessdot$ can be kept in an $\ell x \ell$ matrix, where ℓ is the number of terminal symbols of the grammar. (In [Flo 63], the matrix for an ALGOL-like language is about 35 x 35). The comparison is then just a test of the relation in the matrix element defined by the row corresponding to the top stack terminal symbol and the column corresponding to the incoming symbol.

The space needed for the relations may be reduced to two vectors of length ℓ if two precedence functions f(T) and g(T) can be found such that $T_1 \lessdot T_2$ implies $f(T_1) < g(T_2)$, $T_1 \doteq T_2$ implies $f(T_1) = g(T_2)$ and $T_1 \gtrdot T_2$ implies $f(T_1) > g(T_2)$. These functions can usually be found. Floyd outlines the algorithm for finding the matrix of precedence relations and the functions f and g (if they exist). For the language of Figure 1 the following precedence matrix and functions are generated:

	(	I	*	+	⊥	)
)			⋗	⋗	⋗	⋗
I			⋗	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗	⋗	⋗
+	⋖	⋖	⋖	⋗	⋗	⋗
(	⋖	⋖	⋖	⋖		≐
⊥	⋖	⋖	⋖	⋖	≐	

T	f(T)	g(T)
)	5	1
I	5	6
*	5	4
+	3	2
(	1	6
⊥	1	1

Figure 7 gives the algorithm for recognizing a sentence of an operator precedence grammar. The precedence relations will have been produced from the grammar by the constructor.

Semantic routines may only be called when a prime phrase, or handle, is to be reduced. A separate routine is written to process each different handle. This may mean that the grammar has to be altered to allow the correct semantic interpretation. For instance, the production

<COND> → IF <BE> THEN <EXPR> ELSE <EXPR>

would have to be explicitly written as

<IFCL> → IF <BE>

<IF-THEN> → <IFCL> THEN <EXPR>

<COND> → <IF-THEN> ELSE <EXPR>

so that the tests and jumps may be inserted at the proper places by semantic routines.

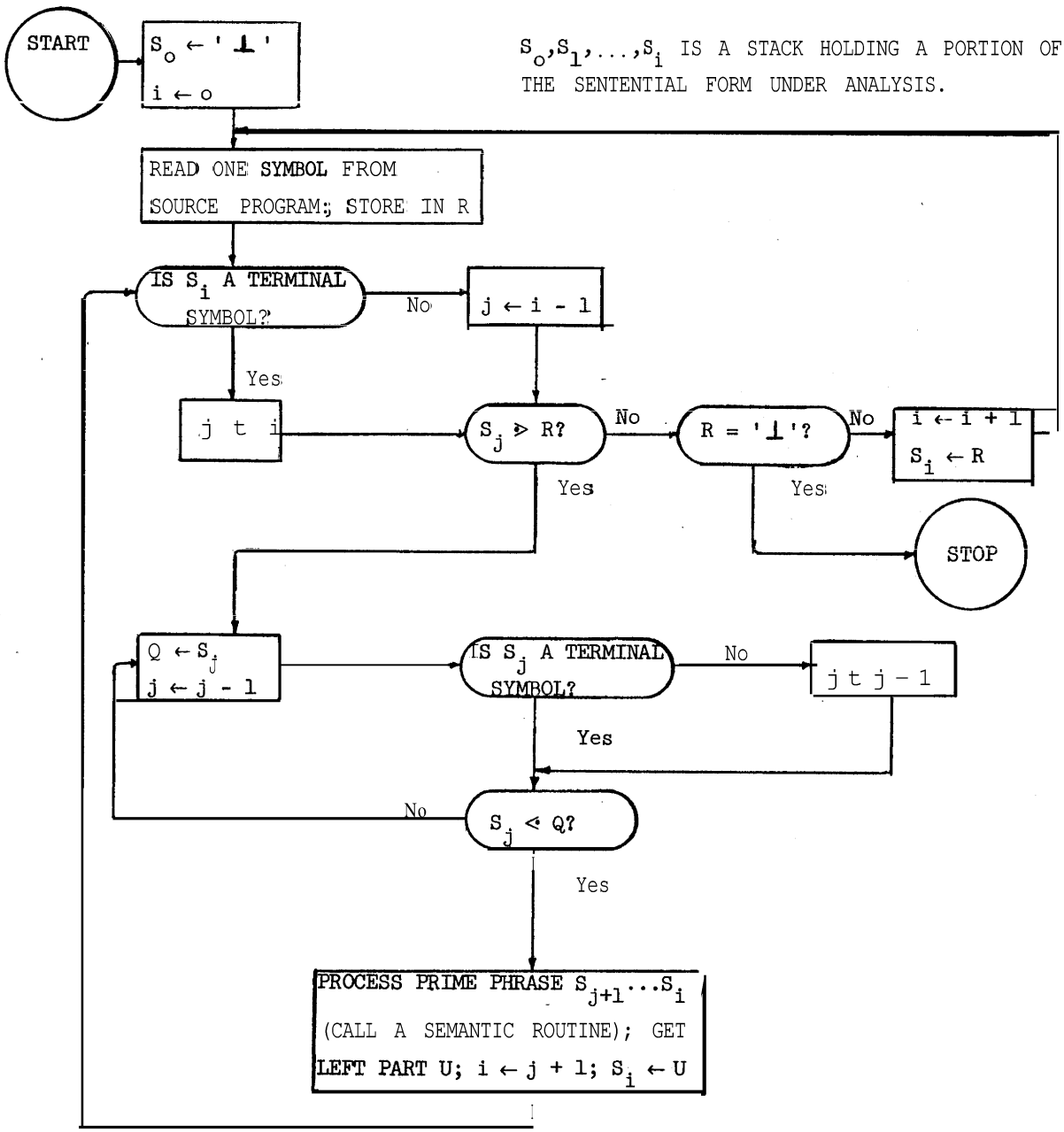


Fig. 7. **Recognizer** using operator precedences

However, the revised grammar to the generator will not in all likelihood be essentially different from the original reference grammar of the language (see for example Floyd's language in [Flo 63]). Although to our knowledge no compiler contains a mechanically constructed recognizer of this type, the precedence technique has been used in quite a few ALGOL, MAD and FORTRAN compilers and will be used in many more. The technique is easy to understand, flexible, and very efficient.

It is rather difficult to figure out a good error recovery scheme if the functions f and g are used, since an error can be detected only when a probable handle turns out not to be one. With the full matrix, an error is detected whenever no relation exists between the top terminal stack symbol and the incoming symbol. Therefore the functions should be used only if a previous pass has provided a complete syntax check.

One objection to this technique is that the language may still contain ambiguous sentences. The structure of the parse tree is unambiguous if the grammar is a precedence grammar, but the names of the nodes may not be. For a prime phrase x there may exist more than one nonterminal to which it may be reduced. This objection is partly answered by the fact that the nonterminals are usually manipulated by semantic routines anyway, and not so much by the syntax. The syntax defines the structure; whether a node is named (say) "integer expression" or "real expression" is a semantic matter.

2. Precedence Languages (Wirth and Weber [Wir 66c])

Wirth and Weber modified Floyd's precedence concept. The grammar is not restricted to an operator grammar and the relations $\lessdot$, $\doteq$ and $\gtrdot$ may hold between all pairs S_1, S_2 of symbols:

1) $S_1 \doteq S_2$ if there is a production $U \to xS_1S_2y$

2) $S_1 \gtrdot S_2$ if there is a production $U \to xU_1S_2y$ (or $U \to xU_1U_2y$) and a derivation $U_1 \overset{*}{\Rightarrow} zS_1$ (and $U_2 \overset{*}{\Rightarrow} S_2w$) for some z.

3) $S_1 \lessdot S_2$ if there is a production $U \to xS_1U_1y$ and a derivation $U_1 \overset{*}{\Rightarrow} S_2z$ for some z.

If at most one relation holds between any pair S_1, S_2 of symbols, and if each right part is the right part of only one production, then the grammar is called a precedence grammar and the language a precedence language. Any sentence of a precedence language has a unique canonical parse. As long as either the relation $\lessdot$ or $\doteq$ holds between the top stack symbol S_i and the incoming symbol T, T is pushed into the stack. When $S_i \gtrdot T$, then the stack-is searched downward for the configuration

$$S_{j-1} \lessdot S_j \doteq \ldots \doteq S_{i-1} \doteq S_i \quad .$$

The handle $S_j \ldots S_i$ is then replaced by the left part U of the unique production $U ::= S_j \ldots S_i$ (if the program is a sentence). The main difference between this technique and Floyd's is that

the relations may hold between any two symbols, and not just terminal symbols. Algorithms for generating the matrix of precedences and functions f and g similar to Floyd's are given in [Wir 66c].

For the grammar of Figure 1 relations + $\doteq$ T, + $\lessdot$ T; $\perp \doteq$ E, $\perp \lessdot$ E; and ($\doteq$ E, ($\lessdot$ E hold. These conflicts may be disposed of by changing the grammar to the following equivalent one:

<Program> → $\perp$ E' $\perp$

E' → E

E → T'

E → E + T'

T' → T

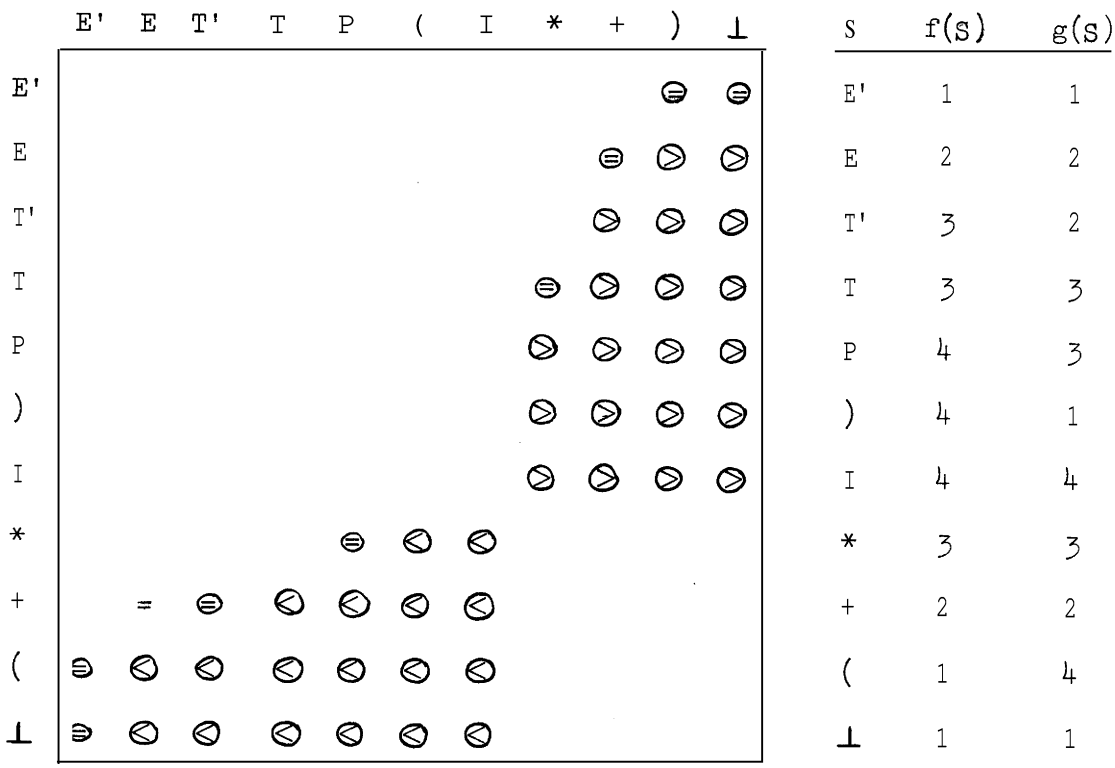
T → P

T → T * P

P → (E')

P → I

The precedence matrix and functions are then



S	f(S)	g(S)
E'	1	1
E	2	2
T'	3	2
T	3	3
P	4	3
)	4	1
I	4	4
*	3	3
+	2	2
(	1	4
⊥	1	1

As with Floyd's recognizer, one may use either the precedence matrix or the functions f and g. The matrix is much larger than Floyd's (over 70 x 70 for ALGOL), since the relations may hold between any two symbols. As with Floyd's recognizer, semantic routines may-only be -called when a handle is detected.

Theoretically, the technique is very sound and efficient. Since the relations may hold between any two symbols, it is in a sense more reliable than Floyd's; if the precedence relations are unique, one knows that a unique canonical parse exists for each sentence. In practice, however, one must manipulate a grammar for an average programming language considerably before it is a precedence grammar. The reason is that not enough

context is used in determining the precedence relations; very often more than one relation holds between two symbols. It may be necessary to insert intermediate productions (as in the above example) or even to use a different symbol for (say) a comma depending on its context. A prescanner must then be changed to look at the context and decide which internal symbol to use for each comma. The final grammar could not be presented to a programmer as a reference to the language.

This recognizer and its constructor have been used to write a sophisticated assembler, PL 360, ([Wir 66a]) and a compiler for a proposed successor to ALGOL [Wir66b]) on the IBM 360.

3. Extended Precedence (McKeeman [McKee 66])

McKeeman extended Wirth's concept by first of all separating the precedence matrix into two tables - one for looking for the tail, the other for the head of a handle - and secondly by having the recognizer look at more context so that fewer precedence conflicts arise. The constructor will therefore accept a much wider class of grammars.

a) The top two symbols S_{i-1}, S_i of the stack and T, the incoming symbol, are used to decide whether T should be put into the stack, or whether S_i is the tail of a handle and a reduction should take place.

b) Similarly, in order to go back in the stack to find the initial symbol of the handle, three symbols instead of two are used.

This technique should be compared with the one proposed by Eickel et al. [Ei 63]. See Section III.B1. In practice, the number of different triples is too large (over 10,000). Also, in most cases two symbols suffice to determine uniquely what is to be done. McKeeman's recognizer compromises by using Wirth's two-argument precedences whenever possible and switching to triples only when necessary. When looking to the right to see if the stack contains a handle, a matrix MATRIX1 with entries $\lessdot$ ($\lessdot$ or $\doteq$) , $\gtrdot$, and $\lessgtr$ ($\gtrdot$ and either $\doteq$ or $\lessdot$) is used. If $\lessgtr$ holds between the top stack symbol S_i and the incoming symbol T then a list of triples is searched to find the value of the following three-argument function P1;

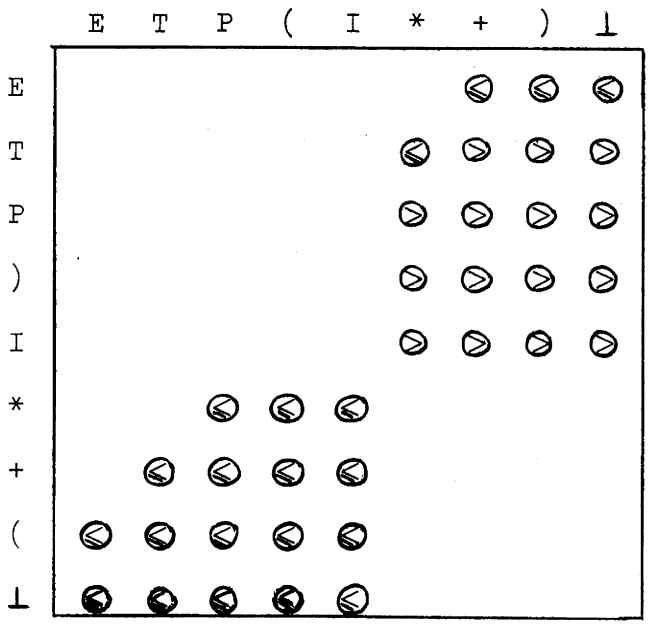
$$P1(S_{i-1},S_i,T) = \begin{cases} \text{TRUE} & S_i \gtrdot T \ (S_i \text{ is tail of a handle}) \text{ in the context } S_{i-1}S_iT \\ \text{FALSE} & T_i \lessdot S \text{ holds in the context } S_{i-1}S_iT \end{cases}$$

Of course this function must be single valued for all triples, and the constructor checks this. A similar matrix MATRIX2 with entries $\gtrless$, $\lessdot$ and $\leqslant$ ($\lessdot$ and either $\doteq$ or $\gtrdot$) and a function P2 are used when looking in the stack for the initial symbol of the handle:

$$P2(S_{j-1},S_j,S_{j+1}) = \begin{cases} \text{TRUE} & S_{j-1} \lessdot S_j \ (S_j \text{ is head of a handle}) \\ & \text{in the context } S_{j-1}S_jS_{j+1} \\ \text{FALSE} & S_{j-1} \gtrless S_j \text{ holds in the context} \\ & S_{j-1}S_jS_{j+1} \ . \end{cases}$$

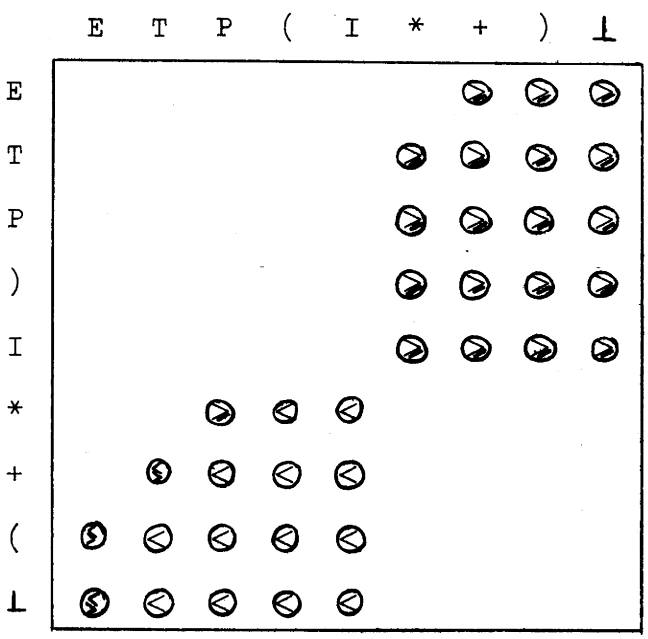
For the grammar in Figure 1 the following matrices and functions P1 and P2 are generated:

MATRIX1



Function P1 not necessary, since the conflict ⋖ does not arise.

MATRIX2



Function P2 (only necessary triples which also form valid substrings of some sentential form listed)

P2(⊥,E,+) = TRUE
P2(⊥,E,⊥) = FALSE
P2((,E,+) = TRUE
P2((,E,)) = FALSE
P2(+,T,*) = TRUE
P2(+,T,+) = FALSE
P2(+,T,)) = FALSE
P2(+,T,⊥) = FALSE

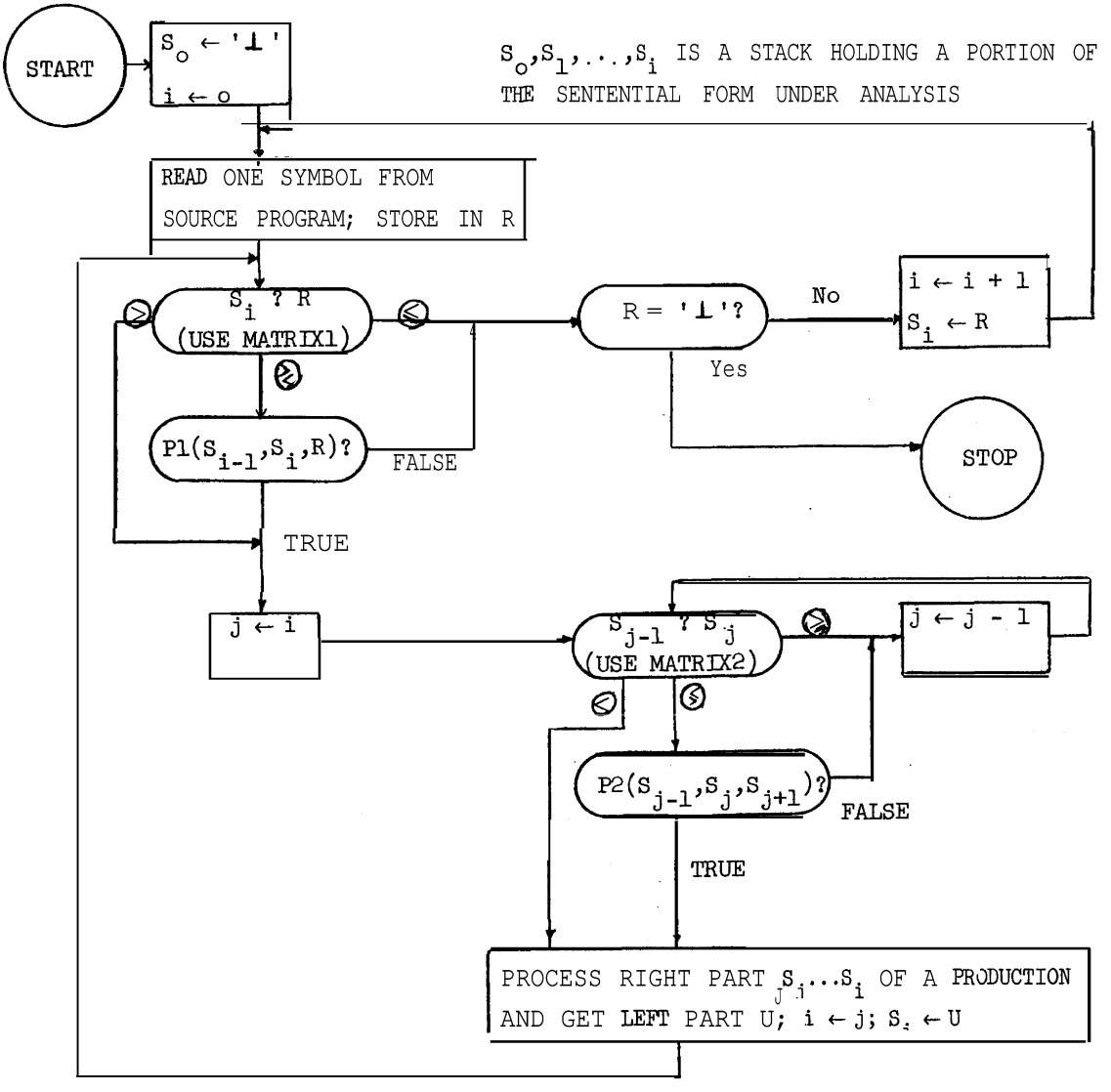


Fig. 8 **Recognizer** using **Wirth** precedences plus **McKeeman** triples

The recognizer which uses the matrices and triples is given in Figure 8. Of course functions f and g may be used as in Floyd's or Wirth's case, if they can be found.

The use of triples helps avoid most of the unpleasantries one encounters with precedence grammars. But, again, semantic routines may only be called when a handle is detected, so that it may be necessary to alter the grammar for this reason. McKeeman is writing a compiler for a subset of PL1 (in itself) on the IBM 360 using this technique. He expects to use the final grammar as a standard reference for the PL1 subset.

4. Transition Matrices (Samelson and Bauer [Sam 60], Gries [Grie 67a])

This technique for parsing sentences was first introduced by Samelson and Bauer. It has been used by the Europeans for writing a number of ALGOL compilers. NELIAC compilers use it under the name CO-NO tables [Hals 62]. In [Grie 67a] a constructor was written for the recognizer. The grammar is restricted to an operator grammar. Essentially one gets a transition matrix by replacing the precedence relations in a Floyd precedence matrix by addresses, or numbers, of subroutines which perform the necessary stack reductions or push the incoming symbol onto the stack.

The constructor uses the following scheme to reduce the number of elements in the stack which must be tested in order to find the beginning of the handle. Suppose that

(4.1) <COND> → IF <BE> THEN <EXPR> ELSE <EXPR>

is a production of the grammar. At one point in parsing a sentence the stack should look like (say):

← BOTTOM	STACK	TOP
.....+	IF <BE>	THEN .

Instead, a representation for "IF <BE> THEN", say "IBT", is put in the stack -- the stack would look like

← BOTTOM	STACK	TOP
~~+	"IF <BE> THEN"	

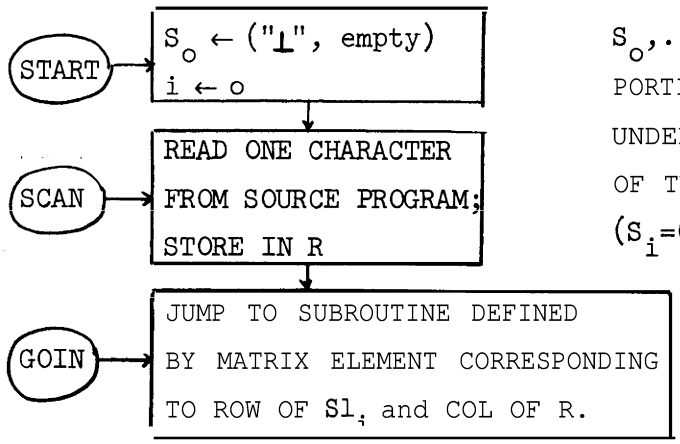
or

← BOTTOM	STACK	TOP
...+	"IBT"	

This is equivalent to changing production (4.1) to

" IF"	→	IF
" IBT"	→	"IF" <BE> THEN
"IBTEE"	→	"IBT" <EXPR> ELSE
<COND>	→	"IBTEE" <EXPR>

The productions are then all of length one, two, or three. One row of the matrix is allotted to each new symbol in quotes. One column is allotted to each possible incoming terminal symbol. A stack element S_i consists of two parts $S1_i$ and $S2_i$. The first is one of the quoted symbols introduced by the generator (a kind of operator), the other is empty or contains a nonterminal symbol (an operand). The basic recognizer is



$S_o, \ldots S_i$ IS A STACK HOLDING A PORTION OF THE SENTENTIAL FORM UNDER ANALYSIS. S_i CONSISTS OF TWO PARTS $S1_i$ and $S2_i$. $(S_i=(S1_i, S2_i))$

The matrix and subroutines produced by the generator for the grammar of Fig. 1 are

	⊥	+	*	(	)	I
"⊥"	1	4	5	6		8
"E+"	2	2	5	6	2	8
"T*"	3	3	3	6	3	8
"("		4	5	6	7	8

1: IF $S2_i$ = E OR $S2_i$ = T OR $S2_i$ = P

THEN SUCCESS EXIT ELSE ERROR;

2: IF $S2_i$ = T OR $S2_i$ = P

THEN BEGIN i ← i - 1; $S2_i$ ← E; GOTO GOIN END ELSE ERROR;

3: IF $S2_i$ = P

THEN BEGIN i ← i - 1; $S2_i$ ← T; GOTO GOIN END ELSE ERROR;

4: IF $S2_i$ = E OR $S2_i$ = T OR $S2_i$ = P

THEN BEGIN i ← i + 1; S_i ← ("E+", empty); GOTO SCAN END ELSE

ERROR;

5: IF $S2_i$ = P OR $S2_i$ = T

THEN BEGIN i ← i + 1; S_i ← ("T*", empty); GOTO SCAN END ELSE

ERROR;

6: IF $S2_i$ = empty

THEN BEGIN i ← i + 1; S_i ← ("(", empty); GOTO SCAN END ELSE

ERROR;

7: IF $S2_i$ = E OR $S2_i$ = T OR $S2_i$ = P

THEN BEGIN i ← i - 1; $S2_i$ ← P; GOTO SCAN END ELSE ERROR;

8: IF $S2_i$ = empty

THEN BEGIN $S2_i$ ← P; GOTO SCAN END ELSE ERROR;

A matrix for ALGOL is about 60 x 40. The checks for ST_i = empty may be deleted by doubling the number of rows of the matrix (see [Grie 67a]). Some alterations are usually necessary once the recognizer is generated, but since semantics may be inserted at any step of the parse (in any of the subroutines 1-8 above), and not only when a right part is recognized, the system is perhaps more flexible than the previous three. The grammar does not have to be changed much, although it must be an operator grammar. The constructor itself has not been used to generate a compiler yet, but the generated recognizers resemble to a large degree recognizers built by hand using the same technique (see [Grie 65]).

This is perhaps the fastest technique. Switching tables are always used when speed is essential. Its drawbacks are the space used and the large number of subroutines needed to implement the technique.

5. Production Language (Floyd [Flo 61], Evans [Ev 64], Earley [Ear 65])

The production language introduced by Floyd and modified by Evans consists of a set of productions of the form

LO: $s_3\ S_2\ S_1 \mid \rightarrow S_2'\ S_1' \mid$ *Gl

A more natural name for this would be a reduction, since it is used to indicate how to reduce, or parse a string.

We start parsing a sentence by putting the first symbol ⊥ of the sentence on the stack. Then we sequence through the productions, comparing the top of the stack with the symbols $S_1, S_2, \ldots$ directly to the left of the first bar | . When a match is found, the matched symbols $S_1 S_2, \ldots$ in the stack are replaced by the symbols $S_1', S_2' \ldots$. (If no replacement is to be made the arrow → and symbols S_1', S_2' do not appear.) The symbol σ appearing as some S_i matches any symbol on the stack. Then, if "*" appears following the second | the next input symbol is scanned and pushed onto the stack. Finally we start comparing symbols of the stack again, beginning with the production labeled by the name appearing at the right of the production (Gl in this case). Any production may be labeled. Earley has written a generator which produces from a suitable grammar a recognizer written in production language.

The production language program generated from the grammar in Fig. 1 is given in Fig. 9.

```
PROGRAMO:        ⊥|              |*EO
                 σ|              |ERROR EXIT
EO:TO:PO:        (|              |*EO
                 I|→         P|*P1
                 σ|              |ERROR EXIT
E1:              ⊥E⊥|            |SUCCESS EXIT
                 ((I))|→         P|*P1
                 E+|             |*TO
                 σ|              |ERROR EXIT
T1:              T*|             |*PO
                 E+Tσ|→          Eσ|E1
                 Tσ|→            Eσ|E1
                 σ|              |ERROR EXIT
P1:              T*Pσ|→          Tσ|T1
                 Pσ|→            Tσ|T1
                 σ|→             |ERROR EXIT
```

Fig. 9 Production language recognizer

Semantics are introduced onee the productions have been generated by inserting "actions" of the form EXEC i , where i is the number of some semantic subroutine, directly after the second bar | in any line of a production.

This production language is the basis for a working ALGOL compiler [EvA 64] and forms a significant part of FSL, a language for writing compilers (see section II.D1). A variation of the production language is also used in TGS (cf. section II.D2). Once one has some practice, it is quite a natural, flexible language to program in. A programmer can learn to write compilers with it relatively easily. No compilers have yet been written using a mechanically constructed recognizer, but the MEC actions may be inserted in any production, so that in general few alterations will have to be made in the grammar. More context can be used by the recognizer, so that a grammar is more likely to be accepted by this constructor than the other four.

It is perhaps the least efficient of the recognizers discussed, since at each step, the stack must be compared with successive productions until a match is found. The productions, however, take up less space than the other recognizers and the efficiency can be improved by good programming when they are constructed by hand.

We would venture to say that this branch of Translator Writing Systems is fairly complete. One can devise only a finite number of really different-left-right recognizers for parsing sentences using limited context. Even the first four recognizers listed here differ only in the programming techniques used -- theoretically they are all [1,1] bounded context in the terminology of section III.B1.

The operator precedence technique is most well-known. It often is used to recognize portions of a language, most frequently arithmetic and Boolean expressions, as is done in the IBM 360 FORTRAN compiler.

See [Ar 66, Grie 65] for documentation of other compilers using this technique. [Gall 67] also mentions it. The transition matrix technique has been used to write several ALGOL compilers [Grie 65, Sam 60] as well as NELIAC compilers, under the name CO-NO table [Hals 62, Mas 60]. Both of the above techniques have undoubtedly been used in many other compilers. The production language is used in an ALGOL compiler [EVA 64], but is also a significant part of two compiler-compilers [Feld 66, Mond 67] in which a number of other compilers have been written [Rov 67, It 66]. Two other compiler-compiler projects use this language [Fie 67, Grie 67b], while independent variations of it have been used by '[Che 65] and others. The precedence and extended precedence techniques have been used mainly by their authors, Wirth [Wir 66a, Wir 66b] and McKeeman [McKee 66]. There are further discussions of syntax techniques in several other sections.

For the theoretically inclined reader, section III.B1 contains discussions on more general, powerful and complicated left-right recognizers, as well as some basic references on the theory of formal languages.

II.B. Syntax Directed Symbol Processors

The programs discussed in this section are not properly called compiler-compilers, although each has been used to write compilers. Their common treatment of compiler-writing as a symbol manipulation task makes each of these programs both more than and less than a TWS. Since such systems are so general, they have been used heavily in the various non-translator tasks described in Section III.A. In fact, the discussion of AED [Ross 66] will be deferred to that section, because its goals have been more general from the outset.

1. TMG (McClure [McCl 65])

The TMG system was developed at Texas Instruments as a tool for writing simple one-pass compilers. The syntax technique is a simple top-down scan with backup. However, the embedding of semantic rules enables one to write a more efficient recognizer than would be possible with pure syntax.

The basic TMG statement form is:

<label> : <actions> / <identifier> **/ <identifier> .

The first <identifier? names the statement to be executed if the <actions> fail and the second <identifier> names the statement to be executed on success. The <action& can be: intermediate goals for the syntax recognizer, string computations on the input, or built-in statements. These <actions> are all to be performed by the translator; output of code is treated by a different construct to be discussed below. There is a character-based symbol table which is built from

input strings using the primitives MARKS and INSTALL. Consider the following example.

```
INTEGER:  ZERO* MARKS DIGIT DIGIT* INSTALL
```

The action ZERO* scans all leading zeros, then MARKS notes the current value of the input-string pointer. The actions DIGIT DIGIT* scan all characters in the class <digit>. The execution of INSTALL causes the string starting at the pointer of MARKS to be entered into the symbol table and a reference to it entered in the intermediate tree. The only other information allowed in the table is a set of declared FLAGS (Boolean variables).

The built-in routines include conditional arithmetic expressions, number conversions and a few input-output functions. There are also some system cells such as J, the input pointer, and SYMNRM, the length of the last string entered. Output is also character-oriented, as the following example will show:

```
LABELFIELD: LABEL = $(P1 / BSS / 0 // $ )
```

This statement would be used to detect the label in some language. The "=" signals an output routine which is bounded by "$(" and "$)". The body of the output statement will form one line of assembly code

```
label       BSS     0     .
```

The symbol "P1" is a command to evaluate the first construct to the left of the "=", presumably the symbolic name of the label. The "/ "

says insert a tab and "BSS" and "0" represent themselves. Finally, the "//" places a carriage return in the output.

The TMG effort was a pilot project and its clumsy syntax would be easy to fix. It has been used to write a number of compilers and a related system TROL has been used by Knuth for teaching compiler-writing. The EPL (Early PL/I) used in MULTICS was written as a two-pass system, using two sets of TMG definitions, to get better code. The TMG system does not seem to be as coherent as some to be considered below, and would benefit from another iteration.

2. GARGOYLE (Garwick [Gar 64])

The GARGOYLE system was developed by a Norseman and is not very well known in Vinland. It is also quite similar to TMG and so it will not be covered in as much detail.

The syntax processor is, once again, basically a top-down recognizer with the ability to direct the search. The descriptive language form is a five-field line, essentially

<label> : <action> ; <next> ; <link> ; <else>

The sequencing rule is more complicated than TMG with ERROR and EXIT being special cases and three successor fields to consider.

The <action& combine tests and output statements in an ALGOL-like syntax more pleasant than that of TMG. For example, a line in the routine COMPILE is:

if U = '+' then f ← 'FAD' ; INSERT

where INSERT (in the <next> field) is an output routine with 'f' as a parameter. In this case, 'U' is a temporary variable previously filled in by character tests on the input string.

There are a number of auxiliary features mentioned, but it is not always clear which ones are built in. The whole paper is somewhat tentative, suggesting that Garwick's intent was to present a schema for a TWS rather than a particular system. We have no information on implementations, uses or extensions of GARGOYLE.

5. COGENT (Reynolds [Rey 65])

The COGENT system was designed at Argonne National Laboratory by John Reynolds and implemented on a CDC 3600. A program written in COGENT has two parts: the syntax and a set of processing routines called generators.

The syntax is given by a synthetic grammar. Syntactic analysis proceeds by producing list structure to represent the syntax tree. For example, use of the production

<TERM> ::= <TERM> + <FACTOR>

would produce a list element <TERM> with pointers to the subexpressions <TERM> and <FACTOR>. Alternatively, one can preceed a production by action labels - names of generators which are capable of conditional analysis of list structures and of (recursively) calling other generators:

PROCESSTERM / <TERM> ::= <TERM> + <FACTOR>.

Instead of a list element <TERM> being created, the generator PROCESSTERM is called with the sublists <TERM> and <FACTOR> as arguments. The output of PROCESSTERM is then placed in the parse tree.

Certain kinds of local ambiguity are allowed in COGENT. The object syntax processor goes into ambiguity mode, switching back and forth between possible parses each time a parse requires a new character. No generator calls are made until the ambiguity is resolved.

The generator language is based on list-processing operations and the mechanism of failure. List elements may have varying numbers of pointers to other elements. The types of list elements include numbers (fixed or floating), generator entry pointers, dummy elements (corresponding to LISP's NIL), identifier elements, and parameter elements. Fixed point numbers may be of any magnitude and take up sufficient words to represent that magnitude. This feature facilitates symbolic mathematics applications of COGENT.

In addition to the conventional assignment statements, generators may use synthetic and analytic assignment statements to describe the synthesis and analysis of list structures. A synthetic assignment statement has the form

<identifier> / = <template , <expression list>

where a <template is essentially a production in parentheses. For example, the execution of the synthetic assignment statement

```
Z / = (TERM / FACTOR * FACTOR) , X, Y
```

where X had the value (FACTOR/ABE) and Y the value (FACTOR/BED), would assign to Z a copy of (TERM / ABE * BED).

Similarly, analytic assignment statements of the form

```
<test expression> = / <template , <identifier list>
```

are used to decompose an expression. The <test expression> is matched against the template. If they match, the value corresponding to the i^{th} parameter (nonterminal) of the template is assigned to the i^{th} <identifier> of the <identifier list>. Thus, if Z has the value (TERM / ABE * BED) , then the statement

```
Z = / (TERM / FACTOR * FACTOR) , X, Y
```

will give x the value (FACTOR / ABE) and Y the value (FACTOR / BED).

If <test expression> and <template do not match, the analytic assignment statement fails. Failure is the method of branching in COGENT. If no conditional statement includes the action that fails, the entire generator fails. Thus failure proceeds up the chain of generator calls until a conditional statement is encountered.

In addition to the above, the following features of COGENT require mention: ID-tables, scanners and internal variables. The action label $IDENT, n/ specifies that the result of that production (which must be a character string), should be placed in identifier table n. If it is already there, a pointer to the old

copy will be returned, i.e. all identifiers in any given table have unique character strings.

Generator entry pointers can be passed as arguments to generators. This is useful, for example, for producing output for cards, printing, etc. One generator, called a scanner, could reduce a list structure to a character string and pass the characters one at a time to an output routine through a formal parameter. There are also several internal variables which may be set or tested by primitive generators and used by various built-in routines. For example STANDSCN, the standard scanner, calls on the routine indicated by an internal variable to convert negative or floating point numbers.

COGENT is admittedly experimental and has several shortcomings: the structure of the language for generators is not as neat as Algol has shown languages can be, one syntax error in the input is fatal, and list processing should be generalized to include arbitrary plex-creation, rather than just plexes based on the syntax. COGENT has been applied to a number of problems in symbolic mathematics. Reynolds has suspended work on COGENT pending the development of a better theory of data structures which he, among others, is working to develop.

4. The META Systems (Schorre [Schor 64] et al.)

The early history of Meta compilers is closely tied to the history of SIGPLAN Working Group 1 on syntax-directed compilers. The latest inventory listed twenty-five different Meta compiler systems on ten different computers. The proliferation of these compilers is due in

part to the fact that they are not only able to compile a metalanguage but can be expressed in their own language and thus compile themselves. Almost all of the systems have been used to implement translators for other languages as well.

Although the original work was diversified, the current systems are generally based on a model known as Meta-II, developed by Schorre. Within this model, the parsing and translation processes for a language are all stated in a set of BNF-like rules. These rules become recursive recognizers with embedded code generators when the language specifications are implemented.

The rules do not allow left recursion; but use instead the (prefix) iteration operator "$". Alternation (the bar in BNF) is indicated by a slash, and parentheses are used for grouping in a normal fashion. The following is a typical rule in Schorre's Meta-II language:

```
SUM = TERM $( '+' TERM .OUT('ADD')/
              '-' TERM .OUT('SUB') );
```

The rule defines a procedure for recognizing a sum in an algebraic language. The word "SUM" -followed-by "=" defines the name of the rule, while the right part of the rule is both an algorithm for testing an input stream for the occurrence of a sum as well as a code generator in case the sum is found. The above rule contains examples of the three basic entities used in most Meta compilers. The mention of the name of another rule, in this case "TERM," causes a recursive call on that'recognizer to be invoked. The occurrence of a literal string '+'

signifies that a test is to be made against the input stream for a plus sign; most Meta systems have built-in recognizers for identifiers and numbers as well as literals. In the ".OUT" construct, we see the embedding of code generation.

The recursive nature of the rules and the method of handling generated labels may be seen in the following example:

```
UNION = INTER ('OR' .OUT('BT' *1) UNION .LABEL *1 / . EMPTY);
INTER = PRIMARY ('AND' .OUT('BF' *1) INTER .LABEL *1 / .EMPTY);
PRIMARY= .ID .OUT('LD' *) / '(' UNION ')';
```

For the input stream "(A OR B) AND (C OR' D)", the following code would be produced, where LD, BT, BF are mnemonics for Load, Branch True, and Branch False respectively:

```
            LD      A
            BT      L1
            LD      B
L1
            BF      L2
            LD      C
            BT      L3
            LD      D
L3
L2
```

The first mention of a *1 within a rule causes both the generation of a label and the output of that label. Subsequent references within the same rule output the same label. That is, when a rule is entered, new labels are generated. These labels only exist while the rule is active. If a call is made to another rule, the labels are pushed onto a stack. Upon return from the called rule, the previous labels are restored. The "*" causes the last item recognized by the primitive .ID to be added to the output. .EMPTY is a primitive which has no effect on the input or output but is always satisfied or true.

Meta 3 was an attempt to extend the basic Meta-II concept so that ALGOL 60 could be compiled for a 7090. It added some ability for semantic tests and register manipulation, but the additions never proved adequate. Meta 5 was the first Meta compiler that allowed backup of the input stream. It also added extensive string push-down stacks, attribute assignment and testing, and output formatting features. An indication of the flexibility of Meta 5 is the fact that it is capable of translating JOVIAL to PL/1. The LOT system ([Kir 66]), another extension of Meta-II, added syntax constructs which gave the programmer complete control of almost all system parameters and flags. Normally, the setting of these parameters is done by control cards, but embedding it in the metalanguage proved extremely useful in the development of debugging aids. The LOT system was also used to gather statistics on the efficiency of top-bottom syntax analysis.

There is currently a very active interest in the development of Meta systems. The tendency in the newer systems is to build parsing

trees and then, with another special-purpose language, test and collapse the trees, producing output as a side effect (cf. Section II.D2). The slowness and inefficiency of Meta compilers is recognized by their authors, but the ease of implementation, the bootstrapping capabilities, and the large class of languages they can handle are used to justify the work that has gone into their development.

References for II.B:

Ab 66, Gar 64, Kirk 65, McCl 65, Met 64, Rey 65, Sch 64, Schor 64,

II.C. Meta-Assemblers and Extendible Compilers

These forms of TWS are similar in that they both attempt to extend the macro concept to higher level programming languages. The basic idea in a macro processor is the systematic replacement of certain symbols with their associated pieces of text. Although almost all modern assemblers have sophisticated macro features, the best descriptions of the idea are in the general papers by Strachey [Str 65] and Mooers and Deutsch [Moo 65]. The meta-assembler and the extendible-compiler are based on two different conceptions of how to extend macros to high level languages. The meta-assembler approach considers the compiler to be special case of the assembler, while the extendible compiler approach is to add text replacement features to standard compilers.

1. General Discussion and METAPLAN (Ferguson [Fer 66])

The article by Ferguson is taken from the San Dimas conference and contains a good introduction to meta-assemblers. The basic ideas arose from observing that all assemblers have many features in common. By building procedures for handling such things as symbol tables, location counters and macros, one could speed up the writing of particular assemblers. To construct an assembler for a particular machine one would specify word size, number representations and the like. Output for each machine would be programmed using format statements and could easily include relocation or symbolic debugging information.

While such a system seems feasible and quite useful, it is not obvious how one would extend it to a TWS.

The use of a meta-assembler as a TWS is based on the previously mentioned assumption that the compiler is a special case of the macro assembler. Discussions of this assumption sound like a reincarnation of the macro vs. high level language debate. The macro assembler side is on the defensive, is outnumbered and therefore has been the most vehement in argument. The whole situation is further complicated by a lack of agreement on what an assembler is (cf. discussion following this paper [Fer66]). An example will suffice for our purposes.

Ferguson describes how a **meta-assembler** would handle the **compiler-**like statement:

```
IF F(A) PLUS 5 EQ G(B) GOT0 L .
```

He would have IF, PLUS, EQ, and GOT0 be defined as (prefix) operators using a scheme called many-many macros. The many-many macro has features for using and updating state information during text replacement. This seems to be considered an outstanding contribution to macro techniques and is certainly a prerequisite for reasonable code selection. The many-many macro is flexible enough to implement any known compiler; the real question is whether many-many macros are a good way of doing it. The answer to this depends on the mechanisms for recording and using state information and these were not discussed in the paper.

2. PLASMA (Graham and Ingerman [GraM 65])

The meta-assembler effort of Graham and Ingerman concentrates mainly on the problems of substitution and binding. They are much less concerned with syntax than Halpern (next discussion), because they assume a syntax-directed front end (presumably [Ing 66]) for a compiler written in their system.

The basic input to their meta-assembler is a "line" which is a list of lists. The first list is a generalized label consisting of a symbol, the number of higher levels at which it is active, and the number of lower levels at which it is active. The second list contains the operation and the third contains the operands. The input is converted into a tree and substitutions are made on the basis of the tree structure. By allowing substitutions by symbol or numeric value, they combine the text replacement with assembly functions.

The authors are continuing their work at RCA, Cherry Hill, and will presumably report on it again. Their current efforts involve even more elaborate substitution processes. They have not, as yet, put forth specific suggestions on how their system might be used as the basis for a compiler.

3. XPOP (Halpern [Hal 64])

Halpern is the most sanguine and vocal of the meta-assembler proponents. His work on meta-assemblers is related to his controversial stands on natural language programs by his statement that XPOP will allow one to implement something "closely approaching"

natural language. One should try to separate his work, which is reasonable, from the tub-thumping which mar his appearances in public or in print.

The XPOP system follows fairly well the general meta-assembler description by Ferguson. The basic input format is, once again, a label followed by an operator and one or more operands. Halpern is very interested in input forms and has three basic ways of altering the syntax of the source language. The- first way is to change the order of parameters by declaring a macro with the new parameter ordering which expands to the original operator. The second feature is the ability to declare new separators and terminators at any point in the text. The most unusual feature is the facility for adding noise words which are ignored, as well as keywords which mark the next symbol as a parameter.

To handle the problems of generating output, XPOP has several embellishments of the macro concept. It is possible to defer the assembly of code sections; the sections awaiting a particular label can accumulate in FIFO or LIFO fashion. There is one illustration of how this feature is used to implement the DO statements in FORTRAN. There is also mention of many-many macros and of assembly time execution facilities. Once again, there is not enough information presented to allow one to judge their suitability for translator writing. The XPOP system has a large variety, of trace and debugging aids which should add significantly to its usefulness.

More recently, Halpern has produced an elaborate defense of XPOP-like systems. He suggests that the <operator> <operand-string> notation of macro systems is the canonical syntax of programming languages as opposed to natural or mathematical languages. He further separates the study of programming languages into three parts: Functional (macros), Notational (change punctuation commands), and Modal (assembly-time executions). Halpern's paper can be taken as the philosophical statement of the meta-assembler position on TWS and compared with other general descriptions of the problem.

4. Extendible Compilers - Basic Concepts.

Many attempts (starting with McIlroy [McIl 60]) have been made to embed macro features in compiler systems. One approach was to retain the macro syntax form, but add a number of built-in features which are compiler-like. The SET system [Ben 64a] included a skeleton compiler with input-output, symbol manipulation, table handling, and list processing features. These built-in routines were combined with translation-time operations (Action Operators) in the attempt to build a TWS. A more successful approach has been to use the structured syntax of high-level languages as a basis for extension.

Many existing compilers incorporate simple forms of macro expansion, the first probably being JOVIAL [Shaw 63]. The most primitive form is pure text replacement without parameter substitution. For example, in B5500 ALGOL one could define a macro with the statement:

```
DEFINE LOOP1 = FOR I ← 1 STEP 1 UNTIL #
```

and later form statements like

```
LOOP1 N DO AII] ← 0
```

which would be expanded into

```
FOR It 1 STEP 1 UNTIL N DO A[I] ← 0 .
```

The next step is to allow a macro definition with parameters. This facility has been included in the AED-0 compiler [Ross 66], among others. In AED-0 one might define a macro with the statement:

```
DEFINE MACRO LOOP (P1,P2) TOBE
FOR P1 ← 1 STEP 1 UNTIL P2 DO ENDMACRO
```

In this case, one could get the same result as above with the short statement

```
LOOP(I,N) AII] ← 0 .
```

These two simple macro forms would form a useful addition to any high level language and one might imagine developing mechanisms which parallel more sophisticated macro techniques. Although AED-0 does permit arbitrary strings as parameters and nested definitions, features like conditional assembly do not seem to have been used in high level languages. One reason for this is that compilers normally depend heavily on the structure of the text; the next two sections describe the complexities that arise in trying to extend compilers with macro techniques.

5. Definitional Extensions (Cheatham [Che 66])

The definitional extension of high level languages is the latest attack on the TWS problem by the Computer Associates group. This has been the most active and productive group in the TWS area and has developed a world-view which should be understood in reading their work. We will discuss the mainstream of their activity in Section II.D2, only a brief introduction will be given here.

Cheatham defines compiling as a six-step process involving: lexical analysis, syntactic analysis, interpretation of the parse, optimization, code selection, and output. The principal driving force behind their work has been run-time efficiency, although other considerations have played an important role from time to time. The current TWS efforts of Computer Associates use a single language TRANDIR for all the steps of compilation. TRANDIR consists essentially of an algebraic section, a pattern matching section (cf. Section II.A5) and a number of built-in functions. The language is procedural and, to date, has been used only by experienced compiler-writers.

The paper under discussion shows signs of having been hastily written and contains references to several internal documents in preparation. This is clearly an early attempt along these lines and will be expanded and clarified in subsequent papers. The extensions to compilers mentioned here fall into two broad categories: a descriptive meta-language L_D and a series of macro facilities.

The descriptive meta-language L_D is meant to be translated into TRANDIR procedures, presumably by a (meta-meta) processor. The

translation of the language L_D is based on a grammar inversion technique combining notions of Wirth and Early (cf. Section II.A). To allow for more powerful languages, one can append predicates (e.g. type checking) and even arbitrary computations to the declarative syntax. Finally, there are rules for outputting intermediate code attached to the syntax rules. The declarative language has not been implemented, but Cheatham claims that it has proved useful for the initial formulation of TRANDIR compilers. While this is probably true, one would expect that the translation to procedural form is not, at present, a mechanical process. Further, the sophistication required of an L_D user does not seem appreciably less than that required by TRANDIR.

The extensions to languages using macro techniques fall into three basic categories: text, syntactic, and computational macros. Text macros are assumed to be well understood and would presumably be similar to those described above. It is in treating syntactic macros that Cheatham begins to face seriously the problems of adapting macro concepts to compilers.

The basic features of syntactic macros are free format and type specifications for parameters: An example would be

```
LET N BE INTEGER
MACRO MATRIX (N) MEANS 'ARRAY[1:N, 1:N]' .
```

The advantage of free format over the conventional <operator> , <operand list> format are obvious; the specification of parameters allows conditional assembly and better error detection. The call of a

syntactic macro would be set off by a special delimiter (e.g. %) and would have to have a detectable termination. These problems can be avoided by adding the macro form directly to the syntax tables of the translator. The corresponding declaration would be:

```
LET N BE INTEGER
SMACRO MATRIX (N) AS ATTRIBUTE MEANS  'ARRAY[1:N,1:N]'
```

where ATTRIBUTE is a syntactic type in the definition of the underlying language . Neither of these schemes presents an implementation problem in TRANGEN (cf. Section III.D2), but either of them could have drastic results if misused.

In discussing syntactic macros, Cheatham touches upon the problem of adding 'semantics' to the macro definition. This is the analogue of the many-many macros and the assembly-time actions used in meta-assemblers. Cheatham's conclusion that this approach is not feasible should be compared with the meta-assembler approach which has put most of its eggs in this basket. His solution is to provide a number of primitive operations (e.g. table expansions) and to point out the existence of a complete meta-language behind the extendible language.

The third type of macro extension is called the computational macro. With this technique the substitutions are made in the intermediate code resulting from a declared macro. This requires that the macro body be restricted to constructs for which the intermediate code can be compiled (with formal parameters) independent of context. If this condition can be met, the computational macro is a useful and efficient tool. A simple computational macro might be the following mapping

function for a 4 x 4 upper left triangular matrix M.

```
TAKE I,J AS INTEGER
MAP M(I,J) = (11-I) * 1/2 + J-6 ;
```

where TAKE and MAP are declarators in the language, Since this code is for array accessing, it should not be inserted into the source text and the computational macro form is most appropriate. As Cheatham points out, computational macros have long been used by compiler writers to produce accessing code for arrays. The paper includes several examples of accessing functions, a subject that will reappear in the discussion of Perlis and Galler paper in the next section. The important point is that Cheatham has provided a procedural way of describing access functions while Perlis and Galler try to generate the code from non-procedural descriptions.

6. ALGOL C (Galler and Perlis [Gall 67])

This is a very long, difficult and important paper by two of the outstanding workers in the field of programming languages. Although there are many significant aspects of the paper, we will discuss here only those dealing with extendible compilers. Other topics will be treated in Section III.B as significant first steps in new research areas.

The basic idea is, once again, to add macro-like facilities to a high level language. For this purpose they define an extension of ALGOL called ALGOL C which is meant to be well suited to extension. Any extension of ALGOL C is called an ALGOL D and a program in any of these can be mechanically reduced to an equivalent ALGOL C program.

The extensions are accomplished through constructs rather like Cheatham's SMACROS. Because they want to do the macro processing in very sophisticated ways, Perlis and Galler allow redefinitions only in a few fixed syntactic categories. The augmented language ALGOL C contains many features for handling arrays as well as those more directly concerned with extendibility. Among the latter are operators for conversion between location and value:

(a) A unary operator with integer result:

loc of x

where x is a <procedure identifier> , <variable, or <array identifier>. loc of x is essentially the address of the word(s) containing the value of x.

(b) Two binary operators whose left operand is a <type> or is missing, implying real, and whose right operand is an integer expression, representing the "address" of some <procedure , <variable> or <array>:

<type> vc of x

<type> pic of x .

These represent "value contents of" and "procedure identifier contents of", respectively. Thus

real vc of (loc of x) = x

if x is a variable of <type> real.

(c) The notions of location and value are extended to <procedure> s with the help of an application operator ⊕. The precise syntax changes are bound up with the array conventions, but revised definitions of <primary> and <function designator> should convey the intent.

<primary> ::= <unsigned number> | <variable | <function designator> | (<arithmetic expression>) | loc of <procedure identifier> | <type> vc of <arithmetic expression>

<function designator> ::= <procedure identifier> ⊕ <actual parameter part> | (pic of <arithmetic expression>) ⊕ <actual parameter part>

Thus, one is able to manipulate the names of procedures in much the same way as address variables and could, for example, have procedure arrays. These additions to ALGOL to form ALGOL C constitute only a small part of the extra mechanism; most of it is embedded in the various forms of ALGOL D.

All ALGOL D languages will have fairly much the same syntax. The common syntax for all ALGOL D's is the same as ALGOL C except for the replacement of <type> , <arithmetic expression> , <Boolean expression> and <assignment statement> with a set of rules which enable the definition of special forms for these syntactic types. The introduction of new definitions occurs as a series of declarations at the beginning

of a block. The detailed description of this process is quite complicated and we will present only an overview followed by an example.

The basic intention is to allow the definition of new data types and their associated operators. The problem of finding symbols for these operators is solved by assuming a large alphabet of boldface characters. By assuming an operator precedence grammar (cf. Section II.A1), one can define the precedence of new operators in relation to operators of known precedence as in MAD [Ar 66]. The remaining problems with operators involve data types and will be deferred for a few sentences.

New data types are defined in terms of ALGOL C or previously defined types by a means statement. This may include formal parameters which, if present, play a crucial role in all further processing, e.g. matrix(u,v) means array [1:u, 1:v].

One then combines operator and type information in a set of context statements. A context statement describes, for an operator, the data types of its operands and its result. It also contains a <string> which is (eventually) reducible to the appropriate ALGOL C text. The following example of [pseudo) LISP definitions should help clarify these notions.

List Definition Set:

The following set of definitions is based on the LISP [McCar 62b] primitives. The basic LISP predicates "atom" and "eq" are assumed to

have been defined as Boolean procedures:

```
Boolean procedure atom(x); list x;
     atom := cdr x = 0;
Boolean procedure eq(x,y); list x,y;
     eq := car x = car y ∧ atom(x) ∧ atom(y);
```

'NIL' in LISP is represented here by 0. The following definitions are used to organize lists as structures of names.

```
(1)  list means integer array [1:2];
(2)  cons ≒ *
(3)  car ⊳ cons;
(4)  cdr ≒ car;
(5)  of ⊲ cons;
(6)  list a cons list b ≡ list 'list(a,b)';
(7)  car list a ≡ list 'a[1]';
(8)  cdr list a ≡ list 'a[2]';
(9)  loc of list a ≡ integer;
(10) integer a := list b ≡ integer 'a := loc of b';
```

Statement (1) defines the new data type list as a two-element integer array. Statements (2) through (5) state the relative precedence of the four LISP operators. Statements (6) through (9) define expressions; e.g. (7) defines the car of a list 'a' to be the first element of the modeling array. Statement (10) defines the assignment statement for assigning a list to an integer variable.

```
(11) op(F) f of list x ≡ list 'E(list (loc of F,0),x)';
(12) op(F) f of op(G) g ≡
        list 'list (loc of list (loc of F,0), loc of G)';
(13) list y of op(F) f ≡ list 'list (y, loc of F)';
(14) list y of list x ≡ list 'E(y,x)';
```

Context definitions (11) through (14) provide an efficient rule for sequencing through a composition of operations on lists, each one of which operates only on atoms to produce atoms or even lists. The procedure E is organized so that as each atom of data is encountered the remaining operators in the composition are applied to it. Thus the lists are not totally decomposed and composed for each successive operator. In a <declaration> such as op (H) h, the <actual parameter> H represents the <procedure to be used to apply h to a list. The lists are assumed to be nonrecursive, in the sense that no list is a sublist of itself.

The block containing these list definitions must also contain the procedure E:

```
list procedure E(f,x); list f,x;
E := if atom(x) then (if atom(f) then (list pic of car f) (x)
     else E(car f, (list pic of cdr f))) else E(f, car x) cons
     E(f,cdr x);
```

An example of a LISP program is:

```
begin op(F)f; op(G)g; integer c; list a, b, d, h, k;
integer procedure subst (x, y, z); list x, y, z;
     subst:= if atom(z) then (if eq(z,y) then x else z) else
     subst(x,y, car z) cons subst (x,y, cdr z);
list procedure F(x); list x; F := subst(a,k,x);
list procedure G(x); list x; G := subst(d,h,x);
c := (f of g) of b end;
```

The example above does justice neither to LISP nor to the Galler-Perlis system. The full design of their system has ALGOL C defined by a similar definition set in the outermost block. In each subsequent block the translator builds a type table and a context table using the local definition set. The actual processing of local ALGOL D text is quite involved. This arises from the facts that contexts are recursive and that ALGOL C text can be interspersed with locally defined text. The discussion in the paper is further complicated by a desire to optimize the computation in addition to producing ALGOL C code.

We have deliberately, if not successfully, distorted the intent of Galler and Perlis' paper. They were also concerned with arrays, and more particularly with saving space in matrix calculations. It would have been preferable on all sides for them to have made the separation of issues themselves. As we have mentioned, the paper contains important discussions of subjects other than extendible compilers. Its contribution to our topic is more theoretical than practical. They

have shown that very sophisticated macro-processing is possible and can lead to substantive changes in an algebraic language. One would guess, however, that inefficiency at translation time and sensitivity to programming errors would seriously restrict its practicality. There is, in addition, a general question of how often one would want to change a high-level language; this will be taken up again in Section III.C.

References for II.C.

Benn 64a, 64b, Brook 60b, Che 64a,66, Fer 66, Gal 67, GraM 65, McIl 60, Mea 63, Moo 65, Str 65

II.D. Compiler-Compilers

The distinguishing characteristic of this set of TWS is the attempt to automate many of the post-syntactic aspects of translator writing. Such systems might better be called compiler-writing-systems because they include significant programs which are resident at translation and execution time, as well as meta-language processors. The programs in this section are much more complex than most of those discussed previously; none has ever been implemented by someone not in contact with a previous effort of the same type. The following excerpt from a paper on FSL outlines basic philosophy and should serve as an adequate introduction to our discussion of compiler-compilers. The other compiler-compiler projects discussed in this section have similar philosophies; we will point out the differences in the appropriate sections.

When a compiler for some language, L, is required, the following steps are taken. First the formal syntax of L, expressed in a syntactic meta-language, is fed into the syntax loader. This program builds tables which will control the recognition and parsing of programs in the language L. Then the semantics of L, written in a semantic meta-language, is fed into the Semantic Loader. This program builds another table, this one containing a description of the meaning of statements in L. Finally, everything to the left of the double line in Figure 1 is discarded, leaving a compiler for L.

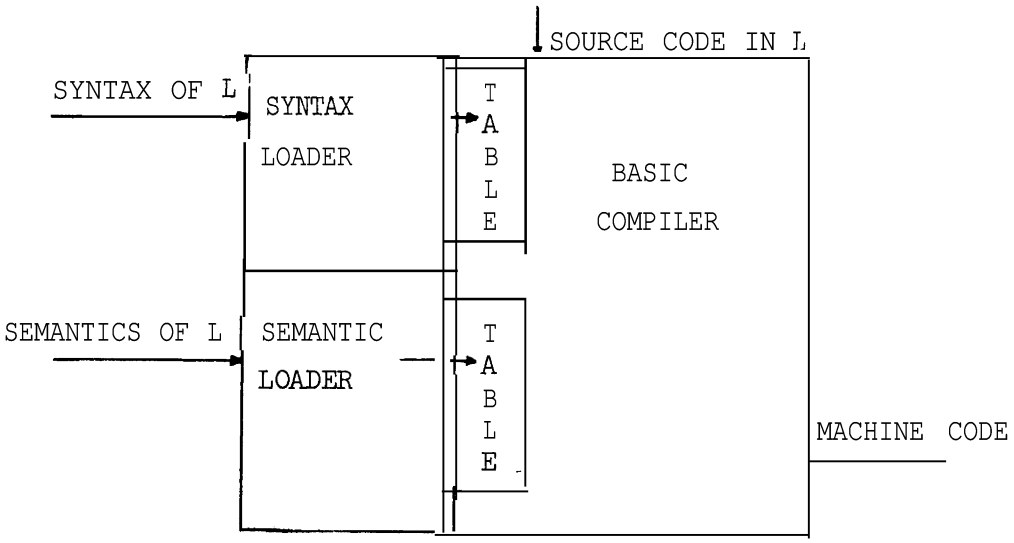


Fig. 10. A compiler-compiler

The resulting compiler is a table-driven translator based on a recognizer using a single pushdown stack. Each element in this stack consists of two machine words -- one for a syntactic construct and the other holding the semantics of that construct. When a particular construct is recognized, its semantic word and the semantic table determine what actions the translator will take. The Basic Compiler includes input-output, code generation routines and other facilities used by all translators.

1. FSL and its descendents (Feldman [Feld 66])

The problem faced in the original FSL effort was the development of a language for describing the post-syntactic (semantic) processing. An adequate semantic meta-language should permit the description of the source language to be as natural as possible. It should be readable so that other people can understand the meaning of the source

language being defined. It should allow a description which is sufficiently precise and complete to enable efficient automatic compilation. Finally, the meta-language should not depend on the characteristics of a particular computer.

Since there are satisfactory ways of representing syntax, the formalization of semantics should make possible a complete formal description of computer languages. With a complete formal description available, one could organize a compiler as shown in Figure 10.

The syntax meta-language used in FSL is very close to the Floyd [Flo 61] and Evans [EvA 64] production language and is discussed in Section II.A5. A statement in this syntax language may include a command "EXEC n" which is a call on the semantic statement labeled n. The only other interaction between syntax and semantics is the pairing of syntactic and semantic descriptions in the pushdown stack. This clean division of syntax and semantics has some advantages, but has proved to be a great handicap in implementing certain languages.

The semantic meta-language, called the Formal Semantic Language (whence FSL), was the main focus of effort and will be discussed in some detail here. The overriding consideration in FSL was machine independence as opposed to object code optimization in the TRANGEN effort discussed below. The plan was to have the meta-language be machine independent, with the machine dependent aspects of translation handled by a large set of primitives imbedded in the basic compiler. Statements in the meta-language would be compiled into machine code made up largely of calls on primitive routines. Some examples should serve to illustrate this approach.

Suppose the syntax phase is processing a REAL declaration and calls semantic routine 1 with the identifier being declared in the second position of the stack (LEFT2).

```
1:  TO ← STORLOC; SET[TO,DOUBLE];
        ENTER [SYMB; LEFT2, TO, REAL, LEV];
    STORLOC ← STORLOC+2
```

'The current value of STORLOC (the storage pointer) is placed in a temporary and tagged with bits marking it a double-precision operand. Then a description of the variable is placed in the symbol table, SYMB. The entries for the variable are its name, the tagged address, the word REAL, and the current level. Finally, STORLOC is increased by two, allocating two cells to the double-precision variable.

When an identifier is scanned in an arithmetic statement, semantic routine 2 is called.

```
2:  IF CONST[LEFT1] THEN RIGHT1 ← LEFT1
        ELSE IF SYMB[LEFT1,,$,] = REAL
            THEN RIGHT1 ← SYMB[LEFT1,$,,]
            ELSE  FAULT1
```

In semantic routine 2, the predicate CONST is applied to the identifier (in LEFT1) to test if it is a constant. If so, the stack is adjusted and the routine terminates. If not, the identifier is a variable and must be looked up in the symbol table. The table-lookup is accomplished in FSL through a special table operand of the form

SYMB[LEFT1,,$,] .

This operand initiates a search of the table SYMB for an entry in the first row which equals the contents of LEFT1. Then the position of the $ is used to select the desired entry of the matched row.

In routine 2 the third entry (data type) of the matched row is selected and compared with the string construct REAL. If they are the same, the variable was declared to be REAL and all is well. In this case the second entry (tagged address) of the matched row in SYMB is assigned as the semantics of the real variable. If the variable is not of type REAL or is not in the table at all, the statement FAULT 1 will be executed. This causes the printing of an error message on the listing of the source language program being compiled.

Finally, suppose the syntax has recognized an addition which is to be compiled and calls semantic routine 3.

3: CODE(VALUE2 ← LEFT4 + LEFT2);

The code brackets 'CODE(' and ')' specify that the statement within them is to be compiled into object code, rather than executed during translation. This statement will produce a call on a code generating routine which uses the semantic descriptions in the second and fourth positions of the stack to compile an addition code-sequence. The result of an addition is itself an expression and the syntax is presumed to have put its symbol (E) into the second position of the stack. The assignment to VALUE2 will associate the semantics of the

result (e.g. DOUBLE, in accumulator) with the syntactic symbol. The FSL system allows almost all constructs to appear inside code brackets (to be done at execution time) or outside code brackets (to be done during translation).

The semantic meta-language, FSL, allows a compiler writer to declare and use a variety of data structures in building a translator. Besides the tables mentioned in the examples, there are stacks, masks, strings, and conventional cells. The language also includes other features such as chaining, addressing levels, and output statements which facilitate compiler writing. The Formula Algol compiler was largely written in FSL and the description [It 66] of that implementation provides a good study of the strengths and weaknesses of FSL.

The weaknesses of FSL can be characterized as the lack of several conveniences and a number of basic structural defects. The lack of conveniences such as index variables, assembly language embedding and debugging aids are due to its development as a thesis (hit and run) project and have been remedied in later systems. The structural defects result mainly from the attempt to preserve machine independence.

An FSL system is useful to the extent that the compiler-writer's needs are met by the facilities of the semantic meta-language. This, in turn, is possible only if there are suitable formalizations of the pertinent concepts. Thus all the research problems listed in Section III.C (e.g. data structures, paging, parallelism) are problems in any FSL system. Neither of the systems now running have good facilities for global code optimization or multipass compilers,

but these problems are being attacked by Gries [Grie 67b] at Stanford and the CABAL group [Fie 67] at Carnegie. There are, however, limits to the level of code optimization which can be achieved in a machine-independent way. There is a sense in which any FSL system is predestined to failure; techniques will always be used before they are sufficiently well understood to be formalized. Such a system can still be very helpful and the search for meta-language representations should lead to careful study of new techniques.

The only other FSL-like system completed to date is VITAL [Mond 67] at the Lincoln Laboratory. VITAL runs in a time-sharing environment and differs from FSL mainly in system features. These, along with a number of notational improvements, make VITAL much easier to use, but are of little theoretical interest. As an illustration we present the routines described above as they would appear in VITAL.

```
1:  ENTER[SYMB; LEFT2, (STORLOC|DOUBLE), REAL, LEVI;
        TALLY[STORLOC,2]
2:  IFNOT LEFT1 IS CONSTANT THEN
        IF SYMB[LEFT1, TYPE] = REAL THEN
            RIGHT1 ← SYMB[LEFT1, SEMANTICS]
        ELSE FAULT 1 ;
3: RIGHT2 ← CODE(LEFT4 + LEFT2)
```

There are also several substantive changes from FSL, including a conditional in the syntax language which depends on semantic information. The combined features of persistent storage and compile-time execution

facilitates the writing of incremental compilers, VITAL also allows the compiler-writer direct access to the accumulator marker and semantic words if he so chooses.

The FSL systems have undoubtedly been handicapped by being implemented on uncommon machines, the G-20 and the TX-2. To compensate for this there are now three separate implementations for the IBM 360 series in progress. The CABAL group at Carnegie [Fie 67] is designing a system for multipass compilers using a semantic language which is a minimal extension of ALGOL in the direction of FSL. The work under Gries at Stanford [Grie67b] will also be multipass-oriented, but will use a special purpose semantic language. The Lincoln Laboratory effort under J. Curry will probably be quite similar to VITAL. All of these projects may be considered attempts to combine the virtues of FSL with those of TGS, our next subject.

2. TGS (Cheatham et al. [Plas 66, Che 65])

One of the most productive groups in TWS research has been the small consulting company, Massachusetts Computer Associates (COMPASS). Although their TWS have undergone many changes, the basic world-view and goals of their effort have remained rather constant. The COMPASS work has been marked by careful attention to systems questions and to object-code optimization. Other aspects of their effort are discussed in Section II.C5 which deals with an extendible compiler scheme within TGS.

The first attack on the TWS problem at COMPASS was called CGS [War 64] and was quite different from their current work. Although

they have abandoned this approach, we will discuss it briefly here because it seems to be rediscovered periodically. The CGS system was based on a top-down recognizer which produced a syntax tree to be used in further analysis. The input to this phase was essentially BNF. The second phase was the generation of intermediate code using a tree-matching language called GSL. The actual code selection process was written in a third language, MDL. This effort was abandoned because trees were found to be slow to build and difficult to do pattern recognition upon,

The TGS systems differ from CGS, as well as the other systems described in this section, in the use of a single language for describing all phases of the compiler. This language, TRANDIR, is compiled into an interpretive code which is processed by the TRANGEN interpreter. If one combines the syntax and semantic loaders of Fig. 10, the FSL model applies quite well to TGS. In fact, there has been good communications between these two efforts and they have converged to a marked degree. The communication has not, however, been perfect; two concurrent implementations of TGS and FSL took place within a few hundred yards of each other without making contact.

The TRANDIR language contains a pattern-matching subset which is essentially the same as the syntax language used in FSL (cf. Section II.A5). The TGS version is more flexible in that it can be used on a variety of stacks and can match on properties other than identity of symbols. The pattern matching features can be used in various code optimization techniques as well as in syntax analysis.

The remaining features in TRANDIR language are quite similar to the semantic language in FSL. There is a "symbol description" (SD)

connected with each syntactic construct which is the analog of the "semantic word" in FSL. There are fairly elaborate facilities for declaring tables, stacks, masks, etc. for use by the translator. These various storage methods with the associated operators provide a very flexible means of recording and accessing the information needed for compiling efficient code. The FSL notion of code brackets is replaced in TGS by a series of symbol manipulation primitives to help the compiler writer produce output code. The operation of a TGS compiler can be best described by working through an example fairly completely.

The example will be taken from a compiler for a miniature algebraic language L_{to} described in [Plas 66]. The basic compilation technique chosen is to use a tabular intermediate code as is common in COMPASS compilers [Che 66]. A typical intermediate code translation of

$$Z \leftarrow X * Y$$

would be

```
0    TIMES   X   Y
0    STORE   Z   0
```

The intermediate code will be processed by a code selection phase which will produce the final output for later assembly.

Consider first the TGS statement:

```
...VAR AE // EMIT(STORE,COMP(1),COMP(0));
             EXCISE; TRY(ENDST).
```

The left part (up to the //) of this statement is a pattern of type <variable> <expression> which is compared with main stack (SYMLIST). If a match is attained the remainder (action part) of the statement is executed. The action EMIT produces a STORE intermediate instruction with the operands being the first and zeroth elements of the stack as matched. Since there is no resulting semantic description (SD), the action EXCISE is used to erase the two matched elements from'the stack. Finally, the action TRY(ENDST) directs TRANGEN to try to match the pattern labelled ENDST.

A somewhat more complicated routine would be used for recognizing a multiplication:

```
...VAL $* VAL // PHRASE(SYMRES(TIMES,COMP(2),COMP(0)));
        AESET:  SYNTYP (COMP(0)) = AE; TRY(AE1)
```

When one understands that "$*" denotes the terminal symbol "*", the left part of this statement should be clear. The action SYMRES is a call on a routine which performs an EMIT of the same parameters and also returns an SD as its value. This SD becomes a parameter to PHRASE which uses it to replace the matched portion of the stack. The action labelled AESET causes the syntactic type of the new top element to be assigned the value "AE". Finally, the statement TRY(AE1) leads to further expression processing.

These two TGS statements, if appearing in reverse order, would compile "Z ← X * Y" into intermediate language. In the real world, typical statements would involve table operations, string commands,

conditionals and other more complicated TRANDIR constructs. There are also fairly sophisticated <procedure> features which improve the readability as well as the writability, of translators.

In any event, the intermediate code will itself be processed by another set of TRANGEN routines called the code selectors. These are written in the same form as the syntax routines considered above. For example:

```
//   TIMES   INMEM  INMEM...
                    LOADMQ(XM+1).
```

This statement has a pattern involving a predicate INMEM (meaning in memory) on stack entries rather than symbols to match. (The delimiters "//" and " . . ." indicate that the pattern is to be matched against the intermediate code portion of the stack). The subroutine LOADMQ is called with a pointer to the second stack operand as parameter. This user-written routine will assemble a LOAD MQ command if necessary and will adjust the SD in the stack to reflect the fact that one operand is now in the MQ register. A similar routine will be used to compile the appropriate multiply sequence. The result will be in the accumulator and TRANGEN will eventually match the statement:

```
//   STORE  **   *INAC . . .
          IF SIGN(SYMBOL(ACHOLDS))THEN
     C5:  EMIT (CHS);
     C4: EMIT (STO, ARG(1));
     C5:  LINE(TEMPS) = 0;
          ACHOLDS = 0; MQHOLDS = 0; TO (STEP)
```

The pattern here contains a "**" which is always matched and a * meaning indirect reference. If the operand in the accumulator, which is described by ACHOLDS is negative a "complement" (CHS) instruction must be emitted. The store command is emitted in any case without any tests on the variable to be replaced. The succeeding actions effect the state of the translator, reclaiming the temporaries and freeing the AC and MQ registers. Finally there is a transfer to the action STEP which sequences through the intermediate code.

The TGS system has been implemented on several computers and has been used in the construction of a variety of compilers. The compiler writers have been professionals and have not been constrained to stay within the formal system. The use of TGS has been sufficiently valuable to COMPASS that they continue to use it on commercial compilers. The main differences between TGS and FSL accurately reflect the difference in design goals: TGS allows more flexibility by requiring more detailed information from the compiler-writer. The efforts of Gries [Grie 67b] at Stanford and Fierst [Fie 66] at Carnegie are attempts to have the best of both by allowing simple code statements as well as multi-phase- processing. Both VITAL [Mond 67] and the most recent TGS [Plas66] are interactive and have sophisticated trace, edit, and debug features.

3. CC (Brooker, Morris, et al. [Brook 67])

The CC (Compiler-Compiler) project at Manchester University is the oldest and one of the most isolated TWS efforts. Rosen [Ros 64a]

has attempted to play Marco Polo to this imperial court, but trade has been slow. The CC system has been running for some time and has been used to implement several algebraic languages [Cou 66, Kerr 67].

The CC effort has concentrated on problems of semantics; the syntax analysis is top-down with memory and one symbol look-ahead (cf. Section IIA). The result of syntax analysis is a complete syntax tree which is used by the semantic phase. This is, of course, a slow process and there are informal provisions for other techniques. We will follow the formal treatment here, taking some liberties with their notation.

The input to the syntax phase is like BNF except for the optional use of a repeat operator (*) to replace simple recursions. The notion of non-terminal symbol is divided into PHRASE and FORMAT. The FORMAT non-terminals may be introduced in macro-fashion and each has an associated (semantic) ROUTINE. The FORMAT symbols are further qualified as [SS], [AS], [BS] meaning respectively source statement, auxiliary statement, and pre-coded basic statement. For example, a source language assignment statement might be defined as:.

FORMAT[SS] = <variable ← <expression> .

Among the useful auxiliary statements would be:

FORMAT[AS] = LOAD <preceeding $\underline{+}$> <term>

FORMAT[AS] = ACC ← ACC <$\underline{+}$> <term>

Each of these would have an associated routine, whose first line contains its calling syntax rule (FORMAT). The routine for the assignment statement might be:

```
1)  ROUTINE[SS] ≡ <variable ← <expression>
2)      LET <expression> ≡ <preceeding ±> <term> <terms>
3)      LOAD <preceeding ±> <term>
4)  L2:  GOTO L1 UNLESS <terms> ≡ <term> <terms>
5)       ACC ← ACC <±> <term>
6)       GOTO L2
7)  L1:  STORE ACC IN <variable>
8)       END
```

In order to understand this routine we need two PHRASE definitions:

```
PHRASE <expression> ≡ <preceeding ±> <term> <terms>
PHRASE <terms> = <±> <term> <terms> | <empty>
```

Notice that the unusual form of recursive definition facilitates sequential code generation.

Line: 1) is the header containing the syntactic construct (FORMAT) associated with this routine. Line 2) is a substitution statement and is not an important consideration here. The rest of the statement is a loop for compiling a string of 'add' and 'subtract' commands and then storing the result. The statement on line 3) is a call on another ROUTINE[AS], this one forming as many successive products and quotients

as possible. Other statement forms such as GOTO and STORE are presumably pre-coded and thus of form ROUTINE[BS]. Notice that statements like that on line 5) imply "using up" syntactic constructs as they are processed.

The built-in part of CC contains, besides [BS] routines, a fairly complete resident system (PERM). There is also a facility for deleting many routines at the completion of the compiler building (PRIMARY) phase. If these routines are left in, the compiler is an extendible one in the sense of Section II.C. In fact, the earlier CC systems would be better described as extendible compilers altogether.

In the earlier versions of CC, the formats and format routines for a language were kept in an encoded form and interpreted by the compiler. The actual mechanism was a tree matching and substitution process somewhat similar to that of Galler and Perlis (cf. Section III.C6). The detailed procedure is quite complicated and is described rather completely in Rosen [Ros 64a]. The current CC system is interesting in that viable extensions to a language can often be "compiled into" the translator with considerable savings in time and space. There are still some routines which must be interpreted and the ratio of the two types for a given extension is not easy to determine.

The CC group has recently produced a number of reports on the uses and performance of their system. These include the first attempt ever to compare a TWS with handwritten compilers [Brook 67]. Brooker was able to (in a year) reduce the space required by a factor of two

and the time by about five by hand coding an Atlas Autocoder compiler. The results are hard to interpret without more information; the formal CC system uses techniques which are intrinsically time and space consuming. One hopes that this attempt will induce the CC group, as well as others, to make more careful studies. There are also two adaptations of CC technique underway in England. The first involves imbedding much of the CC system in the ALGOL-like language ATLAS AUTOCODER [Br 67a]. The other effort is an ambitious attempt to generalize CC to a source and object code independent system [Cou 67].

References for II.D:

Design

Brook 60a, 62a, 63, 67b, 67c, Che 64a, 64c, 65, Cou 67, Feld 64, 66, 67, Fie 67, Grie 67b, Mond 67, Plas 66, Ros 64a, War 61, 64

Uses

Brook 67a, 67b, Cou 66, It 66, Kerr 67, Nap 67, Rov 67

III. Related Topics and Conclusions

III.A. Other Uses of Syntax-Directed Techniques

Very early in the TWS development, it was observed that syntax-directed techniques could be used in a wide variety of problems. A syntax-directed approach can be considered whenever the form of the input to a program contains a significant part of the content. Individual applications of syntax-directed techniques tend not to get written up. The applications presented here are based largely on personal knowledge and, though perhaps representative, are certainly not comprehensive.

The TWS systems described in Section II vary widely in the ease with which they are put to other uses. The syntax-directed symbol processors are the most flexible and seem to be the most widely applied. One such system, AED [Ross 66], was designed from the outset to be a general purpose processor. Because of certain peculiarities of attitude and terminology, the AED project has had little effect on other TWS efforts.

The syntax phase of AED is based on a precedence technique similar to those described in Section II.A. By incorporating type checking and the ability to add hand-coded syntax routines, the AED parser becomes more powerful at the cost of violating the underlying theory. It is, however, the intermediate representation of AED statements that is most interesting. This is based on the use of plexes, which are data structures whose elements each can have many links. The construction and processing of the "modelling plex" are accomplished with a set of macro routines. These might include routines for code generation,

computer graphics or programmed-tool commands. Reference [Ross 63] is a good introduction to the AED system with detailed examples of its use in several problem areas.

The essential features in the AED system are the precedence matrix in syntax and the plex manipulations in semantics. A somewhat different approach to the syntax-directed universe can be developed from the general compiler-compiler model discussed in Section II.D. In this scheme, the entire semantic mechanism, including the choice of data structures, can be different for each application area. In the VITAL [Mond 67] effort, two basically different data structure languages (both written in VITAL) are being compared in a syntax-directed graphics package [Rob 66].

Most of the other applications of TWS systems have been in symbol manipulation tasks of one sort or another. Some of the first applications [Schor 65] were in symbolic mathematics. A TWS would be used to help model the structure of an expression, perhaps for simplification or differentiation. The use of TWS (esp. COGENT, META) in symbolic mathematics is currently widespread and has given rise to systems [Cla 66] constructed specifically for that purpose. There have also been a few pure mathematicians (e.g. [Gro 66]) who have found the syntax-directed model useful.

The most widespread and least surprising application of TWS is in problems of format conversions. These arise in connection with large data files and in translating between closely related source-language to source-language translators. Once again, the syntax-

directed symbol processors of Section II.B have been used the most often. These systems have also been of some use in such varied tasks as: logic design, translating geometric descriptions, and simulation.

There are also a number of applications of TWS techniques to produce command sequences for special purpose devices. For example, a fairly sophisticated TWS [Cas 66] was used in translating commands for various components of a satellite tracking system.

In addition to their direct application in many fields, the TWS have inspired work in several others. One active area has been the syntactic-description of pictures. There are a number of published papers (e.g. [Nar 66]) and a great deal of current work which has not yet seen print. The pattern matching features incorporated in the new list-processing languages [Ab 66, It 66] are partially inspired by TWS.

Computational linguistics, in both its theoretical and practical aspects, is closely related to TWS studies. The applications here, though fewer than one would suspect, have been significant. The syntactic theories of computational linguistics and TWS both are based on the early work of Chomsky [Chom 63] and share many ideas. The implementations of English syntax (esp. [Kun 62]) developed concurrently with top-down TWS, but the natural language efforts have been slow to incorporate the efficiency improvements developed in TWS work. In applied semantics, the DEACON project [Th 66], whose approach was quite novel to linguists, can be looked upon as a straight-forward

application of TWS techniques (cf. [Nap 67]). One can expect to see more interaction between these research areas as linguists attempt to test semantic theories and TWS workers attempt to cope with non-procedural languages.

The last, but by no means the least, of the applications considered here is to teaching. Several of the TWS systems described above have been used as the basis for courses on translator-writing. These have ranged from undergraduate courses to faculty seminars and have been well regarded. Although they can be taught without machine problems, these courses are much more successful when the students have easy access to the TWS under discussion. This approach to teaching was sufficiently appealing to cause D. Knuth at Cal Tech to implement a version of TMG (called TROL) largely for that purpose.

References for III.A.

Ab 66, Brook 67a, Cas 66, Chom 65, Cla 66, Gro 66, Hal 66, It 66, Kun 62, Mond 67, Nap 67, Nar 66, Rob 66, Ross 63, 66, Scho 65, Th 66,

III.B. Related Mathematical Studies

Computer science owes much to mathematics and is beginning to pay off that debt. Both the syntax and semantics of programming languages have inspired formal treatments. In this section we will briefly discuss the developments most relevant to TWS and provide an entree to the literature on the formal aspects of programming languages.

III.B.1. Syntax

We will discuss briefly some theoretically interesting left-right recognizers and their construction algorithms. Of course, given a grammar G and a string x, there is a relatively simple method for testing whether x belongs to L_G. One can generate all strings belonging to L_G of length equal to length (x) and see whether x has been generated. This is not very practical. In contrast to those in II.A., these have not yet been used to write compilers, due to their complexity. The construction algorithms are interesting because they give sufficient conditions for the unambiguity of a grammar, besides mechanically producing the efficient left-right recognizer. By efficient we mean that no backup is necessary - the recognizer can always detect the handle.

a) (1,1) Grammars - Eickel et al. [Ei 63]

By inserting intermediate productions (cf. Section II.A4), the constructor changes the grammar to one consisting of production of length one or two - $U \rightarrow S$ or $U \rightarrow S_1 S_2$.

When looking for a handle at the top of the stack, the two top stack symbols and the incoming terminal symbol must uniquely determine

the step to be taken. Thus, for each triple (S_1,S_2,T) one and only one of the following conditions must hold:

1) S_1S_2 is a handle and one reduction $U ::= S_1S_2$ may be executed.

2) S_2 is a handle and one reduction $U ::= S_2$ may be executed.

3) T must be pushed into the stack.

4) S_1S_2T may not appear as a substring of a sentential form (error).

The algorithm for producing the triples and the corresponding action is given in [Ei 63], along with examples. This algorithm and the recognizers produced have been programmed and tested, but not used to write compilers.

b) Bounded Context Grammars

A grammar is called an (m,n) bounded context grammar if and only if the handle is always uniquely determined by the m symbols to its left and n symbols to its right. A left-right recognizer may thus find the unique canonical parse of a sentence of an (m,n) bounded context grammar by considering at each step at most m symbols to the left (into the stack) and n terminal symbols to the right of a possible handle. The first four types' of grammars discussed in Section II are (1,1) bounded context grammar, as are all grammars accepted by the Eickel-Paul-Bauer-Samelson constructor [Ei 63].

Recognizers for (m,n) bounded context grammars for m> 1, n > 1 are likely to make unreasonable demands on computer time and storage space. Therefore (m,n) bounded context grammars have not been used so far in compilers. There have been three major papers on bounded

context analysis. Each of them defines "context bounded" slightly differently. The idea behind all of them, though, is the same, and we will not discuss the differences here.

The paper by Floyd on Bounded Context [Flo 64a] and the paper by Irons on Structural Connections [Ir 64] should be read by any person interested in delving further into the mysteries of bounded context. However neither gives an algorithm for actually generating the recognizer. Eickel's aim [Ei 64] is to describe the recognizer and its construction in detail (and is therefore less readable than the other two). The recognizer uses the usual stack, and a pointer p to the tail symbol of a possible handle. As in [Ei 63] the grammar is restricted to productions of length 1 or 2 (this is not a restriction on the language). The generator produces 5-tuples

$$(x;S;y,k,U)$$

where x,y are strings with length (x) < m and length (y) < n, S is a symbol, U a non-terminal, and k a number. Suppose the stack contains

$$S_0 \ldots S_{p-1} S_p S_{p+1} \ldots S_i$$

S_p, the symbol at the reduction position, is then tail of a possible handle. The 5-tuples are searched until one is found such that $S = S_p$, x is a tail of $S_0 \ldots S_{P-1}$ and y is a head of $S_{p+1} \ldots S_i$. The step to be taken depends on the corresponding k and U as follows:

k	action
0	stop - syntax error
1	replace handle S_P by U (make a reduction $U \rightarrow S_p$)
2	replace handle $S_{P-1}S_P$ by U (make a reduction $U \rightarrow S_{p-1}S_p$); P +-P-1
3	if p = i then push next symbol onto stack else p $\leftarrow$ p+1
4	push next symbol onto stack (more context needed on the right).

Eickel has programmed and tested both the constructor and recognizer, but no compiler has been written using this technique. The constructor starts by limiting the length of x and y to 1 and producing all possible 5-tuples. If two (or more) 5-tuples exist with the same x,y and S but different i (or the same i but different U), then the grammar is not (1,1) bounded context. For such 5-tuples, the lengths of x and y are alternately (or in some other predetermined order) increased, thus adding more context, until the conflict is resolved or some maximum m,n are reached.

Wirth and Weber [Wir 66c] extended the idea of precedences (see Section II.A2) to strings. Thus we have $x \doteq y$, $x \lessdot y$ and $x \gtrdot y$ where length (x) $\leq$ m and length (y) $\leq$ n. A (m,n) precedence grammar is of course also (m,n) bounded context according to our definition. A precedence grammar according to Section II.A2 is a (1,1) precedence grammar.

c) Deterministic Push-Down Automata (DPDA). Ginsburg and Greibach [Gin 66b]

A DPDA is a formalization of the concept of a left-right recognizer working with a stack and using the usual notation of automata theory - one has a set K of "states" containing a start state $\bar{k}$, a set of inputs α (terminal symbols), a set Γ (corresponding to our nonterminal symbols) containing a start symbol $\bar{U}$, and a mapping δ;

$$\delta : (\text{states} \times (\text{nonterminal symbols}) \times (\text{input symbols})) \rightarrow$$
$$(\text{states} \times (\text{strings of nonterminal symbols}))$$

or

$$\delta : (K \times \Gamma \times (\alpha \cup \{\epsilon\})) \rightarrow (K \times \Gamma^*)$$

This mapping δ must be a function (single valued). Other restrictions are also placed on it to take care of the empty symbol ϵ which may appear anywhere in the input. At each step we have a triple

$$(\underbrace{k}_{\text{state}} , \underbrace{U_1 \ldots U_i}_{\text{stack}} , \underbrace{T_j \ldots T_m}_{\text{rest of input}})$$

(where $i \geq 1$), the initial triple being $(\bar{k}, \bar{U}, T_1 \ldots T_m)$. At each step, with the help of the mapping $(k, U_i, T_j) \rightarrow (k_1, U'_1 \ldots U'_n)$ where $n \geq 0$, the triple gets changed to

$$(k_1, U_1 \ldots U_{i-1} U'_1 \ldots U'_n, T_{j+1} \ldots T_m) .$$

A string (of inputs) is <u>accepted</u> if the final state k_m is a member of a set of final states F.

A language (a set of strings of input symbols derivable from some grammar) is deterministic if it is accepted by some DPDA. Ginsburg and Greibach prove some interesting properties of DPDAs and deterministic languages. Note that a deterministic language is defined by a DPDA - and not by certain properties of the grammar defining the language. What is significant for us here is the relation to LR(k) languages of Knuth (below).

d) LR(k) Grammar (Knuth [Knu 65])

A grammar is LR(k) if and only if a handle is always uniquely determined by the string to its left and the k terminal symbols to its right. The corresponding language is an LR(k) language. Thus, when parsing a sentence using a stack, the left-right recognizer may look at the complete stack (and not just a fixed number of symbols in it) and the following k terminal symbols of the sentence. This is the most general type of grammar for which there exists an efficient left-to-right recognizer that can be mechanically produced from the grammar. In fact, a grammar accepted by any of the other constructors discussed is LR(1). Thus, the LR(k) condition is the most powerful general test for unambiguity that is now available.

Knuth gives two algorithms for deciding whether a grammar is LR(k) or not, for a given k. The second algorithm also constructs the recognizer - if the grammar is LR(k) - essentially in the form of a DPDA (above). Knuth shows that for each LR(k) language L there exists a DPDA which accepts L. Moreover, for each language L accepted by a DPDA there is an LR(1) grammar which defines L. Thus, any LR(k)

language is also LR(1). Earley [Ear 67] has written a constructor for an LR(k) grammar, whose output is in the form of productions, similar to but more complicated than the Floyd-Evans productions.

e) Recursive functions of regular expressions (Tixier [Tix 67])

Many compilers break the syntax analysis into small parts. Thus, one subroutine will recognize <expressions> while another will handle <declarations>. A saving of space arises because the character set involved in each subroutine is quite small. For instance, one might have three 20 x 20 precedence matrices instead of one 60 x 60 matrix. Tixier has formalized this concept quite nicely in his thesis.

One can consider a non-terminal symbol as a variable denoting the set of terminal strings which are derivable from it. The productions can then be transformed into sets of equations using the set operations union (+), product and closure (*). Thus the productions

<identifier> ← <letter>

<identifier> ← <identifier> <letter>

may be written equivalently as

<identifier9 = <letter> + <identifier> <letter>

or

<identifier> = <letter3 <letter>*

Tixier has rewritten the 120 productions for Euler [Wir 67c] as 7 functions of 7 variables, 3 of which we give here (the symbols "(",

")" are meta-symbols used to bracket set expressions):

program = ⊥block⊥

block = begin((new id + label id);)*(i:)*expr(;(i:)*expr)*end

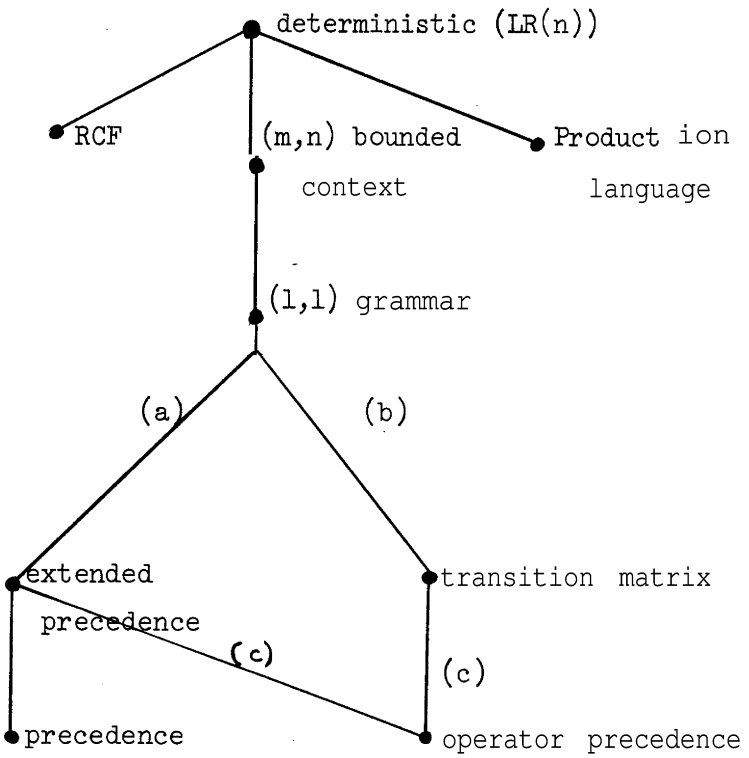
expr = (out + if expr then expr else + i ([expr]+.)* ←)*

(goto primary + block + catena)

The point is that one can now mechanically construct a finite state automaton, which is very efficient, to accept each of the above expressions. One can then connect these automata by a pushdown stack, so that they may (recursively) call each other. Thus, when the finite state automaton for "program" (see above) decides that a "block" must be recognized, it places in the stack a return point to itself and calls the "block" automata.

Tixier has formalized this in his thesis and shows how to construct an efficient restricted DPDA for a certain class of grammars, called RCF, These languages are thus LR(1).

The diagram below presents an inclusion tree for the classes of grammars accepted by the particular constructors discussed in this section and in Section II.A.



(a) Although (1,1) grammars and extended precedence grammars both use triples, the advantage for (1,1) grammars arises from the automatic intermediate reductions performed, which essentially allows more context.

(b) Transition matrix grammars fall somewhere between (1,1) and (0,1) bounded context.

(c) We are making the assumption here that the operator precedence conditions have been augmented to include conditions for a unique canonical parse (cf. Section II.A1). Otherwise inclusion does not hold. The advantage of the matrix technique over operator precedence is, as in (a), the use of automatic intermediate reductions.

References for III.B.1,

Introduction to the theory of formal languages

Bar 64, Gins 66a.

Pure or modified top-down algorithms

Barn 62, Br 62a, Che 64c, GraR 64, Ing 66, Ir 63a, Kun 62, Kir 66, Rey 65, Scho 65, War 64.

Construction of efficient recognizers - sufficient conditions for unambiguity

Ea 65, Ea 67, Ei 63, Ei 64, Flo 63, Flo 64b, Gins 66b, Grie 67a, Ir 64, Knu 65, McKee 66, Paul 62, Wir 66c, Tix 67.

Surveys, tutorials on recognizer techniques

Che 64c, Flo 64b, GraM 64.

Ambiguity in context free languages

Can 62, Flo 62a, Flo 62b, Gor 63, Lang 64, Ross 64.

13 different ways to define languages

Gorn 61.

III.B.2. Semantics

Any formal study of the semantics of programming languages immediately confronts the problem of separating syntax from semantics. Programming languages combine ideas from logic (where the problem is solved) and natural language (where it is no longer taken seriously). In most treatments of programming languages, syntax is taken to be precisely these aspects of language describable in the syntactic meta-language under discussion. This practice has the unpleasant effect of changing the definition of syntax with each change in meta-language.

Computer scientists trained in logic (e.g. [Tix 67]) would like us to adopt the definitions used there although this approach has not proved effective for natural language and has immediate problems in programming languages. For example, are the statements

```
x ← Y / 0.0
L1: GOTO L1
```

well-formed in ALGOL 60? Surely, an algorithm capable of handling data types could detect these errors, and the question is now one of how far to go. It is not obvious that one could produce a notion of syntax which satisfied a logician's tastes and still left well-formedness a decidable property.

The situation is further complicated by the fact that all major languages contain statements unparseable by the formal syntax alone. An example from ALGOL 60 is:

X ← IF B THEN C ELSE D > E

the structure of which depends on whether "C" is Boolean or arithmetic. Thus, in practice, syntax-directed compilers must incorporate "semantic" features in the syntax phase. One ingenious approach to the separation question is the abstract syntax [McCar 62a] of McCarthy. He is mainly concerned with semantics and considers (analytic) syntax to be just the set of predicates and functions necessary to extract pertinent information from the form of a source string. This does not "solve" the problem of defining syntax but does enable one to consider semantics without facing the separation question.

As usual, formal studies of semantics have lagged behind work on the syntax of programming languages. By far the best general work on this subject is [Ste 66] where the discussions, even more than the papers, provide an overview of formal semantics. The various formalizations that have been presented are all procedural; they are either abstract machines or imperative formalisms such as the h-calculus [Chu 51]. This is reasonable to expect, but greatly restricts the choice of existing mathematical models.

Since the formalizations are procedural one might prefer the word "effect" to "meaning" in the description of programming languages. This is not the place to defend the notion of semantics as effect and we will adopt it merely as a convenient way of looking at things. This view does lead one to expect a program to have different effects depending on an "environment" and this will prove useful in our discussion.

It might also lead one to suspect that the choice of semantic meta-language will be influenced by the intended use of a formal description.

The existing efforts in formal semantics may be separated into those concerned with proofs about programs and those interested in elucidating the processing of programs by computers. Among the latter, one might include the semantic meta-languages described in Section II.D, although this is not de rigueur. There are, however, slightly abstracted translation models (e.g. [Wir 66c]) which are considered acceptable. In any such model, a language can have very different effects depending on whether its translator is an interpreter or a compiler. This seems reasonable to programmers, but disturbs mathematical types who would prefer to see meaning reside in the algorithm rather than the program. A related set of developments are the attempts to define all programming languages by reduction to a single high level [Ste66] or machine-like [Brat 61, Ste 61] language.

The approaches to formalization described above are more closely related to TWS, but are far too complex to be very useful in proofs. For those who consider proofs to be the sole end of formalization (and would be reading this paper at all) the preceding paragraph will be considered an anathema. An interesting halfway house is to be found in the work of Van Wijngaarten and de Bakker [Bak 65 , Wij 66]. They attempted to reduce the complexity of their model by using a universal Turing machine. This machine had only a few rules, which would interpret additional rules, eventually forming a translator which would recursively translate e.g. ALGOL. The difficulty was that the formalism was so primitive that the ALGOL semantics became a large paper and

neither proofs nor insight seemed to result.

Most mathematically based attempts at formalization have stressed tractability and have almost all been based on existing mathematics. There are only a few imperative systems in logic, and each has been used in formalizing some aspect of computer science. Most of the work in formal semantics is based on the h-calculus of Church [Chu 51] and the combinator calculus of Curry [Cur 58].

Both of these theories were primarily concerned with the role of variables and their successes in programming languages have been largely in that area. The h-expression plays a crucial role in LISP and is discussed as a programming concept in various LISP documents It is also the most popular vehicle for attempting to formalize semantics. The work of Landin and Strachey [Lande 66] is particularly interesting because they combined their research with the development of an extension of ALGOL 60 called CPL [Burs 65, Cou 65].

The applications of h-calculus to semantics have been pursued most diligently by Landin. In a series of papers he considers the relationships between programming languages (ALGOL) and an augmented h-calculus called imperative applicative expressions (IAE). The declaration and binding of variables in ALGOL is modelled quite clearly and the formalization has helped point out some weak spots in ALGOL. The IAE system (like pure LISP) is purely functional and must represent statements as 0-adic functions with side effects on the environment. In fact, much of Landin's description of ALGOL can be viewed as a generalization of the "program feature" in LISP [McCar 62b]. Thus far, these efforts

have neither achieved the descriptive clarity nor maintained the tractability of h-calculus in accordance with the original plan. The most conspicuous benefit of the work has been CPL [Cou 66] which is an extremely civilized language. There is presently an active group at M.I.T. which is pushing this approach as far as it is ever likely to go.

Although he introduced the h-calculus into computer science, McCarthy has taken a somewhat different approach to formal semantics. His term "theory of computation" indicates that he is more concerned with algorithms than with algorithmic languages. His approach utilizes a state vector, operations upon it, abstract syntax and conditional expressions. Typical state functions are

$C(x,\alpha)$

$A(x,z,\alpha)$

read the contents of symbolic position 'x' in state vector 'α' and the state resulting from substituting 'z' for 'x' in state vector 'α'. He is then able to get conditional expression definitions of machine-code-like operations and constructs found by the abstract syntax. The resulting formalism is fairly tractable and McCarthy and his students have been able to push through a number of proofs [McCar 67].

A more recent, and intuitively more satisfying, approach has been developed by Floyd [Flo 67]. He considers the flow chart of a program written in an ordinary (fixed) programming language. The basic idea

is to attach a proposition to each connection in the flow chart; the proposition is to hold whenever that connection is taken during execution (thought of as interpretation). With these propositions and some related mechanisms, Floyd establishes techniques for proving properties of the form "If the initial state satisfied R1 then the final state will satisfy R2, if reached." Proofs of termination are handled by showing that some function of, say, the positive integers decreases as the program is executed. There are current efforts to automate both the generation of propositions and the proofs of correctness for restricted languages.

Our description of the work in formal semantics has been sufficiently shallow to perhaps be misleading. Most of these efforts have their comrades and fellow-travellers and the development has been richer than we suggested; the references at the end of this section should cover all major trends related to TWS. The impact of formal semantics, especially the proof-oriented kind, has been limited to a few isolated insights. There has been no work having the impact of e.g. Krohn and Rhodes on automata theory. It is our conjecture that this breakthrough is not to be found-in existing imperative logics; programming languages will have to be faced directly as mathematical and natural languages have been.

References for III.B.2

Bak 65, Braf 63, Burg 64, Burs 65, Chu 51, Cal 62, Cur 58, Flo 67, Ir 61, Ir 63b, Landi 63, 65, 66, Luc 65, McCar 62a, 67, Org 66, Rig 62, Ste 64, Tars 56, Tix 67, Zem 66.

III.C. Summary and Research Problems

The TWS described in this paper represent the most recent developments in a long line of research by many outstanding computer scientists. Each category described in Section II has its peculiar strengths and weaknesses and a preferred problem domain. After a brief summary of the relations between the various categories, we will suggest a number of fruitful areas for future research.

The automatic constructors of recognizers, described in Section II.A, are tools which are potentially useful in any problem attacked with a syntax-directed approach. By automatically producing an efficient recognizer, such systems should extend the useful range of syntax-directed techniques. The major problem is to find a convenient way of embedding semantic definitions in the synthetic syntax. A solution to this problem would also produce a marked improvement in the capabilities of the syntax-directed symbol processors of Section II.B. These TWS all have fairly convenient methods for introducing semantics, but all share the use of relatively inefficient recognizers. The already far-reaching applications of such systems could be significantly widened by the development of more efficient recognizers.

The meta-assemblers described in II.C are presently much better suited to assembler-writing than compiler-writing. They have, however, introduced several significant additions to macro languages which will have a long range effect. By extending the facilities of meta-assemblers for translation-time actions and adding a syntax phase one could make them comparable to the syntax-directed symbol processors of Section II.B.

The work on extendible compilers is more recent and difficult to assess accurately, Although it seems clear that some macro facility should be included in any high-level language, the more exotic systems may be limited in their usefulness. In any event, it seems unlikely that extendible compilers will compete with compiler-compilers in the original implementation or radical change of a translator.

The compiler-compilers of Section II.D are the high point in the evolution of specialized TWS. This specialization has made them by far the most useful for compiler-writing, but has its attendent costs. The compiler-compilers are harder to implement and are often unsuited to tasks appreciably different from compiling. As the semantic languages attempt to encompass more sophisticated programming constructs, one can expect the specialization to become even more pronounced. There is, however, a tendency to allow the insertion of different specialized semantic languages in a TWS, preserving the syntax and system features.

None of the TWS discussed here is a panacea. We have attempted to show that it is unreasonable to expect one and the results of various attempts at a universal programming system of any kind tend to support this position. We do feel that, taken as a whole, the TWS efforts have solved many of the significant problems in compiler writing and documentation [Naur 63a]. There are now enough available techniques to satisfy a great variety of possible TWS requirements. It is our contention that future work on general TWS should be considered development and perhaps undertaken by a different set of people. The area most suitable for research seems to be the careful consideration of a number of isolated problems related to TWS.

The syntactic aspects of TWS have received considerable attention and have fewer outstanding questions. The three problems that do come to mind are closely related to semantics and to one another. One problem is to find a satisfactory way of embedding extra-syntactic features to allow "syntax" to correspond more closely to one's intuition [Gil 66]. A related issue is the absence of an adequate technique for embedding semantics in the rules of a synthetic grammar without knowledge of the details of the recognizer constructing program being used. Finally, there is the problem of graceful degradation (this year's OK phrase) in automatic recognizer constructing programs. One would like the system to use efficient techniques where possible and automatically move to more general schemes (rather than quit) when the going gets rough.

There has been much less work on the post-syntactic aspects of TWS. There have been three basically different approaches to this "semantics" problem. The first approach is to provide a general purpose list-processing.or other symbol manipulation capability (cf. Section II.B). The second is to provide a number of data structures and built-in routines especially designed for compiler-writing (cf. II.D2). The third approach partakes of the first two, but also attempts to automate significant parts of the compiler-writing task (cf. II.D). By making use of macros and subroutines, either of the first two techniques can look, to the average user, like the highly automated system. From this point of view, the key problem in semantics is finding general purpose routines for handling significant aspects of compiler writing. We feel that the TWS approach has been

proven feasible and that the general problem should now be considered in the development stage. There are, to be sure, several kinds of programming languages (e.g. simulation [Te 66]) still beyond the pale, but each has a few basic concepts that need to be studied first. In short, future research in TWS should be directed toward understanding (and eventually, automating) the outstanding problems in programming languages.

With this formulation of TWS research, we have, of course, provided a guaranteed annual project for everyone. A justification for this can be found in the many contributions to programming systems which have resulted from considering meta-problems. In the remainder of this section, we will discuss a number of interesting problems which might be amenable to a TWS approach and provide an entrée into the literature for each. The references listed at the end of the section for each subject are either very recent or comprehensive or are already used as a reference in this paper.

One question of long standing that is still open is the formal description of machine languages. A solution here could be used as a third input to a TWS, describing the target machine. This problem has been attacked, both theoretically and directly, but nothing has come close to being usable by a TWS. The availability of parallel processors adds a new level of complexity or, better, a new research area. Most of the work on software for parallel processors has been concerned with particular machines and is not within the scope of this paper. There have been some significant abstract [Kar 66] and concrete [Shed 67, Sto 67] theories which might serve as a foundation for research in

parallelism. Parallelism in high level languages [Dij 65] is also beginning to receive attention.

Another hoary question concerns a theory of code selection and enhancement (the "optimization" problem). Not only has the theory been weak, but there are still only a half-dozen or so types of code enhancement in general use by compiler writers. The most striking improvements in program performance usually come from restructuring the entire approach to the problem. This could be-called optimization-in-the-large but we will discuss it as one aspect of non-procedural programming. The accepted definition of "non-procedural", like that of "semantics", has yet to appear. A programming system will be called non-procedural to the extent that it makes selections and rearrangements of procedural steps in response to some higher order problem statement.

Non-procedural programming languages have been discussed under many rubrics: declarative languages, problem-oriented languages, questionnaire systems and the like. Most of this work is theoretically uninteresting (cf. [YOU 65]); one writes a large routine and the user supplies parameters. Fairly good non-procedural systems for limited problem areas have been developed in computer graphics, relational languages [Rov 67], array processing [Gal 67] and numerical analysis [Ri 66]. The analogue computer, of course, has always been programmed this way and some promising systems [Schl 67] are being developed by extending the languages used in hybrid computing. Cheatham envisions adding non-procedural features of a general sort to the extendible compiler discussed in Section III.C5. Another approach would be to use

the more sophisticated syntax forms and transformations developed in natural language processing.

We have felt for some time that TWS efforts shared many interests with natural language systems. There have been the so-called query languages [Corn 66] and, of course, COBOL [Samm 61], but these make only superficial contact with the problem. The recent interest in conversational and non-procedural programming languages along with the syntax-directed natural language systems (cf. Section III.A) should lead to a significant interchange of ideas.

There are several open problems concerning the connection between TWS and executive systems. One of the major benefits of a TWS is eliminating the effort (often more than half the total) of interfacing each compiler to the executive. One indication of the past work in this area is that the word "executive" has not occurred before this paragraph. There has always been a small group interested in "environmental" questions for compilers [Le 66], but they had little effect before the time-sharing revolution. The (hoped for) availability of multi-access time-sharing systems gives rise to several additional research problems related to TWS.

The main task of any large time-sharing executive is resource allocation. The resources to be allocated include programs such as compilers as well as various memory and processing units. The research problem is to devise a scheme for allowing translators to exchange information with the executive so as to produce significantly better system performance. The most pressing need in current systems is for main memory, and there have been several schemes [Bob 67, Coh 67, Rov 67]

to help reduce swapping for particular languages. A related problem is the optimal (not maximal) use of pure procedure in both the TWS [Feld 67] and the resulting object code. While an elegant compiler-executive interface will be very difficult to achieve, even a theoretically uninteresting solution should prove of great practical value.

There are two other problems relating to executive systems which we will mention briefly here. Control languages should be improved by adding syntax processing; ideally using the same syntax code already in the TWS. A more ambitious project would be the application of syntax-directed techniques to the construction of executive programs themselves. One additional related problem is debugging aids. There has been a great deal of work on on-line debugging systems [EvT 66], but most of it has been at the assembly language level. There have been some good symbolic dump facilities, in particular batch-made compilers but these have not found their way into print or into TWS. There has also been very little effort [Ir 65] on the problems of automatic error detection and recovery in syntax-directed processors. Once again, even a bad system would be of great value to users.

The final research area to be discussed here is the study of data structures. This field seems to include everything from matrix manipulations to file handling, and has strong interrelationships with almost everything. In some sense, data structures are the current problem in computer science and it would be presumptuous to try to survey the outstanding issues. We will mention a few aspects connected with TWS and indicate how data structure considerations occur in the other research problems mentioned here.

One central question in any TWS is the choice of data structures built-in at both translation and execution time. The survey in Chapter II describes the translation time structures; essentially nothing has been done to provide built-in structure operators for execution time. Many sophisticated data-structure languages have been written using TWS (e.g. [Ab 66, It 66, Rov 67]), but the structure operators have all been hand-coded. There have been several recent attempts (e.g. [Ross 66, IBM 66, Wir 66b] to devise a single general data-structure; such a structure could easily be incorporated in a TWS. The problem is that current proposals all become very inefficient in some area where data-structures are now applied. The question of choosing the right structure for a given algorithm takes one far into non-procedural programming. Similarly, one could make major advances in global optimization and natural language processing with data-structure improvements. In fact, there are rich connections among all the research problems mentioned here and many others as well; the TWS problem will, by its nature, always be related to several frontiers of programming research.

Our brief survey of recent TWS efforts has turned out to be an embarassingly long paper. We have attempted to show how a large number of bright people, working almost in isolation, have brought about a reasonable understanding of many aspects of systems programming, With better communication and higher scientific standards, one could hope for even more significant advances and more rapid application of the ideas developed in research. It was this hope that led us to write this paper and perhaps led you to read it.

References for III.C

Theory of Machine Instructions

Brat 61, Bur 64, Car 62, Gil 67, Maur 65, Ste 61.

Parallelism

Dij 65, Kar 66, Kuc 67, Mar 67, Shed 67, Sto 67.

Code Selection and Enhancement

General references, Ar 63, Grie 65, Hill 62.

Non-Procedural Languages

Che 66, Gall 67, Ri 66, Rov 67, Schl 67,. Sib 61, Wil 64b, You 65.

Natural Language Processing

Bar 64, Chom 65, Cra 66, Hal 66, Int 63, Kun 62, Nap 67, Th 66.

Executive Interface

Feld 67, Le 66, Orch 66, Nob 63.

Paging

Bob 67, Coh 67, Den 65, Rov 67.

Debugging

EVA 63, EvT 66, Ir 65.

Data Structures

Ab 66, Brook 67c, Gall 67, IBM 66, It 66, Pra 65, Pra 66, Rov 67, Wir 66b.

BIBLIOGRAPHY

Ab 66 Abrahams, P. W. The LISP 2 programming language and system. Proc. AFIPS FJCC (1966), 661-676.

Ar 63 Arden, B. W., Galler, B. A. and Graham, R. M. An algorithm for equivalence declarations. Comm. ACM 4 (July 1963), 330-334.

Ar 66 Arden, B. W., Galler, B. A. and Graham, R. M. Michigan Algorithmic Decoder. University of Michigan Press, 1966.

Bac 57 Backus, J. W. et al. The FORTRAN automatic coding system. Proc. Winter Joint Comput. Conf. (1957), 188-198.

Bac 59 Backus, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. Proc. International Conf. on Information Processing, UNESCO (1959), 125-132.

- Bak 65 de Bakker, J. Formal definition of algorithmic languages. MR74 Mathematisch Centrum, Amsterdam (May 1965).

Bar 64 Bar Hillel, Y. Language and Information. Addison-Wesley, Palo Alto, 1964.

Barn 62 Barnett, M. P. and Futrelle, R. P. Syntactic analysis by digital computer. Comm. ACM 5 (Oct. 1962), 515-526.

Bart 63 Barton, R. S. A critical review of the state of the programming art. Proc. SJCC (1963), 169-177.

Ben 64a Bennett, R. K. and Kvilekval, A. SET, self extending translator. Data Processing, Inc. (March 1964).

Ben 64b Bennett, R. K. and Neumann, D. H. Extension of existing compilers by sophisticated use of macros. Comm. ACM 7 (Sept. 1964), 541 (actually about assemblers).

Ber 62 Berman, R., Sharp, J. and Sturges, L. Syntactical charts of COBOL 61. Comm. ACM 5 (May 1962), 260.

Bob 67 Bobrow, D. G. and Murphy, D. Structure of a LISP system using two-level storage. Comm. ACM 10 (March 1967), 155-160.

Braf 63 Braffort, P. and Hirschberg, D. (Eds.) Computer Programming and Formal Systems. North-Holland Publishing Co., Amsterdam, 1963.

Brat 61 Bratman, H. An alternate form of the UNCOL diagram. Comm. ACM 4 (March 1961), 142.

Bro 63 Brown, S. A., Drayton, C. E. and Mittman, B. A description of the APT language. Comm. ACM 6 (Nov. 1963), 649-658.

Brook 60a Brooker, R. A. and Morris, D. An assembly program for a phrase structure language. Comp. J. 3 (1960), 168-174.

Brook 60b Brooker, R. and Morris, D. Some proposals for the realization of a certain assembly program. Comp. J. 3 (1960), 220-231.

Brook 61 Brooker, R. and Morris, D. A description of Mercury Autocode in terms of a phrase structure language, Annual Review in Automatic Programming 2 (1961), 29-66.

Brook 62a Brooker, R. and Morris, D. A general translation program for phrase structure languages. J. ACM 9 (Jan. 1962), 1-10.

Brook 62b Brooker, R. et al. Trees and routines. Comp. J. 5 (1962), 33-47.

Brook 63 Brooker, R. et al. The compiler compiler. Annual Review in Automatic Programming 3 (1963), 229.

Brook 67a Brooker, R., Morris, D. and Rohl, J. S. Compiler compiler facilities in Atlas Autocode. Comp. J. 9 (1967), 350-352.

Brook 67b Brooker, R., Morris, D. and Rohl, J. S. Experience with the compiler compiler. Comp. J. 9 (1967), 345-349.

Brook 67c Brooker, R. and Rohl, J. S. Simply partioned data structures: The compiler-compiler reexamined. Machine Intelligence I (ed. by Collins and Michie) Oliver and Boyd (1967).

Burg 64 Burge, W. H. The evaluation, classification and interpretation of expressions. Proc. 19th Natl. ACM Conf., Phila. (1964), A1.4.

Burk 65 Burkhardt, W. Universal programming languages and processors. Proc. FJCC, Las Vegas (1965), 1-21.

Burs 65 Burstell, R. M. Some aspects of CPL semantics. 3, Experimental Programming Rpts., Edinburgh University (April, 1965).

Can 62 Cantor, D. G. On the ambiguity problem of Backus Systems. J. ACM 9 (Oct. 1962), 477-479.

car 62 Caracciolo DiForino, A. On a research project in the field of languages for processor construction. Proc. IFIP (1962), 514-515.

Car 63 Caracciolo DiForino, A. Some remarks on the syntax of symbolic programming languages- Comm. ACM 6 (Aug. 1963), 456.

Cas 66 Castle, J. A command program compiler. General Electric MSD, King of Prussia, Pa. (1966).

Che 64a Cheatham, T. E. The architecture of compilers. CAD-64-2-R, Comp. Assoc. (1964).

Che 64b Cheatham, T. E. et al. Preliminary description of the translator generator system--II. CA-&-1-SD, Comp. Assoc. (1964).

Che 64c Cheatham, T. E. and Sattley, K. Syntax directed compiling. Proc. AFIPS SJCC (1964), 31-57.

Che 65 Cheatham, T. E. The TGS-II translator-generator system. IFIP Congr., New York (1965).

Che 66 Cheatham, T. E. The introduction of definitional facilities into higher level programming languages. Proc. AFIPS FJCC (1966), 623-637.

Chom 63 Chomsky, N. Formal properties of grammars. Handbook of Mathematical Psychology, Vol. 2, Luce, Bush and Galanter (Eds.), J. Wiley, New York (1963), 323-418.

Chom 65 Chomsky, N. Aspects of the Theory of Syntax. The MIT Press, Cambridge, Mass., 1965.

Chu 51 Church, A. The Calculi of Lambda-Conversion. Annals of Math. Studies, no. 6, Princeton University Press, Princeton, 1951.

Cla 66 Clapp, L. A syntax-directed approach to automated aids for symbolic math. Summary in Comm. ACM 9 (Aug. 1966), 549.

Coh 67 Cohen, J. A use of fast and slow memories in list processing languages. Comm. ACM 10,2 (Feb. 1967), 82-86.

Con 63 Conway, M. E. Design of a separable transition-diagram compiler. Comm. ACM 6 (July 1963), 396.

Conn 66 Connors, T. B. ADAM - A generalized data management system. Proc. AFIPS SJCC (1966), 193-203.

cou 66 Coulouris, G. F. and Goodey, T. J. The CPL1 system manual. PID12/GFC, Inst. of Comp. Sci., University of London.

Cou 67 Coulouris, G. The compiler processor project. Internal Report, Imperial College, London (April 1967).

Cra 66 Craig, J. A., Berezner, S. C., Carney, H. C. and Longyear, Co R. DEACON: Direct English Access and CONtrol. Proc. AFIPS FJCC (1966), 365-380.

Cul 62 Culik, K. Formal structure of Algol and simplification of its description. Symbolic Languages in Data Processing, Gordon and Breach, New York (1962), 75-82.

Cur 58 Curry, H. B. and Feys, R. Combinatory Logic, Vol. I, North Holland, Amsterdam, 1958.

Den 65 Dennis, J. B. Segmentation and the design of multiprogrammed computer systems, J. ACM 12 (Oct. 1965), 589-602.

Dij 63 Dijkstra, E. W. On the design of machine independent programming languages. Annual Review in Automatic Programming 3 (1963), 27-42.

Dij 65 Dijkstra, E. W. Solution of a problem in concurrent programming control. Comm. ACM 8 (Sept. 1965), 569.

Ear 65 Earley, J. C. Generating a recognizer for a BNF grammar. Computation Center Report, Carnegie Inst. of Tech. (1965).

Ear 67 Earley, J. C. An LR(K) parsing algorithm. Mimeo, Carnegie Inst. of Tech. (1967).

Ei 63 Eickel, J., Paul, M., Bauer, F. L. and Samelson, K. A syntax controlled generator of formal language processors. Comm. ACM 6 (Aug. 1963), 451-455.

Ei 64 Eickel, J. Generation of parsing algorithms for Chomsky 2-type languages. 6401, Mathematisches Institut der Technischen Hochschule München (1964).

Eng 61 Englund, D. and Clark, E. The CLIP-translator. Comm. ACM 4 (Jan. 1961), 19-22.

EvA 64 Evans, Arthur. An ALGOL 60 compiler. Annual Review in Automatic Programming 4 (1964), 87-124.

EvT 66 Evans, T. and Darley, D. On-line debugging techniques: A survey. Proc. FJCC (1966), 37-50.

Feld 64 Feldman, J. A. A formal semantics for computer oriented languages. Carnegie Inst. of Tech. (1964).

Feld 66 Feldman, J. A. A formal semantics for computer languages and its application in a compiler-compiler. Comm. ACM 9 (Jan. 1966), 3-9.

Feld 67 Feldman, J. A. and Curry, J. The compiler-compiler in a time sharing environment. Lecture notes on Advanced Computer Organization, University of Michigan (1967).

Fer 66 Ferguson, D. E. Evolution of the meta-assembly program. Comm. ACM 9 (March 1966), 190-196.

Fie 67 Fierst, J. CABAL Memos. Computer Center Reports, Carnegie Inst. of Tech. (1967).

Flo 61 Floyd, R. W. A descriptive language for symbol manipulation. J. ACM 8 (Oct. 1961), 579-584.

Flo 62a Floyd, R. W. On ambiguity in phrase structure languages. Comm. ACM 5 (Oct. 1962), 526, 534.

Flo 62b Floyd, R. W. On the non-existence of a phrase structure grammar for ALGOL-60. Comm. ACM 5 (Sept. 1962), 483-484.

Flo 63 Floyd, R. W. Syntactic analysis and operator precedence. J. ACM 10 (July 1963), 316-333.

Flo 64a Floyd, R. W. Bounded context syntactic analysis. Comm. ACM 7 (Feb. 1964), 62-67.

Flo 64b Floyd, R. W. The syntax of programming languages -- a survey. IEEE Transactions Electronic Computers 13,4 (Aug. 1964), 346-353.

Flo 67 Floyd, R. W. Assigning meanings to programs. AMS Symposium in Appl. Math. 19 (1967).

Gall 67 Galler, B. and Perlis, A. J. A proposal for definitions in ALGOL. Comm. ACM 10 (April 1967).

Gar 64 Garwick, J. V. Gargoyle, a language for compiler writing. Comm. ACM 7 (Jan. 1964), 16.

Gil 66 Gilbert, P. On the syntax of algorithmic languages. J. ACM 13 (Jar. 1966), 90-107.

Gil 67 Gilbert, P. and McLellan, W. G. Compiler generation using formal specification of procedure-oriented and machine languages. Proc. AFIPS SJCC (1967), 447-455.

Gin 66a Ginsburg, S. The Mathematical Theory of Context Free Languages. McGraw-Hill, New York, 1966.

Gin 66b Ginsburg, S. and Greibach, S. Deterministic context free languages. Information and Control 9 (1966), 620-648.

Gle 60 Glennie, A. E. On the syntax machine and the construction of a universal compiler. Tech. Rpt. No. 2, Computation Center, Carnegie Inst. of Tech. (1960).

Gor 61 Gorn, S. Specification languages for mechanical languages and their processors, a baker's dozen. Comm. ACM 4 (Dec. 1961), 532-542.

Gor 63 Gorn, S. Detection of generative ambiguities in context-free mechanical languages. J. ACM 10 (April 1963), 196-208.

GraM 65 Graham, M. L. and Ingerman, P. Z. A universal assembly mapping language. Proc. 20th Natl. ACM Conf., Cleveland, Ohio (1965).

GraR 64 Graham, R. M. Bounded context translation. Proc. AFIPS SJCC (1964), 17-29.

Grau 62 Grau, A. A. A translator-oriented symbolic programming language- J. ACM 9 (April 1962), 480-487.

Gre 62 Green, J. Symposium on languages for processor construction. Proc. IFIP (1962), 513-517.

Grie 65 Gries, D., Paul, M. and Wiehle, H. R. Some techniques used in the ALCOR-ILLINOIS 7090. Comm. ACM 8 (Aug. 1965), 496-500.

Grie 67a Gries, D. The use of transition matrices in compiling. Tech. Rpt. CS 57, Computer Science Dept., Stanford University (March 1967).

Grie 67b Gries, D. Internal notes on the compiler writing system. Computer Science Dept., Stanford University (1967).

Grif 65 Griffiths, T. V. and Petrick, S. R. On the relative efficiencies of context-free grammar recognizers. Comm. ACM 8 (May 1965), 289-299.

Gro 66 Gross, M. Applications geometriques des langages formels. ICC Bull. 5 (Sept. 1966), 141-167.

Hal 64 Halpern, M. XPOP: a meta-language without metaphysics. Proc. AFIPS FJCC (1964), 57-68.

Hal 66 Halpern, M. Foundations of the case for natural-language programming. Proc. AFIPS FJCC (1966), 639-649.

Hals 62 Halstead, M. H. Machine-Independent Computer Programming. Spartan Books, Washington, D. C., 1962.

Hill 62 Hill, V., Langmaack, H., Schwarz, H. R., and Seegmüller, G. Efficient handling of subscripted variables in ALGOL 60 compilers. Proc. Symbolic Languages in Data Processing, Gordon and Breach, New York, 1962, 331-340.

Hoa 65 Hoare, C. A. R. A programming language for processor construction. IFIP Congr., New York (1965).

Hus 62 Huskey, Harry D. Languages for aiding compiler writing (panel discussion). Symbolic Languages in Data Processing, Gordon and Breach, New York, 1962, 187-204.

IBM 66 IBM System/360 Operating System PL/I Language Specification. Form C28-6571-4.

Ing 62 Ingerman, P. Z. Techniques for processor construction. Proc. IFIP (1962), 527-528.

Ing 66 Ingerman, P. Z. A Syntax Oriented Translator. Academic Press, New York, 1966.

Int 63 International Standards Organization. Survey of programming languages and processors. Comm. ACM 6 (March 1963), 93.

Ir 61 Irons, E. T. A syntax directed compiler for ALGOL 60. Comm. ACM 4 (Jan. 1961), 51-55.

Ir 63a Irons, E. T. The structure and use of the syntax-directed compiler. Annual Review in Automatic Programming 3 (1963), 207-227.

Ir 63b Irons, E. T. Towards more versatile mechanical translators. Proc. Symposia Appl. Math 15 (1963), 41-50.

Ir 64 Irons, E. T. Structural connections in formal languages. Comm. ACM 7 (Feb. 1964), 67-71.

Ir 65 Irons, E. T. An error correcting parse algorithm, Comm. ACM 6 (Nov. 1965), 669-673.

It 66 Iturriaga, R., Standish, T. A.,, Krutar, R. A. and Earley, J. C. Techniques and advantages of using the formal compiler writing system FSL to implement a formula Algol compiler. Proc. AFIPS SJCC (1966), 241-252.

Kar 66 Karp, R. M. and Miller, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. Appl. Math. 14 (Nov. 1966), 1390-1411.

Kerr 67 Kerr, R. H. and Clegg, J. The Atlas Algol Compiler - an ICT implementation of Algol using the Brooker-Morris Syntax Directed Compiler. Comp. J. (1967).

Kir 66 Kirkley, C. and Rulifson, J. LOTS, a syntax-directed compiler. Internal Report, Stanford Research Institute (May 1966).

Knu 62 Knuth, D. E. History of writing compilers. Digest of Technical Papers, ACM Natl. Conf. (1962), 43, 126.

Knu 65 Knuth, D. E. On the translation of languages from left to right. Information and Control 8 (Oct. 1965), 607-639.

Kuc 67 Kuck, D. Programming the Illiac IV, Talk given at AFIP SJCC (1967). Paper not yet available,

Kun 62 Kuno, S. and Oettinger, A. G. Multiple-path syntactic analyzer, Information Processing 62 (IFIP Congress), Popplewell (Ed.), North-Holland Publishing Co., Amsterdam (1962), 306-311.

Lande 62 Landen, W. H. and Wattenburg, W. H. On the efficient construction of automatic programming systems. Digest of Technical Papers, ACM Natl. Conf. (1962), 91.

Landi 63 Landin, P. J. The mechanical evaluation of expressions. Comp. J. 6 (1963), 308.

Landi 65 Landin, P. J. A correspondence between ALGOL 60 and Church's h-notation. Comm. ACM 8 (Feb. and March 1965), 89-101, 158-167 .

Landi 66 Landin, P. J. The next 700 programming languages. Comm. ACM 9 (March 1966), 157-166.

Landw 64 Landweber, P. S. Decision problems of phrase structure grammars. IEEE Trans. Electronic Computers 13 (Aug. 1964), 354-362.

Lang 64 Langmaack, H. and Eickel, J. Präzisierung der begriffe phrasenstruktur und strukturelle mehrdeutigkeit in Chomsky-sprachen. Rep. no. 614, Rechenzentrum der Tech. Hoch. München (1964).

Lea 64 Leavenworth, B. M. FORTRAN IV as a syntax language. Comm. ACM 7 (Feb. 1964), 72-80.

Leo 66 Leonard, G. and Goodroe, J. More extensible machines. Comm. ACM 9,3 (March 1966), 190-195.

Let 65 Letichevskii, A. A. The representation of context-free languages in automata with a push-down type store. Cybernetics (Kibernetika). Vol. 1, no. 2, The Faraday Press, New York (1965), 81-86.

Luc 65 Lucas, P. Definition of a subset of PL/1 by finite local state vectors. Working Paper to IFIP WG2.1 (July, 1965).

Mas 60 Masterson, K. S. Compilation for two computers with NELIAC. Comm. ACM 3 (Nov. 1960), 607-611.

Mar 67 Martin, D. and Estrin, G. Models of computations and systems. J. ACM 14 (April 1967), 281-294.

Maur 65 Maurer, W. A theory of computer instructions. Memorandum MAC-M-262, Project MAC, MIT (Sept. 1965).

McCar 62a McCarthy, J. Towards a science of computation. C. N. Popplewell (Ed,), Information Procession, IFIP Conf. Munich (1962), 21-28.

McCar 62b McCarthy, J. et al. LISP 1.5 programmers manual. Computation Lab Report, MIT (1962).

McCar 67 McCarthy, J. and Painter, J. Correctness of a compiler for arithmetic expressions. AMS Symposium in Appl. Math. 19 (1967).

McCl 65 McClure, R. M. TMG--a syntax-directed compiler., Proc. 20th Natl. ACM Conf. (1965), 262-274.

McIl 60 McIlroy, M. D. Macro instruction extension of compiler language. Comm. ACM 3 (April 1960), 214-220.

McKee 66 McKeeman, W. M. An approach to computer language design. Tech. Rpt. CS 48, Computer Science Dept., Stanford University (Aug. 1966).

Mea 63 Mealy, G. A generalized assembly system. Rand Mem. RM-3646-PR (Aug. 1963).

Met 64 Metcalfe, H. H. A parametrized compiler based on mechanical linguistics. Annual Review in Automatic Programming 4 (1964), X5-165.

Mond 67 Mondschein, L. VITAL compiler-compiler reference manual. TN 1967 -1, Lincoln Laboratory (Jan. 1967).

Moo 65 Mooers, C. and Deutsch, L. P. TRAC, a text handling language. Proc. 20th Natl. ACM Conf., Cleveland, Ohio (1965).

Nap 67 Napper, R. B. E. The third-order compiler. A context for free man-machine communication. Machine Intelligence I (ed. by Collins, Michie) Oliver and Boyd (1967).

Nar 66 Narasimhan, R. Syntax-directed interpretation of classes of pictures. Comm. ACM 9 (March 1966), 166-173.

Naur 60 Naur, P. (Ed.) Report on the algorithmic language ALGOL 60. Num. Math. 2 (1960), 106-136; Comm. ACM 3 (May 1960), 299-314.

Naur 63a Naur, P. Documentation problems: ALGOL 60. Comm. ACM 6 (March 1963), 77-79.

Naur 63b Naur, P. Revised report on the algorithmic language ALGOL 60. Comm. ACM 6 (Jan. 1963), 1-17; Num. Math. 4 (1963), 420-452; Comp. J. 5 (1963), 349-367.

Nob 63 Noble, A. S. and Talmadge, R. B. Design of an integrated programming and operating system, I and II. IBM Syst. J. 2 (June 1963), 152-181.

op 62 Opler, A. 'Tool' --a processor construction language. Proc. IFIP (1962), 513-514.

Orch 66 Orchard-Hays, William. Multilevel operating systems. Comm. ACM 9 (March 1966), 189-190. (Abstract only).

org 66 Orgass, R. J. A mathematical theory of computing machine structure and programming. Unpublished doctoral thesis. Yale (1966).

Par 61 Parikh, R. J. Language generating devices Quarterly progress report no. 60, Research Laboratory of Electronics, MIT, (Jan. 1961), 199-212. Reprinted with minor editorial revisions under the title: On context-free languages. J. ACM 13 (Oct. 1966), 570-581.

Paul 62 Paul, M. ALGOL 60 processors and a processor generator. Proc. IFIP (1962), 493-497.

Plas 66 Plaskow, J. and Schuman, S. The Trangen system on the M460 computer. AFCRL-66-516 (July, 1966).

Pra 65 Pratt, T. W. Syntax-directed translation for experimental programming languages. TNN-41, University of Texas Computation Center (1965).

Pra 66 Pratt, T. W. and Lindsay, R. K. A processor-building system for experimental programming language. Proc. AFIPS FJCC (1966), 613-621.

Rab 62 Rabinowitz, I. N. Report on the algorithmic language FORTRAN II. Comm. ACM 5 (June 1962), 327-337.

Ran 64 Randell, B. and Russel, D. J. 'ALGOL 60 Implementation. Academic Press, London, 1964.

Rey 65 Reynolds, J. C. An introduction to the cogent programming system. Proc. 20th Natl. ACM Conf. (1965), 422-436.

Ri 66 Rice, J. and Rosen, S. NAPSS, Numerical analysis and problem solving system. Proc. ACM 21 Natl. Conf., Los Angeles (Aug. 1966), 51-56.

Rig 62 Riguet, J. Programmation et theories des categories. Proc. Rome Symposium on Symbolic Languages in Data Processing, Gordon and Breach, New York (1962), 83-98.

Rob 66 Roberts, L. G. A graphical service system with variable syntax. Comm. ACM 9 (March 1966), 173-176.

Ros 64a Rosen, S. A compiler-building system developed by Brooker and Morris. Comm. ACM 7 (July 1964).

Ros 64b Rosen, S. Programming systems and languages. Proc. SJCC, Washington, D. C., (1964), 1-15.

Ross 63 Ross, D. and Rodriguez, J. Theoretical foundations of the computer aided design system. Proc. SJCC (1963), 305-322.

Ross 64 Ross, D. T. On context and ambiguity in parsing. Comm. ACM 7 (Feb. 1964), 131-133.

Ross 66 Ross, D. T. AED bibliography. Memorandum MAC--278-2, Project MAC, MIT (Sept. 1966).

Rov 67 Rovner, P. and Feldman, J. An associative processing system for conventional digital computers. TN 1967-19, Lincoln Laboratory (April 1967).

Rut 62 Rutishauser, H. Panel on techniques for processor construction Proc. IFIP (1962), 524-531.

sam 60 Samelson, K. and Bauer, F. L. Sequential formula translation. Comm. ACM 3 (Feb. 1960), 76-83.

Sam 62 Samelson, K. Programming languages and their processing. Proc. IFIP (1962), 487-492.

samm 61 Sammet, J. E. A definition of COBOL 61. Proc. Natl. ACM Conf., Los Angeles (1961).

Sch 64 Schneider, F. W. and Johnson, G. D. Meta-3; A syntax-directed compiler writing compiler to generate efficient code. P.Dl.5-1, Proc. 19th Annual ACM Conf. (1964).

Schl 67 Schlesinger, S. and Sashkin, L. POSE: a language for posing problems to a computer. Comm. ACM 10 (May 1967), 279-285.

Scho 65 Schorr, H. Analytic differentiation using a syntax directed compiler. Comp. J. 7 (Jan. 1965), 290-298.

Schor 64 Schorre, D. V. Meta II: A syntax-oriented compiler writing language. P.Dl.3., Proc. 19th Natl. ACM Conf. (1964).

Schü 63 Schützenberger, M. P. Context-free languages and push-down automata, Information and Control 6 (Sept. 1963), 246-264.

Sh 58 Share Ad-Hoc Committee on Universal Languages, The problem of programming communication with changing machines: a proposed solution., Comm. ACM 1 (Aug. 1958), 12-18.

Shaw 63 Shaw, C. J. A specification of Jovial. Comm. ACM 6 (Dec. 1963), 721.

Shed 67 Shedler, G. Parallel numerical methods for the solution of equations. Comm. ACM lo (May 1967), 286-291.

Sib 61 Sibley, R. A. The SLANG-system. Comm. ACM 4 (Jan. 1961), 75-84.

Ste 61 Steel, T. B. A first version of UNCOL. Proc. AFIPS Western Joint Comput. Conf-. (1961), 371-378.

Ste 66 Steel, T. B. (Ed.) Formal language description languages for computer programming. (Proc. IFIP Conf., Baden, Sept. 1964), North-Holland Publishing Co., 1966.

Sto 67 Stone, H. S. One-pass compilation of arithmetic expressions for parallel processor. Comm. ACM 10 (April 1967), 220-223.

Str 65 Strachey, C. A general purpose macrogenerator, Comp. J. 8 (1965), 225-241.

Tar 56 Tarski, A. Logic, Semantics, Metamathematics. Clarendon Press, London, 1956.

Tay 61 Taylor, W., Turner, L. and Waychoff, R. A syntactical chart of ALGOL 60. Comm. ACM 14 (Sept. 1961), 393.

Te 66 Teichroew, D. and Lubin, J. F. Computer simulation-discussion of the technique and comparison of languages. Comm. ACM 9 (Oct. 1966), 727-741.

Th 66 Thompson, F. B. English for the computer. Proc. AFIPS FJCC (1966), 349-356.

Tix 67 Tixier, Jr. Recursive functions of regular expressions in language analysis Tech. Rpt. CS 58, Computer Science Dept., Stanford University (March 1967).

war 61 Warshall, S. A syntax directed generator Proc AFIPS Eastern Joint Comput. Conf. (1961), 295-305.

War 64 Warshall, S., and Shapiro, R. M. A general-purpose table-driven compiler. Proc. AFIPS SJCC (1964), 59-65.

Weg 62 Wegner, P. (Ed). Introduction to Systems Programming. Academic Press, New York, 1962.

Wij 66 van Wijngaarden, A. Recursive definition of syntax and semantics in Formal Language Description Languages for Computer Programming, North-Holland Publishing Co. , 1966, 13-24.

Wil 64a Wilkes, M. V. An experiment with a self-compiling compiler for a simple list-processing language. Annual Review in Automatic Programming 4 (1964), 1-48.

Wil 64b Wilkes, M. V. Constraint-type statements in programming languages. Comm. ACM 7 (Oct. 1964), 587.

Wir 66a Wirth, N. A programming language for the 360 computers. Tech. Rpt. CS 53, Computer Science Dept., Stanford University (Dec. 1966).

Wir 66b Wirth, N. and Hoare, C.A.R. A contribution to the development of ALGOL. Comm. ACM 9 (June, 1966), 413-432.

Wir 66c Wirth, N. and Weber, H. EULER - a generalization of ALGOL, and its formal definition: Part I, Part II. Comm. ACM 9 (Jan.-Feb. 1966), 13-25, 89-99.

Yer 65 Yershov, A. P. ALPHA--an automatic programming system of high efficiency. IFIP Congr., New York (1965).

you 65 Young Jr., J. W. Non-procedural languages. 7th Ann. ACM Tech. Symp., Southern Calif. Chapter (March 1965).

Zem 66 Zemanek, H. Semiotics and programming languages. Comm. ACM 9 (March 1966), 139-143.