Programming Languages
N. Wirth, Editor

Translator Writing Systems

By Jerome Feldman and David Gries
Stanford University, Stanford, California

An Exploration of Concepts and Principles

A critical review of recent efforts to automate the writing of translators of programming languages is presented. The formal study of syntax and its application to translator writing are discussed in Section II. Various approaches to automating the postsyntactic (semantic) aspects of translator writing are discussed in Section III, and several related topics in Section IV.

KEY WORDS AND PHRASES: compiler, compiler-compiler, translator, translator writing systems, metacompiler, syntax, semantics, syntax-directed, meta-assembler, macroprocessor, parser, syntactic analysis, generator
CR CATEGORIES: 4.1, 4.10, 4.12, 4.22, 5.23

CONTENTS

I. INTRODUCTION

> ". . . for all of it is contained in a long poem which neither I, nor anyone else, has ever succeeded in wading through."
> So speaks The Devil in Shaw's *Man and Superman*.

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 schemes for automating different parts of the compiler writer's task. This paper is an attempt to critically survey these research efforts. An early version of this survey, Stanford Computer Science Report CS69, June 1967, was circulated widely, and the many thoughtful comments we received have made an inestimable contribution to the accuracy and conceptual clarity of the present paper.

Before we describe the particular systems, we say a few things about the general problem of translator writing. We 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. This lack of literature has forced translator writing system (TWS) designers to try to formalize techniques which have never been described carefully. A further difficulty is that there are no accepted standards of

This work was partially supported by the US Atomic Energy Commission and by the Advanced Research Projects Agency.

performance for translators, only shibboleths, such as efficiency. The efficiency of a compiler depends on its ability to conserve both time and space while translator and during execution of the object program. The ease of use, the error detection and recovery facilities, the editing facilities, and the speed of recompilation have important effects on efficiency. As not all these goals are mutually compatible, one can expect no absolute measure of performance for compilers. The designers of the TWSs considered here have varied greatly in their preferred choice of compromises.

Since compiler writing is a large programming task with many aspects, 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 is as yet no general agreement on terminology, we here 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.

This paper contains neither a history of nor an introduction to the work on TWSs; the references at the end of this section provide what introductory material there is in the literature. Although we compare individual systems and also various techniques, this paper is not intended to be a consumer's guide to translator writing systems. The intended purpose is to consider the existing work carefully in an attempt to form a unified scientific basis for future research. Toward this end we emphasize the intellectual content of the various TWS designs, rather than the system features available in a particular implementation.

The use of TWSs to write commercial compilers is just now becoming common. This lag of about three years is not excessive, but it has led some people to disregard the entire TWS development. While it is true that any particular TWS is more suited to certain compiler characteristics, this does not seem to be the major bar to their use. The successful TWSs have all been done in a research environment by people who have not shown an entrepreneurial bent. Most importantly, the idea of flexible languages, inherent in TWS work, runs counter to the manufacturer's emphasis on ever greater standardization. Although commercial compiler writers are starting to use TWSs, it will take a minor revolution to put TWSs in the hands of the user where they belong.

Unfortunately, one has to exercise considerable care in reading the TWS literature. A system described formally in a paper is rarely adequate to completely handle the applications claimed for it. There is also a strong current of mathematism, that is, the notion that the use of symbolic notation automatically increases the value of a paper. Communication between various workers seems to be poor; there is much rediscovery and little cross-referencing within the field. The existence—and tolerance by referees—of the situation mars an otherwise excellent record in TWS research.

Our review of TWSs is divided into two major headings, syntax and semantics. The work on automated syntax methods is the oldest and best understood aspect of TWS research. Syntax methods are further divided into those of limited generality, which have been used in TWSs, Section II.B, and the theoretically more powerful but as yet inapplicable methods of Section II.C.

The division of semantic considerations into three sections is along somewhat more controversial lines. The syntax-directed symbol processors of Section III.A share the approach of considering translators as a special case of a problem which is best treated generally. The compiler-compilers of III.B attempt to provide many specific mechanisms to help in the postsyntactic processing of programs. Section III.C considers two related sets of attempts to extend the conventional macro assembler to a TWS.

The related topics discussed in Section IV were 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, a number of potentially fruitful research topics related to the future development of translator writing systems are sketched. The bibliography is arranged alphabetically; in addition, references pertinent to each section are indicated at the end of the section.

REFERENCES FOR I

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), 7 (Feb. 64), and 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, BroS 63, EvA 64, Gor 61, IBM 66, Naur 60, 63b, Rab 62, Samm 61, Shaw 63, Tay 61, Wir 66b, 66c.

II. SYNTAX

A. Terminology

One of the minor irritants in 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, pp. 8, 9]. However, as a nonconflicting alternative, the notation of the syntactic metalanguage Backus-Naur Form (BNF), especially the symbol "::=", is used where it is more readable.

In this paper many words are used in both the formal and the informal sense; in this section on syntax the usual sense is the formal, while in sections III and IV, the informal. The formal definitions of such terms as "syntax" and "semantics" are not generally agreed upon, and we

⟨program⟩ → ⊥ E ⊥	⟨program⟩ ::= ⊥ E ⊥
$E \rightarrow T$	$E ::= T \mid E + T$
$E \rightarrow E + T$	$T ::= P \mid T * P$
$T \rightarrow P$	$P ::= I \mid (E)$
$T \rightarrow T * P$	
$P \rightarrow (E)$	
$P \rightarrow I$	
(a) formal notation	(b) BNF notation

Nonterminal symbols: ⟨program⟩ E T P.
Terminal symbols: I ( ) + * ⊥.

FIG. 1. Example of a phrase structure grammar

discuss them further in Section IV. Informally, we consider syntax to be a specification of the well-formed statements of a language, usually incorporating a mechanism for structural descriptions, and semantics to be a specification of how these statements are to be executed by a real or abstract computer.

In general throughout the paper, a *language* $\mathfrak{L}$ is some subset of the set $\mathfrak{A}^*$ of all finite strings of symbols from an *alphabet* $\mathfrak{A}$. The specification of which strings are in the language $\mathfrak{L}$ (syntax of $\mathfrak{L}$) is described in a *syntactic metalanguage*. The syntactic metalanguage is procedural and describes either an algorithm for *generating* strings of $\mathfrak{L}$ (*synthetic* syntax) or for *recognizing* if an element of $\mathfrak{A}^*$ is in $\mathfrak{L}$ (*analytic* syntax). Statements in a syntactic metalanguage are often called *productions*. Any process utilizing a nontrivial analytic syntax is called *syntax-directed*.

The symbols in the alphabet $\mathfrak{A}$ are called *terminal symbols*, and in Section II are denoted by T, T_1, T_2, etc. A syntactic metalanguage may include a set of *nonterminal symbols*, η, not in $\mathfrak{A}$, which are used in defining the language. These nonterminals are normally enclosed in angle brackets "⟨" and "⟩", as in the ALGOL report, and appear in the text as well as in formal syntax rules.

In this section where we deal more formally with syntax, we omit the brackets and *represent all nonterminals by Latin capitals* U, V, and Z. These sections on syntax also require a fairly extensive technical vocabulary, which we now describe in detail.

The *vocabulary* $\mathfrak{V}$ is defined as the union of $\mathfrak{A}$ (the set of terminal symbols) and η (the nonterminal symbols). The symbols S, S_1, S_2, etc. are used to denote members of $\mathfrak{V}$, while strings of symbols (including the empty string Λ) are denoted by lowercase Latin letters u, v, w, $\cdots$. If $z = xy$ is a string, x is a *head* and y a *tail* of z.

We specify a language, $\mathfrak{L}_\mathfrak{G}$, by a *phrase structure grammar*, $\mathfrak{G}$, which is defined as a finite set of *productions* of the form $U_i \rightarrow u_i$ with the following properties:

(1) each u_i is a nonempty string whose symbols are in the vocabulary $\mathfrak{V}$;
(2) each U_i is a nonterminal symbol: $U_i \in \eta$;
(3) There is exactly one U_i, called the *distinguished symbol* Z, which occurs in no u_i.

U_i is called the *left part* and u_i the *right part* of the production $U_i \rightarrow u_i$. Figure 1(a) is an example of a grammar, with Z = ⟨program⟩. Figure 1(b) gives an alternate notation for the same phrase structure grammar, the Backus-

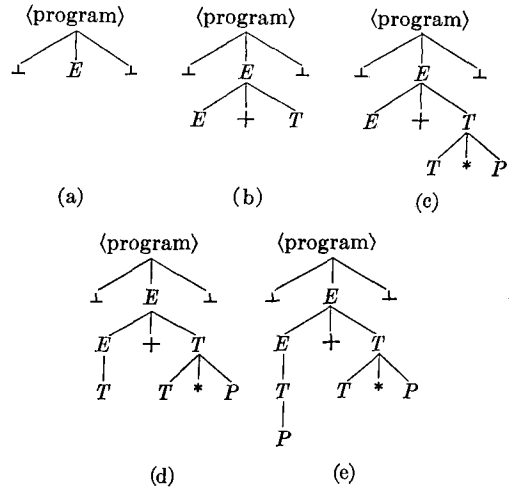


FIG. 2. Syntax trees

Naur Form (BNF). Here "::=" is substituted for "→" and the metasymbol "|" is used to separate different right parts corresponding to the same left part.

In some of the work reviewed, productions with empty right parts are allowed, though we may not always mention it.

In order to show how a (phrase structure) grammar defines a language, we need some further definitions. We say that v is a *direct derivative* of w (written $w \Rightarrow v$) *by application of the production* $U \rightarrow u$ if there are (possibly empty) strings x and y such that $w = xUy$ and $v = xuy$.

The transitive closure of "⇒" is denoted by "$\overset{*}{\Rightarrow}$"; $w \overset{*}{\Rightarrow} v$ if there exist strings $w_0, w_1, \cdots, w_i$ $(i \geq 0)$ such that $w = w_0$, $w_0 \Rightarrow w_1$, $\cdots$, $w_{i-1} \Rightarrow w_i$ and $w_i = v$. v is then called a *derivative* of w, and the sequence $w = w_0 \Rightarrow w_1, \cdots, w_{i-1} \Rightarrow w_i = v$ a derivation of v from w.

The derivatives of the distinguished symbol Z are called *sentential forms*. The language $\mathfrak{L}_\mathfrak{G}$ is defined as the set of *sentences*, i.e. the set of sentential forms consisting only of terminal symbols:

$$\mathfrak{L}_\mathfrak{G} := \{x \mid Z \overset{*}{\Rightarrow} x \quad \text{and} \quad x \in \mathfrak{A}^*\}$$

In the grammar $\mathfrak{G}$ of Figure 1, $\mathfrak{L}_\mathfrak{G}$ is the set of all arithmetic expressions (using operators + and *, parentheses (and) and the operand I). The beginning and end of the arithmetic expressions are explicitly indicated by the symbol ⊥.

The sentential form $\perp P + T * P \perp$ has at least two derivations (according to the grammar of Figure 1):

$$\begin{aligned} \langle\text{program}\rangle &\Rightarrow \perp E \perp \Rightarrow \perp E + T \perp \Rightarrow \perp T + T \perp \\ &\Rightarrow \perp P + T \perp \Rightarrow \perp P + T * P \perp \end{aligned} \tag{2.1}$$

$$\begin{aligned} \langle\text{program}\rangle &\Rightarrow \perp E \perp \Rightarrow \perp E + T \perp \Rightarrow \perp E + T * P \perp \\ &\Rightarrow \perp T + T * P \perp \Rightarrow \perp P + T * P \perp \end{aligned} \tag{2.2}$$

A derivation may be illustrated by a *syntax tree*, which is drawn for derivation (2.2) as follows: Starting from the symbol ⟨program⟩ a *branch* is drawn, as in Figure 2(a). The branch is the set of lines emanating downward from

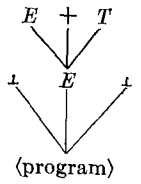


FIG. 3. Upside-down syntax tree

the *node* ⟨program⟩, together with the *nodes* (the labels) at the ends of these lines. These nodes concatenated form the string $\perp E \perp$ which replaced ⟨program⟩ in the derivation. We continue in the same manner; for each application of a production $U ::= x$ (each direct derivation), from the node U, which is being replaced, we draw a branch whose nodes form the replacing string x. This is illustrated for derivation (2.2) by the sequence of syntax trees in Figure 2, with Figure 2(e) representing the complete derivation. Note that the *end nodes* of the tree (those with no branches emanating downward from them), when concatenated, yield the final sentential form.

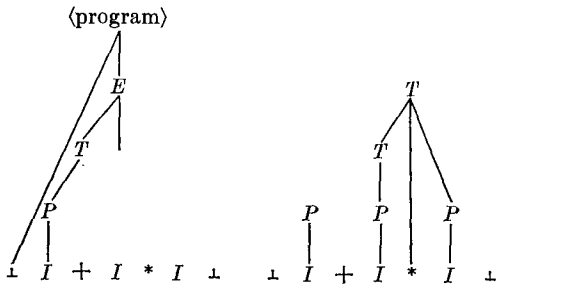
Although there are two derivations of $\perp P + T * P \perp$, both have the same syntax tree, since the derivations differ only in the *order* in which the productions are applied as direct derivations. A grammar $\mathcal{G}$ is said to be *ambiguous* if there is a sentence of $\mathcal{L}_\mathcal{G}$ which has more than one syntax tree relative to $\mathcal{G}$.

Given a synthetic phrase structure grammar one can randomly generate sentences of the language by deriving them and their syntax trees from the distinguished symbol Z. Compilers on the other hand have the opposite problem: given a sentence x and a grammar $\mathcal{G}$, construct a derivation of x and find a corresponding syntax tree. This is called *parsing, recognizing* (whence the term *recognizer*), or *analyzing* the sentence.

There are two different parsing strategies, called *top-down* and *bottom-up*, which are sometimes confused. One reason is that people draw syntax trees differently; the tree in Figure 2(b) (which is how we draw it) could also be drawn as in Figure 3. Another reason is that these two strategies have actually merged as recognizers become sophisticated. We discuss this later in this section and again in Section II.C.5. Both types of strategies are called *left-right*, since the general order of processing the symbols in the sentence is from left to right whenever possible.

A pure top-down recognizer is entirely *goal-oriented*. The main goal is, of course, the distinguished nonterminal symbol Z—a *prediction* is made that the string to be recognized is actually a sentence. Therefore the first step is to see whether the string can be reduced to the right part $S_1S_2\cdots S_n$ of some production $Z \rightarrow S_1S_2\cdots S_n$.

This is done as follows: For the application of the production to be valid, if S_1 is a terminal symbol, then the string must begin with this terminal. If S_1 is nonterminal, a subgoal is established and tried: see whether some head of the string may be reduced to S_1. If this proves true,



(a) partial top-down parse (b) partial bottom-up parse

FIG. 4

S_2 is tested in the same manner, then S_3, and so on. If no match can be found for some S_i, then application of an alternate production $Z \rightarrow S_1'S_2'\cdots S_m'$ is attempted.

Subgoals U are tested in the same manner; a production $U \rightarrow S_1S_2\cdots S_n$ is tested to see whether it can be applied. Thus, new subgoals are continually being generated and attempted. If a subgoal is not met, failure is reported to the next higher level, which must try another alternative.

Left recursion sometimes causes trouble in left-right top-down recognizers; productions of the form $U_1 ::= U_1x$ may cause infinite loops. This is because when U_1 becomes the new subgoal, the first step is to create again a new subgoal U_1. The left recursion problem is sometimes solved by changing the grammar or modifying the recognizer in some way (see below). The order in which the different right parts of productions for the same left part are tested can also play a large role here. If there is only one right part which contains left recursion this one should be the last one tested. However, this might conflict with other ordering problems, such as testing the shortest right part last.

The top-down 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 incorrectly) by constructing the tree from the top node and reaching down to the string (Figure 4(a)). Such recognizers are sometimes called *predictive* [see Kun 62], since at each step they try to predict the connections to be made.

A top-down recognizer may 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 long-range goals (except of course the implicit goal Z). The string is searched for substrings which are right parts of productions. These are then replaced by the corresponding left side. This is illustrated by Figure 4(b). We will go into some detail here in order to introduce terminology needed in later sections.

Although the syntax tree is not present when we start to parse a sentence, it is clearer to present the idea behind bottom-up parsing as if it were. Let us therefore suppose we have a sentential form s and its syntax tree. We define a *phrase* of s to be the set of end nodes of some subtree of the syntax tree. We now define the *handle* of s relative to the syntax tree to be the *leftmost phrase* which contains no

phrases other than itself. For example, in the syntax tree of Figure 2(e) there are four phrases: P, $T * P$, $P + T * P$, and $\perp P + T * P \perp$. Two of these contain no other phrases: P and $T * P$. The handle is the leftmost such phrase: P.

The following algorithm represents the general philosophy behind left-right bottom-up parsing: starting with a sentential form $s = s_0$ (and a syntax tree for it), repeat the following steps for $i = 0, 1, \cdots, n$ until $s_n = Z$ has been produced:

1. Find the handle of s_i (by looking at the syntax tree for s_i).
2. Replace the handle of s_i by the label on the node naming the corresponding branch, yielding the sentential form s_{i+1}.
3. *Prune* the tree by deleting the handle from it.

The sequence $Z = s_n \Rightarrow s_{n-1} \Rightarrow \cdots \Rightarrow s_1 \Rightarrow s_0$ is then a derivation of s_0. For example, given the sentential form $s_0 = \perp P + T * P \perp$ and its syntax tree in Figure 2(e), the handle P is replaced by T and pruned from the tree, yielding $s_1 = \perp T + T * P \perp$ and the syntax tree of Figure 2(d). The handles for the syntax trees in Figures 2(d), 2(c), 2(b), and 2(a) are, respectively, T, $T * P$, $E + T$, and $\perp E \perp$. The syntax tree of Figure 2(c) arises by pruning the handle of 2(d), 2(b) arises by pruning the handle of 2(c), and 2(a) comes similarly from 2(b). This sequence yields the derivation (2.2).

The pure bottom-up recognizer, like the pure top-down recognizer, will normally make reductions (or connections) which turn out to be incorrect. This may be handled in one of two different ways. The first method is to back up (*backup* or *backtracking*) to a point where another alternative may be tried. This involves restoring parts of the string to a previous form or erasing some of the connections made. The second method is to carry out all possible parses in parallel. As some of them lead to "dead ends" (no more reductions or connections are possible), they are dropped. COGENT (Section III.A.3) uses this method in a top-down scheme. See also [Kun 62].

In order to reduce the probability of making incorrect reductions, more sophisticated recognizers have been developed. For instance, before starting out on a new subgoal, a *modified top-down recognizer* might look in a preconstructed table to see whether some derivative of the subgoal can actually start with the initial symbol of the substring in question (*look ahead*), or whether the subgoal being attempted could occur in the partial tree formed so far (*memory*). Examples of modified top-down recognizers are those in [Ir 61, May 61, and War 64]. Most of the syntax-directed symbol processors of Section III.A use modified top-down recognizers.

Similarly, modified bottom-up recognizers look at the context around a possible handle to aid in the decision. In practice, these recognizers have become sophisticated enough so that, with certain restrictions on the grammar, backup or parallelism is unnecessary.

These modifications have contributed to the (con)fusion of the two concepts. It is sometimes very difficult to classify a particular recognizer as bottom-up or top-down. For instance, a production language recognizer as generated by Earley's algorithm (cf. Section II.B.5) 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 goal-oriented. See Section III.C.5 for a further discussion on this problem.

Most of the remaining terminology should be familiar to anyone with general knowledge of computer science. We use a few data structure terms which require definition. The term *list structure* system is 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 and where each element is a block of storage containing several (often two-way) links is called a *plex*. We also use the terms LIFO (last-in-first-out) and FIFO (first-in-first-out) as general rules for handling sequential information.

REFERENCES FOR II. A
Che 64c, Chom 63, Flo 64b, Gin 66a,Ir 61, Kun 62, Naur 60, War 64.

B. Automatically Constructed Recognizers

In this section several practical techniques for parsing, or for recognizing, sentences of languages defined by grammars are described and evaluated. A "practical" technique is one that has been or is being used to write a compiler. Such a recognizer may be in the form of tables to be used by a set of basic routines or in the form of a program in some language. Each of the recognizers described here has an associated algorithm, called a *constructor*, for generating it from a suitable grammar. Finally, all are left-right recognizers which work with one LIFO stack and have no backup facilities.

The property of automatic generation is very important to the compiler writer. Most of the constructors check the grammar for ambiguity 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 guarantees that the recognizer follows the formal syntax.

Unfortunately, these recognizers and their constructors do not solve all problems. First, the existing formal notions of syntax cannot be used to completely describe the syntax of most programming languages. Second, 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. Third, while a technique may be theoretically sound, the restrictions necessary for its use may require substantial alteration of the conventional grammar of the programming language.

We note in passing that the "efficiency" of several recognizers has been compared by Griffiths and Petrick

[Grif 65]. Although it is of some theoretical interest, this comparison is of no practical value in writing compilers, since it is based mainly on the efficiency of Turing machines corresponding to each of the recognizers. We are interested in comparing the recognizers with respect to the following points:

(a) How much space does the recognizer use?
(b) How fast is the recognizer?
(c) How much does a conventional grammar have to be altered in order to be accepted by the constructor?
(d) Once the recognizer is constructed, how easy is it to insert semantics and the syntactic properties not expressed by the grammar?

The reader must note that our comparisons are based on these recognizers as formally applied and that they are general observations; by bit-pushing or devising fast list searching techniques, a particular implementation can greatly increase efficiency. The above properties also depend on the type of compiler being built, the language in question, and so forth. The answer to question (d) in particular depends very much on whether some internal form of the source program or machine code itself is to be generated and on the power of the semantic processes available in the compiler.

Although it is possible to build bottom-up recognizers which allow backup, this has rarely been done. Restrictions are usually placed on the grammar to assure its unambiguity and to assure that the unique handle of any sentential form can be efficiently detected and reduced. All of the recognizers discussed here do this. On the other hand, many of the top-down recognizers in use today allow backup, or they carry out possible parses in parallel; the only restriction is that left recursion is not allowed (see p. 80). The existing top-down recognizers therefore accept a wider class of grammars but tend to be less efficient; backup can lead to very inefficient recognizers if the grammar is not written cleverly.

Pure top-down recognizers were discussed briefly in Section II.A, and therefore are not discussed here. See [War 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, and [Flo 64b] also contains a good description of the technique.

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 give references to papers where both the recognizer and its constructor are discussed. Some theoretically interesting recognizers which can be mechanically constructed, as well as formal properties of systems described here, are discussed briefly in Section II.C.

The grammar in Figure 1 (p. 79) is used throughout this section as an example. At this point it may be advisable for the reader to briefly review Section II.A for definitions and notations.

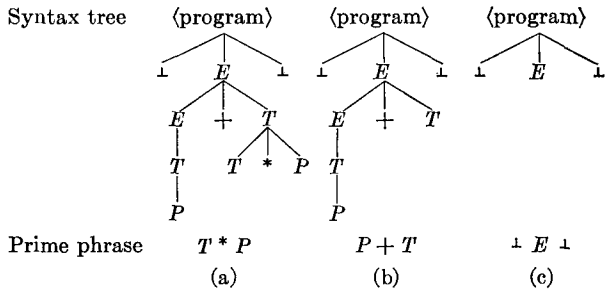


FIG. 5. Parse using operator precedence

B.1. OPERATOR PRECEDENCE (Floyd [Flo 63])

The grammar is first of all 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, and most programming language grammars not in this form can be made into operator grammars without essentially disturbing the structure of the language.

Given an operator grammar, let s be a sentential form. We define a *prime phrase* of s to be a phrase which contains no phrase other than itself but at least one terminal character T. (Compare this with the definitions of phrase and handle on p. 80.) For instance, in Figure 5(a) the phrases are P, $T * P$, $P + T * P$ and $\perp P + T * P \perp$; the prime phrase is $T * P$. Similarly in Figure 5(b) the prime phrase is $P + T$, in 5(c) $\perp E \perp$. The recognizer to be described reduces at each step the *leftmost prime phrase*, and not the handle. However, we still call this a bottom-up, left-right recognizer, since it is proceeding essentially in a left to right manner.

Equivalently, x is a prime phrase of at least one sentential form s if and only if x contains at least one terminal and either there exists a production $U \rightarrow x$ or a production $U \rightarrow x'$ where $x' \overset{*}{\Rightarrow} x$ and the only productions applied in the derivation $x' \overset{*}{\Rightarrow} x$ are of the form $U_i \rightarrow U_j$.

During the parse of a sentence $T_1 \cdots T_m$, a LIFO stack will contain symbols $S_0S_1 \cdots S_i$ of the partially reduced string $S_0S_1 \cdots S_iT_jT_{j+1} \cdots T_m$. 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 symbol of the leftmost prime phrase in the stack; or whether (2) S_i is *not* the tail 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 $U \rightarrow xT_1T_2y$ or $U \rightarrow xT_1U_1T_2y$ where U_1 is nonterminal.
2. $T_1 \gtrdot T_2$ if there is a production $U \rightarrow xU_1T_2y$ and a derivation $U_1 \overset{*}{\Rightarrow} zT_1$ or $U_1 \overset{*}{\Rightarrow} zT_1U_2$ for some z and U_2.
3. $T_1 \lessdot T_2$ if there is a production $U \rightarrow xT_1U_1y$ and a derivation $U_1 \overset{*}{\Rightarrow} T_2z$ or $U_1 \overset{*}{\Rightarrow} U_2T_2z$ for some z and U_2.

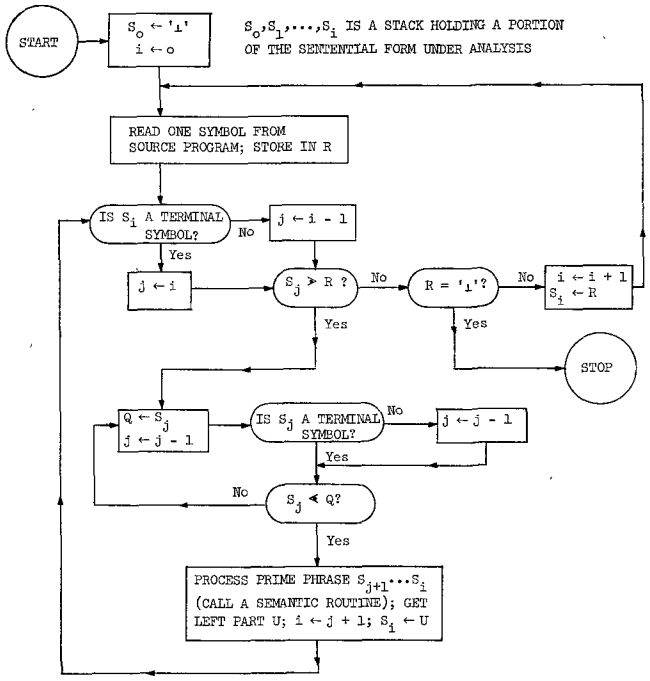


FIG. 6. Recognizer using operator precedences

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 substring which may be reduced (prime phrase). Suppose T_0xT is a substring of a sentential form $s = x_1T_0xTx_2$ and that the terminal symbols in the substring x are, in order, T_1, T_2, $\cdots$, T_n $(n \geq 1)$. Now suppose the following relations hold between T_0, T_1, $\cdots$, T_n and T:

$$T_0 \lessdot T_1 \doteq T_2 \doteq \cdots \doteq T_n \gtrdot T.$$

(Note that nonterminals of x play no role here.) Then x is a prime phrase. Furthermore the reduction of x to some U may always be executed to yield the sentential form $x_1T_0UTx_2$.

The parse of a sentence (or program) is quite straightforward (see Figure 6). 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 string is indeed a sentence of the language, the top stack elements then hold a string T_0x as described above. One searches back in the stack, using the relations, to find T_0 and the beginning of x. x is then a prime phrase and can then be reduced to some U, yielding T_0U in the stack. The process is then repeated by comparing T_0 with T.

As an example, the sentential form $\perp P + T * P \perp$ would be parsed (using the grammar of Figure 1) as illustrated in Figure 5, where each tree is derived by pruning the prime phrase of the preceding syntax tree.

TABLE I

T_1 \ T_2	(	I	*	+	⊥	)
)			⩾	⩾	⩾	⩾
I			⩾	⩾	⩾	⩾
*	⩽	⩽	⩾	⩾	⩾	⩾
+	⩽	⩽	⩽	⩾	⩾	⩾
(	⩽	⩽	⩽	⩽		≐
⊥	⩽	⩽	⩽	⩽	≐	

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

The relations $\gtrdot$, $\doteq$ and $\lessdot$ can be kept in an $l \times l$ 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 × 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 integer *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 $f(T_1) > g(T_2)$.

Floyd outlines the algorithm for finding the matrix of precedence relations, and an algorithm which finds the functions f and g if and only if they exist. For the language of Figure 1 the precedence matrix and functions in Table I are generated.

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 prime phrase 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. (Some compilers actually parse the program twice. The first parse makes a complete syntax check and also allows one to collect global information about variables, blocks, etc. The second parse uses the efficient operator precedence technique—the functions—and the information collected during the first parse to generate code. The trend is, however, to let the syntax checker produce an altered form of the source program—reverse polish, triples, etc. (see Section III.B.2)—from which code may be generated more easily, making a second parse unnecessary.)

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 an operator precedence grammar, but the names of the nodes may not be unambiguous. For a prime phrase x there may exist more than one nonterminal to which it may be reduced, since there is no restriction that right parts of productions be unique. 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.

A semantic routine is called when a prime phrase is recognized and is to be reduced. A separate routine is written to process each different prime phrase. This sometimes requires an alteration of the grammar, depending of course on the semantic processing to be carried out. For instance, the production

⟨cond⟩ → **if** ⟨be⟩ **then** ⟨expr⟩ **else** ⟨expr⟩

would normally be changed to

⟨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.

However, the revised grammar 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 itself 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.

B.2. PRECEDENCE (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 $U \rightarrow xS_1S_2y$.
2. $S_1 \gtrdot S_2$ if there is a production $U \rightarrow xU_1S_2y$ (or $U \rightarrow xU_1U_2y$) and a derivation $U_1 \overset{*}{\Rightarrow} zS_1$ (and $U_2 \overset{*}{\Rightarrow} S_2w$) for some z, w.
3. $S_1 \lessdot S_2$ if there is a production $U \rightarrow xS_1U_1y$ 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 no two productions have identical right parts, then the grammar is called a *precedence grammar* and the language a *precedence language*. Any sentence of a precedence language has a unique syntax tree. When parsing, 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

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

The handle is then $S_j \cdots S_i$ and is replaced by the left part U of the unique production $U ::= S_j \cdots S_i$. The main difference between this technique and Floyd's is that the relations may hold between any two symbols and not just between terminal symbols; therefore, the handle and not the prime phrase is reduced. Algorithms for generating the matrix of precedences and functions f and g similar to Floyd's are given in [Wir 66c].

TABLE II

S_1 \ S_2	E'	E	T'	T	P	(	I	*	+	)	⊥
E'										⊜	⊜
E									⊜	⊛	⊛
T'									⊛	⊛	⊛
T								⊜	⊛	⊛	⊛
P								⊛	⊛	⊛	⊛
)								⊛	⊛	⊛	⊛
I								⊛	⊛	⊛	⊛
*						⊜	⊘	⊘			
+				⊜	⊘	⊘	⊘	⊘			
(	⊜	⊘	⊘	⊘	⊘	⊘	⊘	⊘			
⊥	⊜	⊘	⊘	⊘	⊘	⊘	⊘	⊘			

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

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⟩ → $\perp E' \perp$
$E' \rightarrow E$
$E \rightarrow T'$
$E \rightarrow E + T'$
$T' \rightarrow T$
$T \rightarrow P$
$T \rightarrow T*P$
$P \rightarrow (E')$
$P \rightarrow I$

The precedence matrix and functions for this grammar are given in Table II. Actually, any phrase structure grammar can be modified, without doing violence to its phrase structure, such that there is at most one precedence relation between any two symbols. Ambiguities show up in nonunique right sides of productions [Michael Fisher, Harvard U.]. The problem of multiple right sides makes this rather unpractical, even if the grammar is unambiguous.

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 70×70 for ALGOL), since the relations may hold between any two symbols. Again, 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; in a precedence grammar one *knows* that a unique canonical parse exists for each sentence. In practice, however, the restriction to unique right parts is not followed; each semantic processor for a handle which is reducible in more than one way must determine the correct nonterminal to replace x from the context and global information. This is necessary for productions such as

⟨array identifier⟩ → ⟨identifier⟩
⟨procedure identifier⟩ → ⟨identifier⟩

TABLE III

MATRIX1

S_1 \ S_2	E	T	P	(	I	*	+	)	⊥
E							⋖ ⋗	⋖ ⋗	⋖ ⋗
T						⋖ ⋗	⋗	⋗	⋗
P						⋗	⋗	⋗	⋗
)						⋗	⋗	⋗	⋗
I						⋗	⋗	⋗	⋗
*			⋖	⋖	⋖				
+		⋖	⋖	⋖	⋖				
(	⋖	⋖	⋖	⋖	⋖				
⊥	⋖	⋖	⋖	⋖	⋖				

Function $P1$ not necessary, since the conflict ⋚ does not arise.

MATRIX2

S_1 \ S_2	E	T	P	(	I	*	+	)	⊥
E							⋗	⋗	⋗
T						⋗	⋗	⋗	⋗
P						⋗	⋗	⋗	⋗
)						⋗	⋗	⋗	⋗
I						⋗	⋗	⋗	⋗
*			⋖	⋖	⋖				
+		⋖	⋖	⋖	⋖				
(	⋖	⋖	⋖	⋖	⋖				
⊥	⋖	⋖	⋖	⋖	⋖				

Function $P2$ (Only necessary triples which also form valid substrings of some sentential form listed.)

$P2[\perp,E,+] = \text{TRUE}$
$P2[\perp,E,\perp] = \text{FALSE}$
$P2[(,E,+] = \text{TRUE}$
$P2[(,E,)] = \text{FALSE}$
$P2[+,T,*] = \text{TRUE}$
$P2[+,T,+] = \text{FALSE}$
$P2[+,T,)] = \text{FALSE}$
$P2[+,T,\perp] = \text{FALSE}$

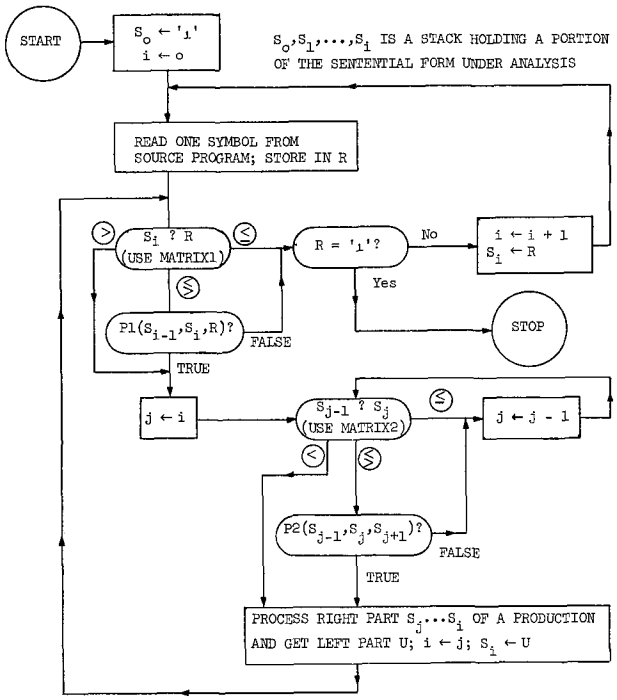


FIG. 7. Recognizer using Wirth precedences plus McKeeman triples

Moreover, 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. In the latter case 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.

B.3. EXTENDED PRECEDENCE (McKeeman [McKee 66])

McKeeman extended Wirth's concept by first separating the precedence matrix into two tables—one for looking for the tail of the handle, the other, for the head of a handle—and then 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 II.C.1. In practice, the number of different triples is too large (over 10,000 for a dialect of PL1). 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 ⋖ (⋖ or ≐), ⋗, and ⋚ (⋗ and either ≐ or ⋖) are used. If ⋚ holds between the top stack symbols S_i and the incoming symbol T then a list of triples is searched to find the value of the following three-argument function $P1$;

$$P1(S_{i-1}, S_i, T) := \begin{cases} \textbf{true} & S_i \gtrdot T \text{ (S_i is tail of a handle) in the context } S_{i-1}\, S_i\, T; \\ \textbf{false} & T_i \lessdot S \text{ holds in the context } S_{i-1} S_i T. \end{cases}$$

Of course this function must be single valued for all triples, and the constructor checks this. A similar matrix MATRIX2 with entries ⋗, ⋖ and ⋋ (⋖ and either ≐ or ⋗) and a function $P2$ are used when looking in the stack for the initial symbols of the handle:

$$P2(S_{j-1}, S_j, S_{j+1}) = \begin{cases} \textbf{true} & S_{j-1} \lessdot S_j \text{ (S_j is head of a handle) in the context } S_{j-1}\, S_j\, S_{j+1}; \\ \textbf{false} & S_{j-1} \gtrdot S_j \text{ holds in the context } S_{j-1}\, S_j\, S_{j+1}. \end{cases}$$

For the grammar of Figure 1, matrices and functions in Table III are generated; the recognizer is given in Figure 7.

The use of triples helps avoid most of the unpleasantness 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 dialect of PL1 on the IBM 360 using this technique. The matrices MATRIX1 and MATRIX2 are about 90 × 45 and 90 × 90 (each matrix element is two bits long), while roughly 450 triples are necessary. An alternative approach now being considered is to throw out the 90 × 90 matrix used to find the head of the handle. Then, when a handle is in the stack, all possible right parts will be compared with the stack contents to determine the correct production to apply.

B.4. Transition Matrices

(Samelson and Bauer [Sam, 60], Gries [Grie 67a])

This technique for parsing sentences was first introduced by Samelson and Bauer. In their version two stacks are used—an *operator* stack and an *operand* stack. The sentence is processed from left to right; incoming identifiers are pushed onto the operand stack, while the incoming operators (+, /, **begin**, etc.) are compared with the top operator on the operator stack. If a reduction cannot be performed, the incoming operator is pushed onto the operator stack. If a reduction can be performed, a subroutine is executed that performs some operation using the top operator stack element and the operands on the operand stack, deletes those elements used, and pushes a resulting operand onto the operand stack. Note the similarity with the operator precedence technique; two terminals (operators) are used to determine the process to be performed—nonterminals (operands) play no role in this. The extra operand stack is used just to make it easy to reference the terminals and nonterminals separately and is a practical, not a theoretical, consideration. The only real difference is that, while the operator precedence technique uses a matrix of *precedences*, the transition matrix technique uses a matrix of *subroutine names*. The top operator stack element and the incoming symbol determine an element of the matrix which is the name of a subroutine to execute; this subroutine then performs the necessary reduction or pushes the symbol into the operator or operand stack. Thus the productions do not have to be searched at each step to determine the reduction to make.

The transition matrix has been used as an analytic syntax language in a number of compilers. Gries has written a constructor which builds a transition matrix recognizer for a large class of operator grammars. The restriction to operator grammars was made so that the constructed recognizers would be similar to the recognizers produced by hand.

The constructor begins by using the following scheme to reduce the number of elements in the stack which must be tested in order to find the beginning of the prime phrase (not the handle). Suppose that

⟨cond⟩ → **if** ⟨be⟩ **then** ⟨expr⟩ **else** ⟨expr⟩ (4.1)

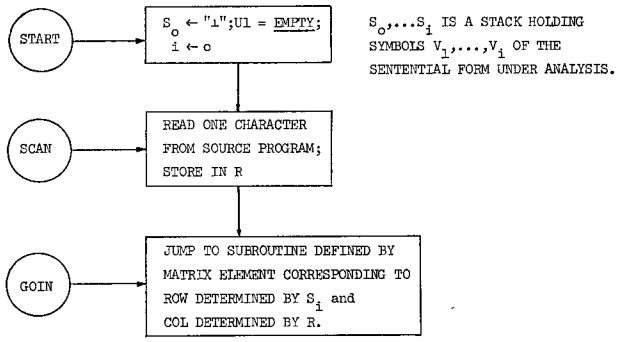


FIG. 8. Transition matrix recognizer

is a production of the grammar. This would be changed to the four productions

"if" → **if**
"ibt" → "if" ⟨be⟩ **then**
"ibtee" → "ibt" ⟨expr⟩ **else**
⟨cond⟩ → "ibtee" ⟨expr⟩

These intermediate reductions merely allow us to put into the stack a representation "ibt" of the three symbols **if** ⟨be⟩ **then.** Similarly we represent **if** ⟨be⟩ **then** ⟨expr⟩ **else** by "ibtee". These new "quoted" nonterminals we represent by V, V_1, V_2, etc.

At any step of the parse, then, the stack will consist of symbols V_j, $j = 1, \cdots, i$, each of which is a quoted symbol—a representation for the head of the right part of some original production—plus perhaps a final nonterminal operand U. The sequence

$$V_1V_2 \cdots V_i(U)T_jT_{j+1} \cdots T_m$$

is thus not a sentential form but a *representation* of one. (The parentheses around U indicate that it may or may not be there.) In general there are three possible actions at each step of the parse:

1. $(U)T_j$ may form the head of another right part, yielding upon reduction $V_1V_2 \cdots V_iV_{i+1}T_{j+1} \cdots T_m$;
2. $V_i(U)$ and perhaps T_j may form a right part to be reduced, yielding $V_1V_2 \cdots V_{i-1}U_1T_j \cdots T_m$ or $V_1V_2 \cdots V_{i-1}U_1T_{j+1} \cdots T_m$;
3. $V_i(U)T_j$ may form the head of some right part, yielding upon reduction $V_1V_2 \cdots V_{i-1}V_i'T_{j+1} \cdots T_m$.

The constructor checks each pair V, T and each triple V, U, T to see which of the three possibilities exist. If at most one exists for each pair and triple, and if the reduction to be performed is unique, then the transition matrix and subroutines are constructed as follows: One row is allotted to each quoted symbol V and one column to each possible terminal symbol T. In each matrix element $M_{V,T}$ is stored the number of a subroutine to execute if V appears as V_i and T as the incoming symbol. A corresponding subroutine is constructed which checks for the presence of the nonterminal U and executes the appropriate reduction. We use a single stack, V_1, V_2, $\cdots$, V_i

TABLE IV

V_i \ T_j	⊥	+	*	(	)	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 U1 = E or U1 = T or U1 = P
    then SUCCESS EXIT else ERROR;
2:  if U1 = T or U1 = P
    then begin i ← i − 1; U1 ← E; go to GO IN end else
      ERROR;
3:  if U1 = P
    then begin i ← i − 1; U1 ← T; go to GO IN end else
      ERROR;
4:  if U1 = E or U1 = T or U1 = P
    then begin i ← i + 1; Si ← "E+"; U1 ←empty; go to
      SCAN end else ERROR;
5:  if U1 = P or U1 = T
    then begin i ← i + 1; Si ← "T*"; U1 ←empty; go to
      SCAN end else ERROR;
6:  if U1 = empty
    then begin i ← i + 1; Si ← "("; U1 ←empty; go to
      SCAN end else ERROR;
7:  if U1 = E or U1 = T or U1 = P
    then begin i ← i − 1; U1 ← P; go to SCAN end else
      ERROR;
8:  if U1 = empty
    then begin U1 ← P; go to SCAN end else ERROR;
```

and put the U in location $U1$. The basic recognizer is given in Figure 8; the matrix and subroutines generated from the grammar in Figure 1 are given in Table IV.

A matrix for ALGOL is about 50 × 40, with perhaps 500 subroutines. The checks for $U1$ = **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 semantics may be inserted at any step of the parse (in any of the subroutines 1–8, Table IV), and not just when a right part is recognized. The grammar does not have to be changed much, although it must be an operator grammar. The constructor itself has not yet been used to generate a compiler, but the generated recognizers closely resemble recognizers built by hand using the same technique (see [Grie 65]).

This is perhaps the fastest technique. In general, switching tables are used whenever speed is essential. Note that the productions need not be searched each time to determine the reduction to make and the semantic routine to execute. Its drawbacks are the space used and the large number of subroutines needed to implement the technique.

B.5. PRODUCTION LANGUAGE

(Floyd [Flo 61], Evans [Ev 64], Earley [Ear 65])

Production language is an *analytic* syntactic metalanguage for writing compilers, introduced by Floyd and modified by Evans. It consists of a set of *productions* (note carefully the different use of the word *productions*), an example of which is:

```
PROGRAMO:   ⊥  |          |*E0
            σ  |          |ERROR EXIT
E0:T0:P0:   (  |          |*E0
            I  |  →    P  |*P1
            σ  |          |ERROR EXIT
E1:         ⊥E⊥ |         |SUCCESS EXIT
            (E) |  →    P  |*P1
            E + |          |*T0
            σ  |          |ERROR EXIT
T1:         T* |          |*P0
          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

$L0$: $S_3S_2S_1 \mid \rightarrow S_2'S_1' \mid *G1$

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 , ··· directly to the left of the first bar "|". When a match is found, the matched symbols S_1,S_2 , ··· in the stack are replaced by the symbols S_1', S_2' ··· . (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 ($G1$ in this case). Any production may be labeled. Earley has written a constructor which produces, from a suitable (synthetic) phrase structure grammar, a recognizer written in production language.

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

Semantics are introduced once 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.

It is important to realize that a production language description is a *deterministic* description of a language. It is actually a language for writing recognizers which parse sentences of a language. This is not the case with the usual phrase structure grammar.

Production language, or a variation of it, has been used in a number of systems. 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. The EXEC 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 by the other four.

We would venture to say that this branch of TWS 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 II.C.

The operator precedence technique is the most well-known of the techniques. It is often used to recognize portions of a language, most frequently arithmetic and Boolean expressions, as is done in the IBM 360 (H-level) FORTRAN compiler. See [Ar 66, Grie 65] for documentation of other compilers using this technique. [Gall 67] also mentions it. The transition matrix technique (but not its constructor) has been used to write several ALGOL compilers [Grie 65, Sam 60], especially within the ALCOR group, 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 it 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], and 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].

In operator precedence, precedence, and extended precedence recognizers, each time a handle (or prime phrase) is recognized, the productions must be searched to find the symbol to which the handle should be reduced and the semantic routine to be executed. Similarly, when using the Floyd-Evans analytic production language, the stack may be matched against several possibilities before a reduction can be performed. All these methods are therefore slowed down unless some efficient table searching can be performed. The transition matrix technique solves the problem by coding a separate subroutine for each possible reduction and by using the switching table; the disadvantage here is the amount of space used for the matrix and subroutines.

For the theoretically inclined reader, we now proceed to discussions of more general, powerful, and complicated (and therefore less efficient) left-right recognizers. Basic references on the theory of formal languages are also given at the end of the next section.

REFERENCES FOR II.B

Operator precedence: Ar 66, Flo 63, Gall 67, Grie 65.
Precedence and extended precedence: McKee 66, Wir 66a, Wir 66b, Wir 66c.
Transition matrices: Grie 65, Grie 67, Hals 62, Mas 60, Sam 60.
Production language: Che 65, EvA 64, Feld 66, Fie 67, Grie 67b, It 66, Mond 67, Rov 67.

C. Formal Studies of Syntax

C.1. BOUNDED CONTEXT GRAMMARS (Eickel [Ei 63, 64] Floyd [Flo 64a], Irons [Ir 64], Wirth and Weber [Wir, 66c])

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. During a parse, a left-right recognizer may thus find the handle of a sentential form 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.B are essentially (1,1) bounded context grammars.

In a sense, to construct a (bottom-up) recognizer for an (m, n) bounded context grammar $\mathcal{G}$ is to construct an equivalent *context sensitive grammar* [Gin 66a, p. 9] $\mathcal{G}_1$ from $\mathcal{G}$. A context sensitive grammar is a grammar with productions of the form $xUy \rightarrow xuy$.

Thus, in such a grammar, a replacement of u by U can be performed only if x is to the left and y to the right of u. When building the recognizer, context is added to the left and right in each production (and thus more productions are constructed), until the grammar states explicitly and unambiguously in what context each reduction can be performed.

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.

One of the earlier papers on constructing recognizers for (1,1) bounded context grammars was [Ei 63]. The first step in the construction algorithm is to insert intermediate productions to change the length of right parts of productions to one or two. Thus the productions

$$U_1 \rightarrow abcd, \quad U_2 \rightarrow abd$$

would be changed mechanically to

$$U_3 \rightarrow ab, \quad U_4 \rightarrow U_3c, \quad U_1 \rightarrow U_4d, \quad U_2 \rightarrow U_3d.$$

(Contrast this with the similar technique used in Section II.B.4.) Now when the recognizer looks for the handle at the top of the stack, the two top stack symbols S_i and S_{i-1} and the incoming terminal symbol T_j 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 onto the stack.
4. S_1S_2T may not appear as a substring of a sentential form.

The algorithm for producing the triples and the corresponding actions 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.

There have been three major papers on general bounded context analysis. Each defines "context bounded" slightly differently. The idea behind all of them, though, is the same, and we do 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, although neither gives an algorithm for actually constructing the recognizer. Eickel's aim [Ei 64] is to describe the recognizer and its construction in detail (and his paper is therefore less readable than the other two). The recognizer uses the usual stack. As in [Ei 63] the grammar is first reduced to one in which all productions have length 1 or 2. The constructor then produces 5-tuples

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

where x, y are strings with length $(x) \leq m$ and length $(y) \leq n$, S is a symbol, U a nonterminal, and k a positive integer. Suppose the stack contains

$$S_0 \cdots S_{i-1}S_i$$

and let

$$T_jT_{j+1} \cdots T_{j+l-1}\ T_{j+l} \cdots T_m$$

be the rest of the input string, where $l > 0$. The number l specifies that the first l symbols of the input string are needed as context to the right at this point. The 5-tuples are searched until one is found such that $S = S_i$, x is a tail of $S_0 \cdots S_{i-1}$, and y is a head of $T_j \cdots T_{j+l-1}$.

The step to be taken depends on the corresponding k and U as follows:

k action

0 stop—syntax error
1 replace handle S_i by U (make a reduction $U \rightarrow S_i$)
2 $i \leftarrow i - 1$; $S_i \leftarrow U$ (replace handle $S_{i-1}\ S_i$ by U)
3 **if** $l = 0$ **then** $l \leftarrow 1$ **else begin** $i \leftarrow i + 1$; $S_i \leftarrow T_j$; delete T_j from input string; $l \leftarrow l - 1$; **end**
4 $l \leftarrow l + 1$ (more context needed on the right).

Note that, although a grammar is (m, n) bounded context, n symbols to the right are not always necessary. The recognizer uses only as much context as is necessary to determine the action to be taken; this is the reason for the number l above.

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 k (or the same k but different U), then the grammar is not (1,1) bounded context. For such 5-tuples, the lengths of x and y are increased, thus adding more context (and more 5-tuples), until the conflict is resolved or some maximum m, n are reached.

Wirth and Weber [Wir 66c] have extended the idea of precedences between symbols X and Y (see Section II.B.2) to strings x and y. Thus we have $x \ominus y$, $x \circledcirc y$, and $x \circledast y$ where length $(x) \leq m$ and length $(y) \leq n$. An (m, n) precedence grammar is of course also (m, n) bounded context according to our definition. A precedence grammar according to Section II.B.2 is a (1,1) precedence grammar.

C.2. Deterministic Pushdown Automata (Ginsburg and Greibach [Gin 66b])

A DPDA is an automata-theoretic formalization of the concept of a left-right recognizer working with a stack. One has a finite set $\mathfrak{S} = \{\mathfrak{s}_0, \cdots, \mathfrak{s}_n\}$ of "states" containing a start state $\mathfrak{s}_0$, a set of inputs $\mathfrak{A}$ (terminal symbols), a set η (corresponding to our nonterminal symbols) containing a start symbol Z and a mapping δ;

δ: (states $\times$ (nonterminal symbols) $\times$ (input symbols)) $\rightarrow$ (states $\times$ (strings of nonterminal symbols))

or

$$\delta\colon\ (\mathfrak{S} \times \eta \times (\mathfrak{A} \cup \{\wedge\})) \rightarrow (\mathfrak{S} \times \eta^*).$$

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

$$\underbrace{(\mathfrak{s}_p,}_{\text{state}}\ \underbrace{U_1 \cdots U_i,}_{\text{stack}}\ \underbrace{T_j \cdots T_m)}_{\text{rest of input}}$$

(where $i \geq 1$), the initial triple being $(\mathfrak{s}_0, Z, T_1 \cdots T_m)$. At each step, with the help of the mapping $(\mathfrak{s}_p, U_i, T_j) \rightarrow (\mathfrak{s}_q, U_1' \cdots U_n')$ where $n \geq 0$, the triple gets changed to

$$(\mathfrak{s}_q, U_1 \cdots U_{i-1}U_1' \cdots U_n', T_{j+1} \cdots T_m).$$

A string (of inputs) is *accepted* if the final state $\mathfrak{s}_m$ 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. Note that a deterministic language is defined by a *machine*—a DPDA—and *not* by certain properties of a grammar defining the language. Ginsburg and Greibach prove some interesting properties of DPDAs and deterministic languages. What is significant for us here is the relation to LR(k) languages of Knuth (Section II.C.3).

Note that one can implement a DPDA using a transition matrix M, where each possible state $\mathfrak{s}_p$ is represented by a row and each terminal T by a column. At each step the matrix element $M_{\mathfrak{s}_p,T_j}$ then determines a subroutine which performs the appropriate mapping depending on the U_i at the top of the stack. The transition matrix technique is thus fairly general; but the constructor written by Gries (Section II.B.4) only accepts a subset of the (1,1) bounded context grammars.

C.3. LR(k) Grammars (Knuth [Knu 65])

A grammar is LR(k) if and only if a handle (p. 80) is always uniquely determined by the entire string to its left and the k terminal symbols to its right. The corre-

sponding 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-right bottom-up recognizer that can be mechanically produced from the grammar. In fact, a grammar accepted by any of the other constructors discussed is LR(k) for some k. Thus the LR(k) condition is also the most powerful general test for unambiguity that is now available.

Knuth gives two algorithms for deciding whether or not a grammar is LR(k) 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). This may look strange at first sight, since there are, in general, an infinite number of strings $S_1, S_2, \cdots, S_i$ which may appear in the stack and thus an infinite number of strings which must be used to make parsing decisions, while a DPDA requires only the top stack element, a state, and the incoming symbol.

However, since the number of productions is finite, at any step of a parse there are only a finite number of possible actions, no matter which symbols $S_1, S_2, \cdots, S_{i-1}$ are in the stack. Thus it is only necessary to classify the possible strings $S_1, S_2, \cdots, S_{i-1}$ into classes, called "states" $\mathcal{S}_p$ and show that, if the grammar is LR(k) for some k, one can describe the necessary single-valued mapping:

$$(\mathcal{S}_p, S_i, T_j) \rightarrow (\mathcal{S}_q, S_1' \cdots, S_n').$$

Of course we have greatly simplified the process here in trying to get across the idea; the symbols S_i in the stack of the DPDA are not only symbols of the original grammar but may themselves be complicated sets in order to be able to determine the necessary single-valued mapping.

Knuth also proves by construction that for each language $\mathfrak{L}$ accepted by a DPDA there is an LR(1) grammar which defines $\mathfrak{L}$. Thus any LR(k) language is also LR(1), although not with the same grammar. Earley [Ear 67] has written a constructor for LR(k) grammars, whose output is in the form of productions similar to the Floyd-Evans productions.

C.4. Recursive Functions of Regular Expressions (Conway [Con 63], Tixier [Tix 67])

Conway discussed a compiler whose syntax is specified by a number of *transition diagrams*, each of which recognizes strings derivable from a certain nonterminal, such as in Figure 10.

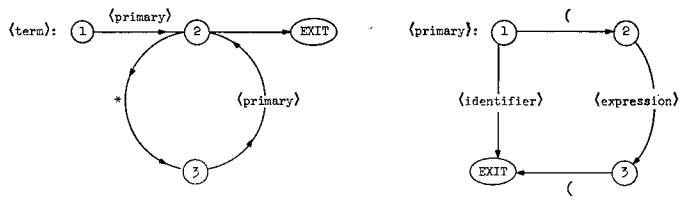


Fig. 10. Two transition diagrams

At each step of the parse of a sentence, there exists a *current node* of some *current diagram*. The action performed by the left-right recognizer (working with the diagrams plus a pushdown stack) is determined as follows:

(a) If the next input symbol (a terminal) matches the name of one of the lines emanating from the current node, traverse that line to the new current node and scan the next input symbol.

(b) If no match as in (a) occurs, and if one line is labeled with a nonterminal U, then traverse that line, push the current diagram name and the new current node onto the stack, and change the current node to the first node of the diagram labeled U.

(c) If (a) and (b) do not occur and there is an unlabeled line, traverse that line.

(d) If the current node is an EXIT, we have recognized a string derivable from the nonterminal described by the current diagram. Change the current node (and diagram) to the one specified by the top stack element and delete that element from the stack.

This is a top-down left-right recognizer without backup; note the prediction in step (b) that a string derivable from U will be recognized. This method effectively breaks the syntax analysis into small, simple parts and saves space, since the character set involved in each subroutine is quite small. It is clear that each transition diagram represents a finite state automaton which is capable of recursively calling other finite state automata.

Tixier independently formalized this concept in his thesis. He considers the productions $U_i \rightarrow x_i$ to be regular expressions $U_i = x_i$, where the set operations union (+), product, and closure (*) are used. Thus the productions

⟨identifier⟩ → ⟨letter⟩

⟨identifier⟩ → ⟨identifier⟩ ⟨letter⟩

may be written equivalently as

⟨identifier⟩ = ⟨letter⟩ + ⟨identifier⟩ ⟨letter⟩

or

⟨identifier⟩ = ⟨letter⟩ ⟨letter⟩*.

Tixier has rewritten the 120 productions for Euler [Wir 67c] as 7 functions of the 7 variables, 3 of which we give here, with the symbols "(", ")" used as metasymbols to bracket set expressions:

program = *block*

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

expr = (**out** + **if** *expr* **then** *expr* **else** + *id* ([*expr*] +.)*←)*
(**go to** *primary* + *block* + *catena*)

Tixier's constructor can manipulate the equations (productions) of a grammar, if desired, to arrive at the smallest number of equations, the main goal being to parse as much as possible via efficient finite state automata. For a suitable grammar the constructor then builds a (modified top-down) recognizer which can parse any sentence of the language unambiguously and without backup, in the form of a set of finite state automata (one for each equation) calling each other recursively. The main

difficulty here is to be able to determine unambiguously from the current node and the input symbol *if* another finite state automaton should be called, and if so, *which one*. An accepted grammar, together with its language, is called *regular context free* (RCF).

The constructor actually builds an efficient, restricted DPDA. Thus RCF languages are LR(1) languages.

C.5. Summary

Figure 11 presents an inclusion tree for the classes of grammars accepted by the particular constructors discussed in this paper. This tree may be confusing in some places. Some metalanguages, such as Floyd's production language, are powerful, but a particular constructor of a recognizer using such a metalanguage may restrict the grammars acceptable. If a node has a reference on it, the node refers to the language (or constructor) defined in that paper. The following should be noted (see Figure 11):

(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. Of course, any constructor could be changed to include these intermediate reductions.

(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 disallow identical right sides. Otherwise inclusion does not hold. The advantage of the matrix technique over operator precedence is, as in (a), the use of automatic intermediate reductions.

(d) Feldman shows [Feld 64] that each DPDA is equivalent to some program written in production language.

Let us for a moment return to the problem of classifying recognizers as top-down or bottom-up. Tixier's recognizer (Section III.C.4) is definitely top-down; you can see the prediction (the goal) being made when he switches to a new finite state automaton. Is Knuth's constructed recognizer for LR(k) grammars top-down? Some would say no. The recognizer just makes reductions using the k symbols to the right and the stack symbols to the left as context. The concept of LR(k) itself, together with its recognizer, is an extension of the (m, n) bounded context recognizers, which are generally bottom-up. Furthermore, the extra "states" have been added to the stack just to aid in the determination of the reduction to be performed. However, others say that by using these states a prediction or goal is actually being introduced. In favor of this is the fact that both Tixier's recognizer, which is top-down, and Knuth's recognizer are both DPDA. Here we can truly say that these concepts have merged, and a case could be made for either.

We have attempted to survey a few of the concepts occurring in the study of syntax related to compiler writing; much has been omitted. Mention should be made of Gilbert [Gil 66] who adds to a context sensitive grammar a *selection function* that indicates (based on the sentential form) which production to use (if several possibilities

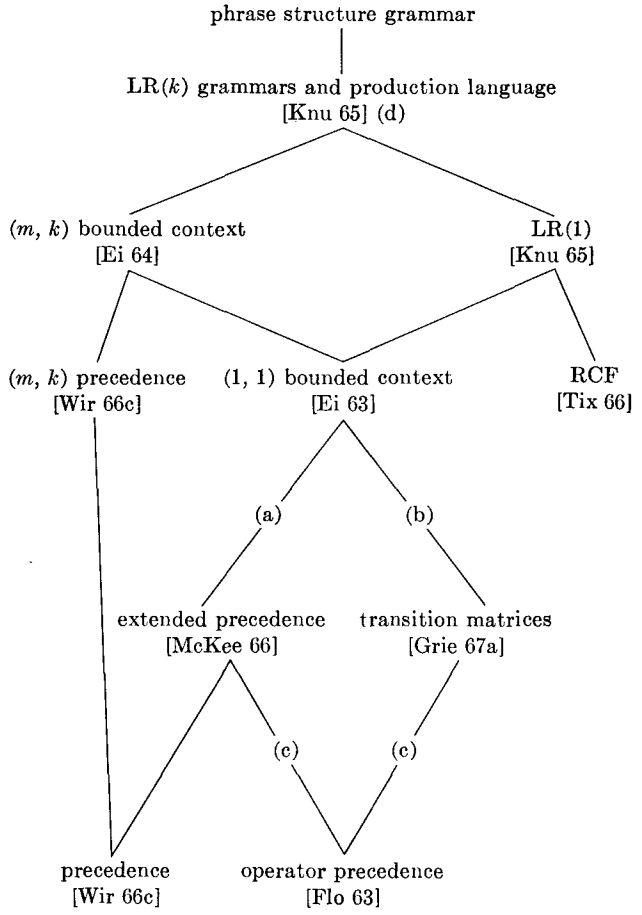


Fig. 11. Inclusion tree

exist) in making the reduction. Thus with the use of the selection function the synthetic grammar can be used as an *analytic* grammar. Gilbert proves several properties of these grammars and selection functions and shows that one can completely describe the syntax of languages such as Algol and Fortran (including comparisons of declarations and uses of variables).

We have not surveyed top-down recognizers in detail, since they are covered well in [Che 64c] and [Flo 64b]. The problem of ambiguity in context free languages has been covered only slightly as it relates to TWSs. The automata theory field is also related but has not been mentioned.

In fact, we have had to omit from the bibliography many papers dealing with context free grammars, automata theory, and machines. Many of these, and references to almost all the others, can be found in the *Journal of the Association for Computing Machinery*, in *Information and Control*, and in [Gins 66a].

References for II.C

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, Kir 66, Kun 62, 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, Gil 66, 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, GraR 64.

Ambiguity in context free languages: Can 62, Flo 62a, Flo 62b, Gin 66a, Gor 63, Lang 64, Ross 64.

Thirteen different ways to define languages: Gorn 61.

III. SEMANTICS

A. 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. Most of the early TWS efforts were of this type, the most notable being [Ir 61]. Since such systems are more general, they have been used heavily in the various nontranslator tasks described in Section IV.A. In fact, the discussion of AED [Ross 67] is deferred to that section, because its goals have been more general from the outset.

A.1. TMG (McClure [McCl 65])

The TMG system was developed at Texas Instruments as a tool for writing simple one-pass compilers that produce symbolic output. The syntax technique is a simple top-down scan with backup. However, the embedding of semantic rules allows the recognizer to be more efficient by eliminating some syntactically possible goals on semantic grounds.

The basic TMG statement form is a sequence of actions separated by spaces. Any action may be preceded by a ⟨label⟩, and it may be followed by a ⟨destination⟩ which is taken in case the action fails. The ⟨actions⟩ 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 in building an intermediate tree. The actual output of code is done by a different set of routines, which are 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 action *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 to be entered in the intermediate tree. The only other information allowed in the table is a set of declared *FLAGS* (Boolean variables) used to describe the attributes of identifiers.

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 = $($*P*1 / *BSS* / 0 // $).

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

value (*P*1) *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 output routines operate from top to bottom on the intermediate tree representation of a program. Thus a $\$Pn$ in an output routine may refer to a subtree and the evaluation of $\$Pn$ will then involve a recursive call on another output routine. It is also possible to pass parameters by value to the inner routine. The paper gives several examples of these functions and includes a brief discussion of the error recovery capabilities of TMG.

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 of the systems considered below and would benefit from another iteration.

A.2. THE META SYSTEMS (Schorre, [Schor 64], Schneider and Johnson [Sch 64])

The META systems are the product of the Los Angeles SIGPLAN working group on syntax-directed compilers. Although the original work was diversified, the current systems are generally based on a model known as META-II, developed by Schorre [Schor 64]. 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 implemented. The rules do not allow left recursion, using instead the prefix iteration operator $. Terminal symbols are quoted; system symbols are preceded by "."; and all unmarked symbols are user's nonterminals. Parentheses are used to group alternates within right parts. The following rules are used in translating Boolean expressions:

1. *UNION = INTER ('OR' .OUT('BT' *1) UNION .LABEL *1|.EMPTY);*
2. *INTER = BPRIMARY ('AND' .OUT('BF' *1) INTER .LABEL *1 | .EMPTY);*
3. *BPRIMARY = .ID .OUT('LD' *) | '('UNION')';*

The last rule defines a procedure for recognizing a Boolean primary in an algebraic language. The word *BPRIMARY* 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 union and a code generator in case an identifier (*.ID*) is found. The above rules contain examples of the three basic entities used in most META compilers. The mention of the name of another rule, in this case *UNION*, causes a recursive call on that recognizer to be invoked. The occurrence of a literal string "(" states that the input stream is to be tested for a left parenthesis. The output statement *.OUT* pro-

duces a line of text, where "*" always refers to the last item recognized by the primitive nonterminal *.ID*.

The first mention of a *1 within a rule (as in Rule 1 above) 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 may be generated. These labels exist only 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 action *.LABEL* *1 indicates that the label corresponding to *1 is to be written out. *.EMPTY* is a primitive nonterminal which has no effect on the input but is always satisfied or true.

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 usefulness of META-II was severely limited by the lack of facilities for backup or for reordering the output. There have been several attempts to extend the META techniques to a complete TWS. META-3 [Schor 64] 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 has been used in a number of format conversion and source-to-source language translations, but has not been used for compilers. The most recent development is TREE META, a multipass system using complex processing of intermediate syntax trees. The slowness and inefficiency of META compilers is recognized by their authors, but the ease of implementation, the boot-strapping capabilities, and the large class of languages they can handle are used to justify the work that has gone into their development.

A.3. COGENT (Reynolds [Rey 65])

The COGENT system, designed at Argonne National Laboratory and implemented on a CDC 3600, draws heavily on the ideas of Brooker and Morris (Section III.B.3), Irons [Ir 61, May 61], and LISP COGENT is very well thought out and is considerably more comprehensive than the other systems described in this section. The COGENT compiler is written completely in its own language. By boot-strapping three times, its own compilation speed has been increased by a factor of six.

A program written in COGENT consists of two parts: the syntax and a set of processing routines called generators. The syntax is given by a synthetic phase structure grammar. Almost any context free grammar is acceptable, including those with left recursion; only a few restrictions are made concerning empty right parts. The recognizer which uses the grammar is modified top-down, with alternatives at each step being processed in parallel. A string is accepted if the recognizer finds a unique syntax tree for it.

Syntactic analysis produces a list structure to represent the intermediate tree. For example, use of the production

⟨term⟩ ::= ⟨term⟩ + ⟨factor⟩

would produce a list element ⟨term⟩ with pointers to the subtrees for ⟨term⟩ and ⟨factor⟩.

One can precede any production by a name of a generator (semantic routine), which is then executed when that production is used in building the tree. When there is more than one possible syntax tree (due to parallel processing of alternatives), the execution of these generators is delayed and syntax analysis continues until the local ambiguity is resolved and only one tree remains. Then all the generators are called in the correct order.

As an example, consider the labeled production:

processterm / ⟨term⟩ ::= ⟨term⟩ + ⟨factor⟩

When a subtree with ⟨term⟩ as the root is completely formed, the generator *processterm* will be called, with the subtrees for ⟨term⟩ and ⟨factor⟩ as arguments. *Processterm* may manipulate these subtrees, delete them, produce code corresponding to them, and so forth.

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, 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⟩, used for pattern matching, looks like a production in parentheses with "/" substituted for ":: =". The statement causes the ⟨identifier⟩ to be a copy of the ⟨template⟩, in which the *i*th parameter (nonterminal) is replaced by the valve of the *i*th expression in the ⟨expression list⟩. For example, the execution of the synthetic assignment statement

Z /= (⟨term⟩ / ⟨factor⟩*⟨factor⟩), X, Y

where X has 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 ith parameter (nonterminal) of the template is assigned to the ith identifier of the ⟨identifier list⟩. Thus if Z has the value (⟨term⟩/ *ABE* * *BED*), then the statement

$$Z = / (\langle term\rangle / \langle factor\rangle * \langle factor\rangle), X, Y$$

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. This failure proceeds up the chain of generator calls until some conditional statement is encountered.

A program in COGENT can use any number of symbol tables. The action label $IDENT n specifies that the result of that production (which must be a character string) should be placed in symbol 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. Each entry in a table consists of the identifier plus a pointer element, which normally points to the attributes of that identifier.

Output is achieved by calling the routine *PUTP* with a single parameter—the internal code of a character to be placed in the output line. When the line is filled, it is written out and a new line started; however, *OUTP* can be called to print out a line before it is completely filled. Another primitive generator, *STANDSCN* (*X*, *PUTP*), will map a list structure X into the string S represented by its end nodes. *STANDSCN* finds the symbols in S and passes them one at a time to *PUTP*.

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. 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 on (cf. Section IV.C).

A.4. ETC (Garwick [Gar 64], Gilbert [Gil 66] and Pratt [Pra 65])

In this section we describe three other efforts, that are best described as syntax-directed symbol processors. For various reasons, these systems have not had the impact of those discussed above and are presented in less detail.

The GARGOYLE system [Gar 64] was developed for the Control Data 3600 by Jan Garwick at the Norwegian Defense Establishment. There are reasonably complete descriptions in internal reports, but the published paper [Gar 64], which was written first, gives only a vague picture of GARGOYLE.

GARGOYLE is like TMG (Section III.A.1) in many ways; the recognizer is top-down with a facility for directing the search. All syntactic and semantic statements are written in a five-entry tabular form: ⟨label⟩ ⟨else⟩ ⟨next⟩ ⟨link⟩ ⟨action⟩. The sequencing rules are quite complex (partly because backup is handled implicitly) and are normally done by a "steering routine." The backup mechanism also requires complicated data handling involving stacking of some variables and copying of others. All five entries are used in multiple ways; e.g. the label may instead be a code controlling how the line is to be interpreted. The following example would be used to process an ⟨assignment statement⟩.

⟨label⟩	⟨else⟩	⟨next⟩	⟨link⟩	⟨action⟩
R		*ASSIGN*		
L				*VAR*
0		*VARIABLE*	1	
1		*NEXT*	2	**if** *SYMB*=:*COLEQ* **then** *VAR*←*WORD*
2	3	*ASSIGN*	4	
3		*EXPR*	4	
4		*RETURN*		*OUTTEXT* (*"STO"*, 10); *OUTFIELD* (*VAR*, 20);

The first line is the header for routine *ASSIGN*, whose local variable, *VAR*, is declared in the second line. If *VARIABLE* finds a ⟨variable⟩ then the next symbol must be *COLEQ* (":=") or the routine fails. Line 2 is a recursive call of *ASSIGN* for treating multiple left sides; the first backup will lead to *EXPR* (which will compile the right side), and subsequent returns will execute statement 4 once for each variable in the multiple assignment. The ⟨action⟩ in statement 4 produces text for storing into each successive value of *VAR*.

The language of the ⟨action⟩ column includes partial word operators and a few fixed routines for table searching, input-output, etc. The GARGOYLE user is expected to embed assembly statements frequently, and the entire ⟨action⟩ language appears similar to the high level machine language of [Wir 66a]. GARGOYLE has been used by its author to help implement a complex language [Gar 66], but its wider use will require a somewhat cleaner design and considerably better publication.

The TWS of Gilbert and McLellan [Gil 67] is based on some syntactic ideas of Gilbert (Section II.C, [Gil 66]) and an attempt to revive the UNCOL [Ste 61] concept. Source programs are to be translated into a machine-independent language, BASE, which in turn can be translated into many machine languages. The first translation is described in an intermediate language which is a macro assembly language having a few fixed data structure conventions; the second translation is described in a string form like that of TMG. Although there are a number of good ideas in the paper, they are not significantly different from others in this section. Some of the bad ideas, however, are uniquely illustrative.

The UNCOL notion of a machine-oriented but machine-independent language has always foundered on the diversity of languages and computers. Gilbert and McLellan attempt to avoid this by allowing new operators to be de-

fined in BASE and passed blindly to each BASE-to-machine-language translator. This gives the appearance of machine-independence but leaves untouched the basic problem of which macros to choose. The authors also make a point of the fact that their system is "rigorously based." This presumably encouraged them to use the set of strings "$A^mB^nA^mB^nCCC$" as the programming language example illustrating all aspects of their system. Finally, in a classic example of misplaced rigor, they exclude infinite loops from their system by not permitting loops and **go to** statements. The only reference in this paper [Gil 67] to other TWS literature is [Ir 61].

The AMOS system developed by Pratt and Lindsay [Pra 65, Pra 66] is a direct application of list-processing ideas to TWSs. The source language is translated into an intermediate language (I-language) which is interpreted by AMOS. The I-language is a simple list processing language with some string processing operations and crude arithmetic; e.g. "**ADD* P1, P2, P3*" means "add *P1* to *P2* and put the result in *P3*." The I-language was designed to be minimal and to be expanded in macro fashion. The syntax is treated by a variant of production language (Section II.B.5) relying heavily on recursive calls; the semantics is written in I-language. The most interesting feature of AMOS is the attempt to provide for the translation of data structures as well as programs. AMOS has had some minor successes in handling list structures, but the problem deserves much more attention (cf. Section IV.C).

REFERENCES FOR III.A
Ab 66, Gar 64, Gar 66, Gil 66, Gil 67, Kirk 65, Ir 61, McCl65, Met 64, Pra 65, Pra 66, Rey 65, Sch 64, Schm 63, Schor 64.

B. Compiler-Compilers

The distinguishing characteristic of this set of TWSs is the attempt to automate many of the postsyntactic 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 metalanguage processors. The programs in this section are more complex than most of those discussed previously; none has ever been successfully implemented by someone not in contact with a previous effort of the same type. The following excerpt from a paper on FSL (Formal Semantic Language) [Feld 66] outlines basic philosophy and should serve as an adequate introduction to our discussion of compiler-compilers.

When a compiler for some language, £, is required, the following steps are taken. First the formal syntax of £, expressed in a syntactic metalanguage, is fed into the syntax loader. This program builds tables which will control the recognition and parsing of programs in the language £. Then the semantics of £, written in a semantic metalanguage, is fed into the semantic loader. This program builds another table, this one containing a description of the meaning of statements in £. Finally, everything to the left of the double line in Figure 12 is discarded, leaving a compiler for £.

The resulting compiler is a table-driven translator based on a recognizer using a single LIFO stack. Each element in this main stack consists of two machine words—one for a syntactic construct and the other holding a semantic description of that construct.

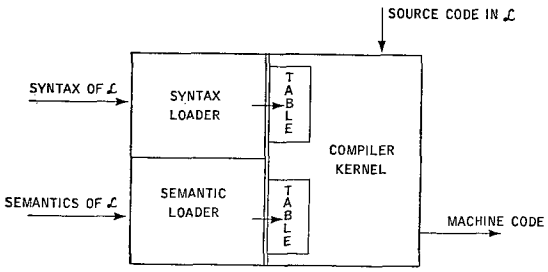


FIG. 12. A compiler-compiler

When a particular construct is recognized, its semantic word and the semantic table determine what actions the translator will take.

The compiler kernel includes input-output, code generation routines, and other facilities used by all translators.

B.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 postsyntactic (semantic) processing. An adequate semantic metalanguage 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 metalanguage should not depend on the characteristics of a particular computer.

The syntax metalanguage used in FSL is very close to the production language discussed in Section II.B.5. 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 main stack.

The semantic metalanguage, FSL, was the main focus of effort and is 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 metalanguage be machine independent, with the machine dependent aspects of translation handled by a large set of primitives embedded in the compiler kernel. Statements in the metalanguage would be compiled (whence compiler-compiler) 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: ENTER[SYMB; LEFT2, (STORLOC | DOUBLE), REAL,
       LEV];
   STORLOC ← STORLOC + 2
```

Here a description of the variable is placed in the symbol table, *SYMB*. The entries for the variable are its name, a tagged address, the word *REAL*, and the current block 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 NOT LEFT1 IS CONSTANT THEN
     IF SYMB[LEFT1, TYPE] = REAL THEN
       RIGHT1 ← SYMB[LEFT1, SEMANTICS]
     ELSE FAULT 1
```

In semantic Routine 2, if the identifier (in *LEFT*1) is a constant, 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

⟨table name⟩ [⟨operand⟩, ⟨position name⟩].

This instance of a table operand initiates a search of the table *SYMB* for an entry (row) whose first column equals the contents of *LEFT*1. Then the specified position (*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 *SEMANTICS* (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: RIGHT2 ← CODE(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. The execution of 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 a code sequence for addition. The semantic descriptions include the data type, sign, index attributes, and current location of an operand; these, along with the state of the translator, are enough to produce locally good code. The result of an addition is itself an expression, and the syntax is presumed to have put the syntactic symbol, *E*, into the second position of the stack (cf. line *T*1 + 1, Figure 9, p. 87). The assignment to *RIGHT*2 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 metalanguage, FSL, allows a compiler writer to declare and use a variety of data structures in building a translator. Besides the tables and cells mentioned in the examples, there are stacks, masks, and strings. The system includes a number of auxiliary routines (e.g. format, file manipulation) available at both translation and execution time. 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, recursive subroutines, 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 metalanguage. This, in turn, is possible only where there are suitable formalizations of the pertinent concepts. Thus all the research problems listed in Section IV.C (e.g. data structures, paging, parallelism) are problems in any FSL system. One common misconception is that FSL requires code to be produced immediately when a construct is recognized. One is allowed to defer code generation indefinitely, but the systems now running do not have particularly good facilities for global code optimization or multipass compilers.

These problems are being attacked in several current FSL-like projects. 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 metalanguage representations should lead to careful study of new techniques. In addition, a particular implementation will normally include informal techniques (e.g. assembly language) for handling constructs not yet formalized.

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 (used in this description), make VITAL much easier to use but are of little theoretical interest. Among the more significant changes is the executable syntactic class name which reduces the size of the syntax table by about one fourth and increases speed. All text is saved in linked blocks of dictionary pointers; this facilitates line editing and reduces recompilation time by about one half. The combined features of persistent storage and compile-time execution aid in the writing of incremental compilers. The user is given considerably more flexibility in register allocation but can choose to abrogate this responsibility as in the original system. A minor but philosophically important change was the addition to the production language of a syntactic ⟨action⟩, TEST, which depends on a variable set by the semantics. This violates the BNF tradition, but it was found to be necessary for some translators and a great convenience in several others.

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 prog-

ress. 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 [Grie 67b] 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.

B.2. TGS (Plaskow and Schuman [Plas 66], Cheatham [Che 65])

One of the most productive groups in TWS research has been the small consulting company, Massachusetts Computer Associates (COMPASS), now part of Applied Data Research. Although their TWSs have undergone many changes, the basic world view and goals of their effort have remained rather constant. They define compiling as a six-step process: 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.B.5), and a number of built-in functions. Other aspects of their efforts are discussed in Section II.C.5 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 Figure 12, the FSL model applies quite well to TGS. In fact, there has been good communication 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.B.5). 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, 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 $\mathcal{L}_{t0}$ 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

① *TIMES* *X* *Y*
④ *STORE* *Z* ①

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

Consider the first 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 labeled *AESET* causes the syntactic type of the new top element to be assigned the value *AE*. Finally, the statement *TRY*(*AE*1) leads to further expression processing.

These two TGS statements, if executed in reverse order, would compile $Z \leftarrow X * Y$ 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 stack.) The subroutine *LOADMQ* is called with a pointer to the second stack operand as parameter. This user-written routine will assemble a *LOADMQ* 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
        EMIT (CHS);
        EMIT (STO, ARG(1));
        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 affect the state of the translator, reclaiming the temporaries and freeing the AC and MQ registers. Finally there is a transfer to the action labeled *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. More recently [Che 66], Cheatham has suggested using a declarative metalanguage $\mathcal{L}_D$ which is meant to be translated into TRANDIR procedures, presumably by a (meta-meta) processor. The translation of the language $\mathcal{L}_D$ is based on a mechanical constructor combining notions of Wirth and Early (cf. Section II.B). 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 $\mathcal{L}_D$ user does not seem appreciably less than that required by TRANDIR.

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 bracket statements as well as multiphase processing. Both VITAL [Mond 67] and the most recent TGS [Plas 66] are interactive and have sophisticated trace, edit, and debug features.

B.3. CC (Brooker, Morris, et al. [Brook 67a, b, c])

The CC (Compiler-Compiler) project started at Manchester University is one of the oldest and most isolated TWS efforts. Although the CC system has been running for some time and has been used to implement several algebraic languages [Cou 66, Kerr 67], the published descriptions are inadequate, and the CC is not generally understood.

The CC effort has concentrated on problems of semantics; the syntax analysis is top-down with memory and one symbol look-ahead (cf. Section II.A). 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 are following the formal treatment here, taking many liberties with their notation.

The input to the syntax phase is similar to BNF with the additions of "?" which can appear within angle brackets (meaning optional) and the repeat operation "*" (to replace left recursion). The following statements could be used to specify the syntax of an assignment statement for arithmetic sums.

1. *FORMAT* [*SS*] = ⟨variable⟩ ← ⟨sum⟩
2. *PHRASE* ⟨sum⟩ = ⟨sign?⟩ ⟨term⟩ ⟨terms⟩
3. *PHRASE* ⟨term⟩ = ⟨variable⟩ | ⟨number⟩ | (⟨sum⟩)
4. *PHRASE* ⟨terms⟩ = ⟨sign⟩ ⟨term⟩ ⟨terms⟩ | ⟨empty⟩

Line 1 is called a *format definition* and makes use of the auxiliary *phrase definitions* on Lines 2–4. The *SS* specifies the *class* of this format and will be discussed below. Both format and phrase definitions are used as "productions" by the top-down recognizer. The difference is that, when an intermediate tree corresponding to a format definition is completely formed, an associated *format* (semantic) *routine* is called to process it. The format routine associated

with Line 1 would be written as follows:

```
 5.  ROUTINE [SS] ≡ ⟨variable⟩ ← ⟨sum⟩
 6.    Let ⟨sum⟩ = ⟨sign?⟩ ⟨term⟩ ⟨terms⟩
 7.    ACC ← ⟨sign?⟩ ⟨term⟩
 8.  L1: GO TO L2 UNLESS ⟨terms⟩ = ⟨sign⟩ ⟨term⟩ ⟨terms⟩
 9.    ACC ← ACC ⟨sign⟩ ⟨term⟩
10.    GO TO L1
11.  L2: STORE ACC IN ⟨variable⟩
12.    END
```

Lines 5 states that this routine is associated with the format of Line 1 and will be called when the syntax has matched Line 1 and built the appropriate intermediate tree. Line 6 assigns descriptors from the intermediate tree as the values of ⟨sign?⟩, ⟨term⟩ and ⟨terms⟩. After code to load the accumulator is compiled (Line 7), the tree is examined to see if the ⟨sum⟩ had more than one ⟨term⟩. If not, the routine compiles a store instruction (Line 11) and exits. More complicated ⟨sum⟩s are treated by the loop of Lines 8, 9, 10. The actual output of code is implementation dependent and is usually done by simple string manipulation routines.

There are three main classes of statements used in CC: basic (*BS*), master (*MP*), and source (*SS*). The *BS* sublanguage parallels the semantic sublanguages of FSL and TGS; it includes code generation, list processing, and lexical analysis routines in an algebraic language. These *BS* statements are further divided into precompiled statements (e.g. Lines 6, 8, 10, 12 above) and translator-specific compilations of *BS* statements (e.g. Line 7, 9, 11) defined by *FORMAT* [*BS*] statements and their associated *ROUTINE*s. *BS* statements can occur only within a format routine.

Statements in the *MP* class include the *FORMAT*, *PHRASE*, and *ROUTINE* statements themselves (Lines 1–5) as well as editing and system dump instructions. None of these constructs can occur within a format routine. The final statements class, *SS*, contains the source language statements themselves. These may be interlaced with *BS* and *MP* statements making CC, in effect, a powerful extendible compiler in the sense of Section III.C. Although the CC system was originally designed to operate this way, actual practice has been somewhat different. One writes the definition of, say, FORTRAN as a set of *FORMAT* [*SS*] and *ROUTINE* [*SS*] statements and the CC compiles these into tables. One then records this updated copy of CC with a switch set to have CC accept only *SS* statements, yielding a conventional FORTRAN compiler. When used this way, CC can be modeled by Figure 12 with *FORMAT* and *PHRASE* statements as the syntax and with *ROUTINE*s as the semantics, linked by an intermediate tree rather than a stack. Notice that CC does not have a facility for handling descriptors for intermediate symbols as FSL and TGS do. Because CC uses a top-down recognizer, constructs are used as they are processed; this eliminates intermediate descriptors, but does force an ALGOL compiler to be written as one large *ROUTINE*.

The CC group has recently produced reports on the uses and performance of their system. These include the first attempt to compare a TWS with hand written compilers [Brook 67b]. Brooker was able in a year to reduce the space requirement by a factor of 1.6 and the time by 1.7 by hand-coding the Atlas Autocode compiler. These results are hard to interpret without more information, but they suggest that compiler-compilers need not be as extravagant in the use of space and time as many people have imagined. This is also suggested by the results of Kerr [Kerr 67].

The CC has been successfully embedded in an ALGOL-like language Atlas Autocode [Brook 67a]. An adaptation of the system called SPG (System Program Generator) is currently under development by Morris at Manchester. SPG is aimed at the systems programmer who has a knowledge of its underlying mechanisms. Implementations of the CC now exist in Atlas, the KDF-9, and the G-21, and an effort is underway at Carnegie-Mellon on the IBM 360/67.

REFERENCES FOR III.B
Design: Brook 60a, 62a, 63, 67b, 67c, Che 64a, 64c, 65, Cou 67, Feld 64, 66, 67, Fie 67, Grie 67b, Mond 67, Mor 67, Nor 63, Plas 66, Ros 64a, War 61, 64.
Uses: Brook 67a, 67b, Cou 66, It 66, Kerr 67, Nap 67, Rov 67.

C. Meta-Assemblers and Extendible Compilers

These forms of TWSs 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 direct 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 a special case of the assembler, while the extendible compiler approach adds text replacement features to standard compilers. Both of these approaches are becoming very popular; a number of papers which appeared too late to be considered here are listed in the references at the end of this section.

C.1. GENERAL DISCUSSION AND METAPLAN (Ferguson [Fer 66])

The article by Ferguson is taken from the ACM Programming Language Conference, San Dimas, California, 1965, 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 for writing assemblers, 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 versus 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 the paper [Fer 66]). 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) GO TO L.

He would have *IF*, *PLUS*, *EQ*, and *GO TO* be defined as (prefix) operators using a scheme called many-many macro. The many-many macro has features for using and updating state information during text replacement. This attacks the main problem in the more general use of macros—the effective use of global properties (state information) in the assembly process. The many-many macro is probably flexible enough to implement any known compiler; the real question is whether the many-many macro is a good way of doing it. The answer to this depends on the mechanisms for recording and using state information, and these are not discussed in the paper.

C.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 symbolic 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.

C.3. XPOP (Halpern [Hal 64, 67c])

The XPOP system has been implemented on the IBM 7090 and is well documented internally [Hal 67c]. At least one language, ALTEXT [Star 65], has been implemented in the system. A few examples from a program written in ALTEXT will illustrate the types of macro calls possible:

DO THRU LAB1 I=1 TO T−1 BY 1
IF 1 CHAR AT NAMES I IS EQUAL TO NAMES I+1, GO TO LAB2
IF U IS LESS THAN 3, GO TO PUT
IF MATCH GO TO (LAB1, LAB2, LAB3), I
COPY 11 CHAR CARDS 1 TO REJECT 1

Any statement in this language is a call on a macro written in IBM 7090 assembly language; thus the programmer may freely intersperse assembly language with his high level language statements. The following features of XPOP make it possible to define macros which handle statements as illustrated above:

1. Macros are usually recognized by the first word of a statement (*DO*, *IF*, *GO TO*), but in some cases the macro name may appear elsewhere on the line.
2. Within each macro definition one can define the punctuation to be used in processing the rest of the statement as parameters of that macro.
3. Within each macro definition one can define noise words (which are ignored) and keywords (which are used in determining parameters) for processing the rest of the statements as parameters.
4. It is possible to defer the assembly of sections of code until a particular label appears. (This is used in generating instructions for the *DO* statement above.)
5. Instructions within macro definitions may be executed at assembly time, providing for checking of (say) global attributes of names and conditional assembly instructions. This provides one with the ability to write compilers in assembly language (as usual) except that the instructions have to be assembled each time they are to be executed.
6. The system has a large number of useful trace and debugging aids.

XPOP has several disadvantages when viewed as a compiler-writing tool. Everything must be written in assembly language or in previously defined macros. Secondly, there are no facilities for implementing symbol tables, etc., to hold attributes of variables, beyond writing them explicitly in assembly language. ALTEXT does very little checking for correct use of names. Thirdly, while the XPOP system has a built-in compiler for arithmetic expressions, it compiles either all floating point instructions or all fixed point instructions (depending on a switch); no type checking is done and no mixed expressions are possible. Finally, languages such as ALGOL which have a high degree of structure cannot be implemented easily; macro calls may not be nested in an easy manner. It is safe to say that one is never sure whether enough macro features have been provided. Addition of a new statement to a language may necessitate another macro feature, just as the feature Number 4 above was implemented to take care of *DO* statements.

Halpern states that he does not intend to replace the other compiler-writing tools used for implementing ALGOL-like languages. He is interested in processing languages whose programs look like English and believes his system is good for this.

Halpern is the most sanguine and vocal of the meta-assembler proponents. His work on meta-assemblers is

related to his controversial stand on natural language programs by his statement that XPOP will allow one to implement something "closely approaching" natural language. Halpern's paper [Hal 67b] is 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. Halpern also separates the study of programming languages into three parts: functional (macros), notational (change punctuation commands, etc.), and modal (assembly time executions).

C.4. Extendible Compilers—Basic Concepts

Many attempts (starting with McIlroy [McI 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 (including PL/I [IBM 66]) 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 LOOP 1 = *FOR I* ← 1 *STEP* 1 *UNTIL N* #

and later write statements like

LOOP 1 *N DO A*[1] ← 0

which would be expanded into

FOR I ← 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 (*P*1, *P*2) *TOBE*
*FOR P*1 ← 1 *STEP* 1 *UNTIL P*2 *DO ENDMACRO.*

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

LOOP(*I*, *N*) *A*[*I*] ← 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 widely 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.

C.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 work is best understood in the light of their previous TWS work, which is discussed in detail in Section III.B.2.

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 metalanguage $\mathfrak{L}_D$ (discussed in Section III.B.2) and a series of macro facilities.

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 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.

There are two kinds of syntactic macros considered together; basic features of both are *free format* and *type specification for parameters*. An example would be

LET N BE INTEGER
MACRO SQUARE N MATRIX MEANS '*ARRAY*[1:*N*, 1:*N*]'.

The advantage of free format over the conventional ⟨operator⟩, ⟨operand list⟩ format is obvious; the specification of parameter types allows conditional assembly and better error detection. The call of this syntactic macro would be set off by a special delimiter (e.g. %) and would have a detectable termination. The second approach, which avoids the use of special delimiters, is to add the macro form directly to the syntax tables of the translator. The corresponding declaration would be:

LET N BE INTEGER
SMACRO SQUARE N MATRIX AS ⟨attribute⟩ *MEANS*
'*ARRAY*[1:*N*, 1:*N*]'

where ⟨attribute⟩ is a syntactic type in the definition of the underlying language. Both *MACROS* and *SMACROS* would be implemented by substituting the descriptors (cf. Section III.B.2) of the appropriate actual parameters. Neither of these schemes presents an implementation problem in Trangen, 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 analog of the many-many macro 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 the most of its eggs in this basket. His solution is to provide a number of primitive operations (e.g. table expansions) and to point to the existence of a complete metalanguage (Trangen) behind the extendible language.

The third type of macro extension is called the computa-

tional macro. With this technique the substitutions are made in the intermediate code resulting from a declared macro. The intermediate code for the macro body is produced (with formal parameters) in advance; so this technique is restricted to constructs for which the intermediate code can be compiled 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 function for a 4 × 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 reappears in the discussion of Perlis and Galler's 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 nonprocedural descriptions.

C.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 discuss here only those dealing with extendible compilers. Other topics are treated in Section IV.C 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 a version of ALGOL [Naur 63b] 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 of ALGOL C program. The extensions are accomplished through constructs rather like Cheatham's *SMACROS* which add to the syntax tables of the translator. Because they want to do the macro processing in very sophisticated ways, Perlis and Galler allow redefinitions only in a few fixed categories. The base 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⟩ 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 Ds 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 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 problems 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.B.1), 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 to clarify these notions.

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);
```

The following definitions are then 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 and specifies that it is to be treated as a *list*. Statement (10) defines the assignment statement for assigning a list to an integer variable. The paper also includes definitions of the EVAL function and of various sequencing operators over list structures.

This example 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 texts 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 are 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 show 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 applicability. There is, in addition, a general question of how often one would want to change a high-level language; this is taken up again in Sections IV.C.

REFERENCES FOR III.C
Benn 64a, 64b Brook 60b, BroP 67, Che 64a, 66, Fer 66, Gal 67, Gar 66, GraM 65, Hal 67a,b,c, IBM 66, Lea 66, McIl 60, Mea 63, Moo 65, Star 65, Str 65, Wai 67.

IV. RELATED TOPICS AND CONCLUSIONS

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 TWSs described in Section III 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, 67] 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 the precedence technique similar to those described in Section II.B. 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 as well as data. 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. References [Ross 63, 67] are good introductions 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 III.B. 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] which is itself based on VITAL.

Most of the other applications of TWSs have been in symbol manipulation tasks of one sort or another. Some of the first applications [Scho 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 TWSs (especially 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 TWSs is in problems of format conversions. These arise in connection with large data files and in translating between closely related source languages. Once again, the syntax-directed symbol processors of Section III.A have been used the most often. These systems have also been of some use in such varied tasks as: logic design, translating geometric descriptions, simulation, and logging routines. 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 TWSs have inspired work in several others. One active area has been the syntactic description of pictures. The syntax-directed approach to picture processing seems to be increasingly popular [cf. ShaA 67, An 67], but one early worker [Nar 66] appears (in some sections of [Nar 67]) to reject this approach. The pattern matching features incorporated in the new list processing languages [Ab 66, It 66] are partially inspired by TWSs, and the related field of artificial intelligence has some syntax-directed projects underway.

The field of computational linguistics in both its theoretical and practical aspects is closely related to TWS studies. The applications here, though fewer than one would expect, have been significant. The syntactic theories of computational linguistics and TWSs both are based on the early work of Chomsky [Chom 63] and share many ideas. The implementations of English syntax (especially [Kun 62]) developed concurrently with top-down TWSs, 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 straightforward application of TWS techniques (cf. [Nap 67, Col 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 nonprocedural languages.

The last, but by no means the least of the applications considered here, is to teaching. Several of the TWSs 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 such courses can be taught without machine problems, they are much more successful when the students have easy access to the TWSs under discussion.

REFERENCES FOR IV.A
Ab 66, An 66, Brook 67a, Cas 66, Chom 65, Cla 66, Col 67, Gro 66, Hal 66, 67b, It 66, Kun 62, Mond 67, Nap 67, Nar 66, 67, Rob 66, Ross 63, 66, 67, Scho 65, ShaA 67, Th 66.

B. Formal Studies of Semantics

Computer science owes much to mathematics and is beginning to pay off that debt. Both the syntax (Section II.C) and semantics of programming languages have inspired formal treatment. In this section we discuss briefly the developments most relevant to TWSs and provide an entrée to the literature on the formal semantics of programming languages.

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 those aspects of language describable in the syntactic metalanguage under discussion. This practice has the unpleasant effect of changing the definition of syntax with each change in metalanguage.

Computer scientists trained in logic (e.g. [Tix 67]) would like us to adopt the definitions used there: any property of a string which can be described in terms of its form is syntactic. Thus, whether or not a string is a theorem in some calculus is a syntactic, though perhaps undecidable, question. This approach has not proved effective for natural language and has immediate problems in programming languages. For example, are the statements

$$X \leftarrow Y/0.0$$
$$L1\text{: } \textbf{go to } L1$$

syntactically well-formed in ALGOL? 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 of programs a decidable property.

The situation is further complicated by the fact that all major languages contain statements unparsable by the formal syntax alone. A good example is the labelled *END* construct in PL/I ([IBM 66]) and even ALGOL is still not free of such constructs [Knu 67]. Thus, in practice, syntax-directed compilers must incorporate "semantic" features in the syntactic 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 it 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 λ-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 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 metalanguage 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 metalanguages described in Section III.B., 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 in the program. A related set of developments is the attempt to define all programming languages by reduction to a single high level [Ste 66] or machine-like [Brat 61, Ste 61] language.

The approaches to formalization described above are more closely related to TWSs but are far too complex to be very useful in proofs. For those who consider proofs to be the sole end of formalization (and who are reading this paper at all) the preceding paragraph will have been considered an anathema. 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 λ-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 λ-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 [Landi 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 λ-calculus to semantics have been pursued most diligently by Landin. In a series of papers he considers relationships between programming languages (ALGOL) and an augmented λ-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 desired descriptive clarity nor maintained the tractability of λ-calculus in accordance with the original plan. The most conspicuous benefit of this work has been CPL [Cou 66], which is an extremely civilized language. There is presently an active group at MIT which is pushing this approach as far as it is ever likely to go.

Although he introduced the λ-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 $c(x,d)$—the contents of symbolic position x in state vector c—and $a(x,z,d)$—the state resulting from substituting z for x in the state vector d. He is then able to get conditional expression definitions of machine code-like operations and higher level constructs described 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]. Most recently, Painter [Pai 67] was able to prove the correctness of a "compiler."

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 (applying to the state vector) at 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 statements 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.

There have also been several approaches to formalization of semantics which lie between these extremes. One approach [Don 67] uses a version of post canonical systems [Pos 43] to describe both the syntax (including type-matching, etc.) and compilation of programs. The definitions attained appear reasonable, but it must be seen whether they are of any use in compilation or in proofs. Another adaption of existing mathematics has been attempted by van Wijngarten and de Bakker [Bak 65, 67,

Wij 66]. They try to reduce the complexity of the semantic model by using a universal machine which can read and interpret simple and powerful rules. The rules are used cumulatively to define what amounts to a Markov (Mark 59] algorithm description of the source language. The difficulty is that the formalism is so primitive that the description of ALGOL is literally a book [Bak 67] and neither proofs nor insight seem likely to result.

There have also been a number of attempts to define abstract machines to carry out the semantics of programming languages. The most ambitious of these is the RASP of [Elg 64], but this work has apparently not been continued. An interesting recent paper by [Nar 67] contains a formalization which combines many features discussed above. Narasimhan defines languages and machines in closely related formalisms involving flow charts, functions on state variables, declarations, and selection and addressing operators. The approach seems promising, but there are no concrete results yet, and one of his basic assumptions is highly questionable. His requirement that a translator be as simple as possible leads Narasimhan to the conclusions that syntax and recursion are of no value. He also states (without references) that TWS efforts have all failed and interest in the field is waning.

Perhaps our description of the work in formal semantics has been sufficiently shallow to be misleading. Most of these efforts have their comrades and fellow travelers, and the development is richer than we suggest; the references at the end of this section should cover all major trends related to TWSs. The impact of formal semantics, especially the proof-oriented kind, has been limited to a few isolated insights. No work has had 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. Minsky and Papert [Min 67] have expressed a similar belief:

> Good theories rarely develop outside of the context of well-understood real problems, and it is perhaps not surprising that work directed sharply toward obtaining an "abstract theory of computation," e.g. the mathematical developments in current theories of recursive functions, automata, formal linguistics and the like, has been disappointing in the extent of its practical illumination, despite its often elegant mathematical quality.

REFERENCES FOR IV.B
Bak 65, 67, Braf 63, Burg 64, Burs 65, Cal 62, Chu 51, Cul 67, Cur 58, Don 67, Elg 67, Flo 67, Ir 61, 63b, Knu 67, Landi 63, 65, 66, Luc 65, McCar 62a, 67, Min 67, Nar 67, Org 67, Pai 66, Rig 62, Ste 64, Tars 56, Tix 67, Wir 66c, Zar 67, Zem 66.

C. Summary and Research Problems

The TWSs 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 III has its peculiar strengths and weaknesses and a preferred problem domain. After summarizing the relations between the various categories, we suggest a number of fruitful areas for future research.

The automatic constructors of recognizers, described in Section II.B, 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 III.A. These TWSs 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 Section III.C.1–3 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 processors of Section III.A.

The compiler-compilers of Section III.B are the high point in the evolution of specialized TWSs. Although specialization has made them by far the most useful for compiler writing it 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 such systems to become even more specialized. There is, of course, a risk of overspecialization, and some TWS workers feel that a more general syntax-directed processor like COGENT (Section III.B.3) will have greater survival value.

The work on extendible compilers (Section III.C.4–6) is more recent and is 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. Ideally, one would like to be able to extend a language in macro fashion and later incorporate the extensions efficiently in the compiler. The CC system (Section III.B.3) has both facilities, and although it does not solve the problem, it would be a good facility for experimenting with solutions.

None of the TWSs 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. There are now enough available techniques to satisfy a great variety of possible TWS requirements and the outstanding problems are in specific topics.

The syntactic aspects of TWSs have received considerable attention and have fewer outstanding questions. 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, Don 67]. 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 in automatic recognizer constructing programs. One would like the system to use efficient techniques wherever possible and automatically move to slower, more general schemes rather than quit when the going gets rough. In addition, the problem of automatic recovery from syntax errors could use considerably more attention [EvA 63, Ir 65].

There has been much less work on the postsyntactic aspects of TWSs. Three basically different approaches to this "semantics" problem are: first, to provide a general purpose list-processing or other symbol manipulation capability (cf. Section III.A); second, to provide a number of data structures and built-in routines especially designed for compiler-writing (cf. Section III.B); and third, to combine these facilities with code brackets and a machine independent specification of output (Section III.B.1). By making use of macros and subroutines, either of the first two techniques can look, to the average user, like the 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 TWSs 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 metaproblems. In the remainder of this section, we 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 this section for each subject are either very recent, or comprehensive, or they have already been used as references 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 discuss it as one aspect of nonprocedural programming. The accepted definition of "nonprocedural," like that of "semantics," has yet to appear. A programming system will be called nonprocedural to the extent that it makes selections and rearrangements of procedural steps in response to some higher order problem statement.

Nonprocedural 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 nonprocedural 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 analog 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 nonprocedural features of a general sort to the extendible compiler discussed in Section III.C.5. 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 [Com 66] and, of course, COBOL [Samm 61], but these make only superficial contact with the problem. The recent interest in conversational and nonprocedural programming languages along with the syntax-directed natural language systems (cf. Section IV.A) should lead to significant interchange of ideas.

There are several open problems concerning the connection between TWSs 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 have always been small groups interested in "environmental" questions for compilers [Leo 66],

but they had little effect before the time sharing revolution. The (hoped for) availability of large multiaccess timesharing systems gives rise to several additional research problems related to TWSs. The main task of any large timesharing 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 time 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.

Two other problems relating to executive systems are mentioned 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, except for Project MAC. There have been some good symbolic dump facilities in particular batch-made compilers, but there have not found their way into print or into TWSs. 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 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 areas of computer science. 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 mention a few aspects connected with TWSs 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 Section 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 universal 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 nonprocedural 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 embarrassingly long paper. We have attempted to show how a considerable 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 IV.C

Theory of machine instructions: Brat 61, Bur 64, Car 62, Don 67, Elg 64, Gil 67, Maur 65, Nar 67, Ste 61.
Parallelism: Dij 65, Kar 66, Kuc 67, Mark 67, Shed 67, Sto 67.
Code selection and enhancement, general references: Ar 61, Grie 65, Hill 62, Hor 66, Lucc 67.
Nonprocedural languages: Che 66, Gall 67, Ri 66, Rov 67, Schl 67, Sib 61, Wil 64b, You 65.
Natural language processing: Bar 64, Chom 65, Col 67, Cra 66, Hal 66, Int 63, Kun 62, Nap 67, Nar 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, Sta 67, Wir 66b.

Acknowledgments. The authors would like to thank John Reynolds and the many other people who contributed suggestions and criticism.

RECEIVED JUNE 1967; REVISED OCTOBER 1967

V. BIBLIOGRAPHY

Ab 66 ABRAHAMS, P W. The LISP 2 programming language and system. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 661–676.

An 66 ANDERSON, R. H. A two dimensional syntax for mathematical notation. Unpublished report. Harvard U., Cambridge, Mass., 1966.

Ar 61 ARDEN, B. W., GALLER, B. A., AND GRAHAM, R. M. An algorithm for equivalence declarations. *Comm. ACM 4* (July 1961), 310–314.

Ar 66 ——, ——, AND ——. *Michigan Algorithmic Decoder*. U. of Michigan Press, Ann Arbor, Mich., 1966.

Bac 57 BACKUS, J. W. ET AL. The FORTRAN automatic coding system. Proc. Western Joint Comput. Conf. 1957, pp. 188–198.

Bac 59 BACKUS, J. W. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. Proc. Internat. Conf. on Information Processing, UNESCO, 1959, pp. 125–132.

Bak 65 DE BAKKER, J. Formal definition of algorithmic languages. MR74, Mathematisch Centrum, Amsterdam, May 1965.

Bak 67 ——. Formal definition of programming languages. Mathematisch Centrum Tract No. 16, Amsterdam, 1967.

Bar 64 BAR HILLEL, Y. *Language and Information*. Addison-Wesley Publishing Company, Inc., Reading, Mass., 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. AFIPS 1963 SJCC, Vol. 23, pp. 169–177.

Ben 64a BENNETT, R. K., AND KVILEKVAL, A. SET, self extending translator. Data Processing, Inc., March 1964.

Ben 64b ——, 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.

Brook 60a BROOKER, R. A., AND MORRIS, D. An assembly program for a phrase structure language. *Comput. J. 3* (1960), 168–174.

Brook 60b ——, AND ——. Some proposals for the realization of a certain assembly program. *Comput. J. 3* (1960), 220–231.

Brook 61 ——, AND ——. A description of Mercury Autocode in terms of a phrase structure language. *Annual Review in Automatic Programming, Vol. 2*, 1961, pp. 29–66.

Brook 62a ——, AND ——. A general translation program for phrase structure languages. *J. ACM 9* (Jan. 1962), 1–10.

Brook 62b —— ET AL. Trees and routines. *Comput. J. 5* (1962), 33–47.

Brook 63 —— ET AL. The compiler-compiler. *Annual Review in Automatic Programming, Vol. 3*, 1963, p. 229.

Brook 67a ——, MORRIS, D., AND ROHL, J. S. Compiler-compiler facilities in Atlas Autocode. *Comput. J. 9* (1967), 350–352.

Brook 67b ——, ——, AND ——. Experience with the compiler-compiler. *Comput. J. 9* (1967), 345–349.

Brook 67c ——, AND ROHL, J. S. Simply partitioned data structures: The compiler-compiler reexamined. *Machine Intelligence I*, Collins and Michie, (Eds.), Oliver and Boyd, London, 1967.

BroP 67 BROWN, P. J. The ML/I macro processor. *Comm. ACM 10* (Oct. 1967), 618–623.

BroS 63 BROWN, S. A., DRAYTON, C. E., AND MITTMAN, B. A description of the APT language. *Comm. ACM 6* (Nov. 1963), 649–658.

Burg 64 BURGE, W. H. The evaluation, classification and interpretation of expressions. Proc. ACM 19th Nat. Conf., 1964, p. A1.4.

Burk 65 BURKHARDT, W. Universal programming languages and processors. Proc. AFIPS 1965 FJCC, Vol. 27, pp. 1–21.

Burs 65 BURSTELL, R. M. Some aspects of CPL sematics. No. 3, Experimental Programming Rpts., Edinburgh U., Edinburgh, 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 Congress, Munich, 1962, pp. 514–515.

Car 63 ——. 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, Computer Associates, Inc., Wakefield, Mass., 1964.

Che 64b —— ET AL. Preliminary description of the translator generator system, II. CA-64-1-SD, Computer Associates, Inc., Wakefield, Mass., 1964.

Che 64c ——, AND SATTLEY, K. Syntax directed compiling. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 31–57.

Che 65 ——. The TGS-II translator-generator system. Proc. IFIP Congress, New York, 1965, pp 592–593.

Che 66 ——. The introduction of definitional facilities into higher level programming languages. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 623–637.

Chom 63 CHOMSKY, N. Formal properties of grammars. In *Handbook of Mathematical Psychology, Vol. 2*, Luce, Bush and Galanter (Eds.), John Wiley & Sons, Inc., 1963, pp 323–418.

Chom 65 ——. *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 U. Press, Princeton, N. J., 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* (Feb. 1967), 82–86.

Col 67 — COLES, S. Syntax-directed interpretation of natural language. Doctoral dissertation. Carnegie-Mellon Inst., Pittsburgh, Pa., 1967.

Conn 66 — CONNORS, T. B. ADAM—a generalized data management system. Proc. AFIPS 1966 SJCC, Vol. 28, pp. 193–203.

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

Cou 66 — COULOURIS, G. F., AND GOODEY, T. J. The CPL1 system manual. PID12/GFC, Inst. of Comput. Sci., U. of London, London.

Cou 67 — ——. The compiler processor project. Internal Rep., Imperial College, London, April 1967.

Cra 66 — CRAIG, J. A., BEREZNER, S. C., CARNEY, H. C., AND LONGYEAR, C. R. DEACON: Direct English Access and CONtrol. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 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, pp. 75–82.

Cur 58 — CURRY, H. B., AND FEYS, R. *Combinatory Logic, Vol. I.* North-Holland Publishing Co., 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, Vol. 3*, 1963, pp. 27–42.

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

Don 67 — DONOVAN, J. J., AND LEDGARD, H. F. Canonic systems and their application to programming languages. Mem. MAC-M-347, Project MAC, MIT, Cambridge, Mass., April, 1967.

Ear 65 — EARLEY, J. C. Generating a recognizer for a BNF grammar. Computation Center Rep., Carnegie Inst. of Tech., Pittsburgh, Pa., 1965.

Ear 67 — ——. An LR(K) parsing algorithm. Carnegie Inst. of Tech., Pittsburgh, Pa., 1967, (mimeo).

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 — ——. Generation of parsing algorithms for Chomsky 2-type languages. 6401, Mathematisches Institut der Technischen Hochschule, Munich, 1964.

Elg 64 — ELGOT, C. C., AND ROBINSON, A. Random-access stored-program machines, an approach to programming languages. *J. ACM 11* (Oct. 1964), 365–399.

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, Vol. 4*, 1964, pp. 87–124.

EvT 66 — EVANS, T., AND DARLEY, D. On-line debugging techniques: a survey. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 37–50.

Feld 64 — FELDMAN, J. A. A formal semantics for computer oriented languages. Carnegie Inst. of Tech., Pittsburgh, Pa., 1964.

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

Feld 67 — ——, AND CURRY, J. The compiler-compiler in a time sharing environment. Lecture notes on Advanced Computer Organization, U. of Michigan, Ann Arbor, Mich., 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 Rep., Carnegie Inst. of Tech., Pittsburgh, Pa., 1967.

Flo 61 — FLOYD, R. W. A descriptive language for symbol manipulation. *J. ACM 8* (Oct. 1961), 579–584.

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

Flo 62b — ——. On the nonexistence of a phrase structure grammar for ALGOL-60. *Comm. ACM 5* (Sept. 1962), 483–484.

Flo 63 — ——. Syntactic analysis and operator precedence. *J. ACM 10* (July 1963), 316–333.

Flo 64a — ——. Bounded context syntactic analysis. *Comm. ACM 7* (Feb. 1964), 62–67.

Flo 64b — ——. The syntax of programming languages—a survey. *IEEE Trans. EC13*, 4 (Aug. 1964), 346–353.

Flo 67 — ——. 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), 204–219.

Gar 64 — GARWICK, J. V. GARGOYLE, a language for compiler writing. *Comm. ACM 7* (Jan. 1964), 16.

Gar 66 — ——, BELL, J., AND KRIDER, L. The GPL language. TER-05, Control Data, Palo Alto, Calif., 1966.

Gea 65 — GEAR, C. W. High speed compilation of efficient object code. *Comm. ACM 8* (Aug. 1965), 483–488.

Gil 66 — GILBERT, P. On the syntax of algorithmic languages. *J. ACM 13* (Jan. 1966), 90–107.

Gil 67 — ——, AND MCLELLAN, W. G. Compiler generation using formal specification of procedure-oriented and machine languages. Proc. AFIPS 1967 SJCC, Vol. 30, pp. 447–455.

Gin 66a — GINSBURG, S. *The Mathematical Theory of Context Free Languages.* McGraw-Hill, Inc., New York, 1966.

Gin 66b — ——, AND GREIBACH, S. Deterministic context free languages. *Inf. Contr. 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., Pittsburgh, Pa., 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 — ——. 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. ACM 20th Nat. Conf., 1965, pp. 409–420.

GraR 64 — GRAHAM, R. M. Bounded context translation. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 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 Congress, Munich, 1962, pp. 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 ——. The use of transition matrices in compiling. Tech. Rpt. CS 57, Computer Science Dept., Stanford U., Stanford, Calif., March 1967 and *Comm ACM 11* (Jan. 1968), 26–34.

Grie 67b ——. Internal notes on the compiler writing system. Computer Science Dept., Stanford U., Stanford, Calif., 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 metalanguage without metaphysics. Proc. AFIPS 1964 FJCC, Vol. 26, pp. 57–68.

Hal 66 ——. Foundations of the case for natural-language programming. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 639–649.

Hal 67a ——. Toward a general processor for programming languages. *Comm. ACM 11*, (Jan. 1968), 15–26.

Hal 67b ——. Foundations of the case for natural language programming. *IEEE Spectrum* (March 1967), 140–149.

Hal 67c ——. *A manual of the XPOP programming system*. Electronic Sciences Lab. Lockheed Missiles & Space Company, Palo Alto, Calif., March 1967.

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. Notes from NATO summer school lectures, 1966.

Hor 66 HORWITZ, L. P., KARP, R. M., MILLER, R. E., AND WINOGRAD, S. Index register allocation. *J. ACM 13* (Jan. 1966), 43–61.

Hus 62 HUSKEY, HARRY D. Languages for aiding compiler writing (panel discussion). Proc. 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 Congress, Munich 1962, pp. 527–528.

Ing 66 ——. *A Syntax Oriented Translator*. Academic Press, Inc., 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 ——. The structure and use of the syntax-directed compiler. *Annual Review in Automatic Programming, Vol. 3*, 1963, pp. 207–227.

Ir 63b ——. Towards more versatile mechanical translators. *AMS Symposium in Appl. Math. 15*, 1963, pp. 41–50.

Ir 64 ——. Structural connections in formal languages. *Comm. ACM 7* (Feb. 1964), 67–71.

Ir 65 ——. 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 1966 SJCC, Vol. 28, pp. 241–252.

Kar 66 KARP, R. M., AND MILLER, R. E. Properties of a model for parallel computations: determinacy, termination, queueing. *SIAM J.* (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. *Comput. J.* (1967).

Kir 66 KIRKLEY, C., AND RULIFSON, J. LOTS, a syntax-directed compiler. Internal Rep., Stanford Research Inst., Menlo Park, Calif., May 1966.

Knu 62 KNUTH, D. E. History of writing compilers. Proc. ACM 17th Natl. Conf., 1962, pp. 43, 126.

Knu 65 ——. On the translation of languages from left to right. *Inf. Contr. 8* (Oct. 1965), 607–639.

Knu 67 ——. The remaining trouble spots in ALGOL 60. *Comm. ACM 10* (Oct. 1967), 611–618.

Kuc 67 KUCK, D. Programming the ILLIAC IV. Talk given at AFIPS 1967 SJCC. 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, pp. 306–311.

Lande 62 LANDEN, W. H., AND WATTENBURG, W. H. On the efficient construction of automatic programming systems. Proc. ACM 17th Natl. Conf., 1962, p. 91.

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

Landi 65 ——. A correspondence between ALGOL 60 and Church's λ-notation. *Comm. ACM 8* (Feb. and March 1965), 89–101, 158–167.

Landi 66 ——. 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. EC, 13* (Aug. 1964), 354–362.

Lang 64 LANGMAACK, H., AND EICKEL, J. Präzisierung der begriffe Phrasenstruktur und strukturelle Mehrdeutigkeit in Chomsky-Sprachen. Rep. No. 6414, Rechenzentrum der Technischen Hochschule, Munich, 1964.

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

Lea 66 ——. Syntax macros and extended translation. *Comm. ACM 9* (Nov. 1966), 790–793.

Leo 66 LEONARD, G., AND GOODROE, J. More extensible machines. *Comm. ACM 9* (March 1966), 183–188.

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.

Lucc 67 — LUCCIO, F. A comment on index register allocation. *Comm. ACM 10* (Sept. 1967), 572–574.

Mark 61 — MARKOV, A. A. *Theory of Algorithms.* US Bureau of Standards, OTS 60-51085, available from Clearing House, Springfield, Va.

Mar 67 — MARTIN, D., AND ESTRIN, G. Models of computations and systems. *J. ACM 14* (April 1967), 281–294.

Mas 60 — MASTERSON, K. S. Compilation for two computers with NELIAC. *Comm. ACM 3* (Nov. 1960), 607–611.

Maur 65 — MAURER, W. A theory of computer instructions. Mem. MAC-M-262, Project MAC, MIT Cambridge, Mass., Sept. 1965.

May 61 — MAYOH, B. H. Letter to the editor correcting Ir 61. *Comm. ACM 4* (June 1961), 284.

McCar 62a — MCCARTHY, J. Toward a science of computation. *Information Processing 62* (IFIP Congress), Popplewell (Ed.). North-Holland Publishing Co., Amsterdam, 1962, pp. 21–28.

McCar 62b — —— ET AL. LISP 1.5 programmers manual. Computation Lab Report, MIT (1962).

McCar 67 — ——, 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. ACM 20th Natl. Conf., 1965, pp. 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 U., Stanford, Calif., Aug. 1966.

Mea 63 — MEALY, G. A generalized assembly system. RM-3646-PR Rand Corp., Santa Monica, Calif., Aug. 1963.

Met 64 — METCALFE, H. H. A parametrized compiler based on mechanical linguistics. *Annual Review in Automatic Programming, Vol. 4*, 1964, pp. 125–165.

Min 67 — MINSKY, M. L., AND PAPERT, S. Perceptions and pattern recognition. Mem. MAC-M-358, Project MAC, MIT, Cambridge, Mass., Sept. 1967.

Mond 67 — MONDSCHEIN, L. VITAL compiler-compiler reference manual. TN 1967-1, Lincoln Lab., MIT, Lexington, Mass., Jan. 1967.

Moo 65 — MOOERS, C., AND DEUTSCH, L. P. TRAC, a text handling language. Proc. ACM 20th Natl. Conf., 1965, pp. 229–246.

Mor 67 — MORRIS, D., AND WILSON, I. A system program generator. Computer Science Dept., U. of Manchester, Manchester, 1967

Nap 67 — NAPPER, R. B. E. The third-order compiler. A context for free man-machine communication. *Machine Intelligence I*, Collins and Michie (Eds.). Oliver and Boyd, London, 1967.

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

Nar 67 — ——. Programming languages and computers: a unified meta theory. In *Advances in Computer 8*. Academic Press, Inc., New York, 1967, Chap. 5.

Naur 60 — NAUR, P. (Ed.) Report on the algorithmic language ALGOL 60. *Numer. Math. 2* (1960), 106–136; *Comm. ACM 3* (May 1960), 299–314.

Naur 63a — ——. Documentation problems: ALGOL 60. *Comm. ACM 6* (March 1963), 77–79.

Naur 63b — ——. Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (Jan. 1963), 1–17; *Numer. 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.

Nor 63 — NORTHCOTE, R. S. The structure and use of a compiler-compiler system. Proc. Australian Comput. Conf., Dec. 1963.

Op 62 — OPLER, A. "Tool"—a processor construction language. Proc. IFIP Congress, Munich, 1962, pp. 513–514.

Orch 66 — ORCHARD-HAYS, WILLIAM. Multilevel operating systems. *Comm. ACM 9* (March 1966), 189–190, (abstract only).

Org 67 — ORGASS, R. J. A mathematical theory of computing machine structure and programming. RC1863, IBM, Yorktown Heights, New York, 1967.

Pai 66 — PAINTER, J. Semantic correctness of a compiler for an ALGOL-like language. AI Rep. No. 44. Stanford U., Stanford, Calif., 1966.

Par 61 — PARIKH, R. J. Language generating devices. Quarterly progress rep. no. 60, Research Lab. 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 Congress, Munich, 1962, pp. 493–497.

Plas 66 — PLASKOW, J., AND SCHUMAN, S. The TRANGEN system on the M460 computer. AFCRL-66-516 (July 1966).

Pos 43 — POST, E. L. Formal reductions of the general combinatorial decision problem, *Am. J. Math. 66* (1943), 197–217.

Pra 65 — PRATT, T. W. Syntax-directed translation for experimental programming languages. TNN-41, Computation Center U. of Texas, Austin, Texas, 1965.

Pra 66 — ——, AND LINDSAY, R. K. A processor-building system for experimental programming language. Proc. AFIPS 1966 FJCC, Vol. 29, pp. 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, Inc., London, 1964.

Rey 65 — REYNOLDS, J. C. An introduction to the COGENT programming system. Proc. ACM 20th Natl. Conf., 1965, pp. 422–436.

Ri 66 — RICE, J., AND ROSEN, S. NAPSS, numerical analysis and problem solving system. Proc. ACM 21st Natl. Conf., 1966, pp. 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), pp. 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), 403–414.

Ros 64b — ——. Programming systems and languages. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 1–15.

Ross 63 — Ross, D., and Rodriguez, J. Theoretical foundations of the computer aided design system. Proc. AFIPS 1963 SJCC, Vol. 23, pp. 305–322.

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

Ross 66 — ——. AED bibliography. Mem. MAC-M-278-2, Project MAC, MIT Cambridge, Mass., Sept. 1966.

Ross 67 — ——. The AED approach to generalized computer-aided design. Proc. ACM 22nd Natl. Conf. 1967, pp. 367–385.

Rov 67 — Rovner, P., and Feldman, J. An associative processing system for conventional digital computers. TN 1967-19, Lincoln Lab., MIT, Lexington, Mass., April 1967.

Rut 62 — Rutishauser, H. Panel on techniques for processor construction. Proc. IFIP Congress, Munich, 1962, pp. 524–531.

Sam 60 — Samelson, K., and Bauer, F. L. Sequential formula translation. *Comm. ACM 3* (Feb. 1960), 76–83.

Sam 62 — ——. Programming languages and their processing. Proc. IFIP Congress, Munich, 1962, pp. 487–492.

Samm 61 — Sammet, J. E. A definition of COBOL 61. Proc. ACM 16th Natl. Conf., 1961.

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

Schm 63 — Schmidt, L. Implementation of a symbol manipulator for heuristic translation. Proc. ACM 18th Natl. Conf., 1963.

Sch 64 — Schneider, F. W., and Johnson, G. D. META-3; A syntax-directed compiler writing compiler to generate efficient code. Proc. ACM 19th Natl. Conf. 1964, p. D1.5-1.

Scho 65 — Schorr, H. Analytic differentiation using a syntax directed compiler. *Comput. J. 7* (Jan. 1965), 290–298.

Schor 64 — Schorre, D. V. META-II: A syntax-oriented compiler writing language. Proc. ACM 19th Natl. Conf., 1964, p. D1.3.

Schü 63 — Schützenberger, M. P. Context-free languages and pushdown automata. *Inf. Contr. 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.

ShaA 67 — Shaw, A. A formal description and parsing of pictures. Doctoral dissertation, Computer Science Dept., Stanford U., Stanford, Calif., 1968.

Shaw 63 — Shaw, C. J. A specification of JOVIAL. *Comm. ACM 6* (Dec. 1963), 721–735.

Shed 67 — Shedler, G. Parallel numerical methods for the solution of equations. *Comm. ACM 10* (May 1967), 286–291.

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

Sta 67 — Standish, T. A. A data definition facility for programming languages. Computer Science Rep., Carnegie Inst. of Tech. Pittsburgh, Pa., May 1967.

Star 65 — Stark, R. *ALTEXT—multiple purpose language.* Lockheed Missiles & Space Company Tech. Rep. 6-75-65-15, March, 1965.

Ste 61 — Steel, T. B. A first version of UNCOL. Proc. Western Joint Comput. Conf., 1961, pp. 371–378.

Ste 66 — —— (Ed.) Formal language description languages for computer programming. Proc. IFIP Conf., Baden, Sept. 1964, North-Holland Publishing Co., Amsterdam, 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. *Comput. 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 1966 FJCC, Vol. 29, pp. 349–356.

Tix 67 — Tixier, V. Recursive functions of regular expressions in language analysis. Tech. Rpt. CS 58, Computer Science Dept., Stanford U., Stanford, Calif., March 1967.

Tro 67 — Trout, R. G. A compiler-compiler system. Proc. ACM 22nd Natl Conf., 1967, pp. 317–322.

Wai 67 — Waite, W. A language independent macro processor. *Comm. ACM 10* (July 1967), 433–441.

War 61 — Warshall, S. A syntax directed generator. Proc. Eastern Joint Comput. Conf., 1961, pp. 295–305.

War 64 — ——, and Shapiro, R. M. A general-purpose table-driven compiler. Proc. AFIPS 1964 SJCC, Vol. 25, pp. 59–65.

Weg 62 — Wegner, P. (Ed.) *Introduction to Systems Programming.* Academic Press, Inc., 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., Amsterdam, 1966, pp. 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, Vol. 4*, 1964, pp. 1–48.

Wil 64b — ——. 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 U., Stanford, Calif., Dec. 1966.

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

Wir 66c — ——, 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. Proc. IFIP Congr., New York, 1965, pp. 622–623.

You 65 — Young, J. W., Jr. Nonprocedural languages. 7th Ann. ACM Tech. Symp., S. Calif. Chapter, March 1965.

Zar 67 — Zara, R. F. A semantic model for a language processor. Proc. ACM 22nd Natl. Conf., 1967, pp. 323–339.

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