SYNTAX-DIRECTED COMPILING
T. E. Cheatham, Jr., and Kirk Sattley
Computer Associates, Inc.
Lakeside Office Park
Wakefield, Massachusetts

INTRODUCTION

This paper is primarily concerned with the analysis of source statements in a programming language, although some of the ideas and techniques may be applicable to the analysis of source statements in a natural language. We are particularly concerned with those techniques which might be classed as predictive; the companion paper by Graham is concerned with other ("nonpredictive") techniques of analysis. Very broadly the techniques we will discuss operate as follows: Given a set of rules (Syntax Specification) for forming allowable constructs, eventually resulting in a statement (or sentence, word, program, etc.) of a language, we analyze a source statement in that language by guessing, or predicting, how the statement is constructed and either verifying that this is the case or backing up to try again, assuming some other method of construction. We keep a "history" of our attempts and when we have determined the exact way in which the statement is constructed we can use this "history" of its construction for further processing of the components of the statement.

We will be concerned, secondarily, with the synthesis of machine coding, given an analysis of a source statement. We do not, however, discuss in any detail the difficult (and, at this point, not terribly well understood) problems of synthesizing highly efficient coding. Reference [1] contains a brief discussion of this problem.

We are concerned hardly at all with the extremely important and often neglected problems of the environment in which a compiler or code resulting from a compiler is to operate. Reference [3] sketches our position in this matter.

The phrase "syntax-directed" in the title refers to the method by which the compiler is given the syntactic specification of the language it is to compile. That is, rather than having the syntactic structure of the language reflected in the actual encoding of the compiler algorithm, a "syntax-directed" compiler contains (or uses, as parametric data) a relatively simple and direct encoding of the syntactic structure of the language, for example, as it might be expressed in Backus Normal Form. By "simple and direct encoding," we mean, for instance, numerical codes for the distinct syntactic types of the language, and direct pointers representing the relations of concatenation and alternative choice, plus perhaps some sorting.

This paper is not intended as a review or critique of syntax-directed compilers or compiler techniques nor have we presented a comprehensive bibliography on the subject. Rather, our purpose is tutorial—to present in as straightforward a manner as possible the essential ideas of syntax-directed compilers. Unfortunately, there is, at the present time, no completely adequate review paper on the subject; Floyd does include a rather complete bibliography.
Our presentation commences with a discussion of syntax and the syntactic specifications of languages—programming languages in particular. We then discuss techniques for encoding the syntax into tables and develop a simple algorithm, the ANALYZER, which can perform a syntactic analysis of source material, using this tabled syntax specification as data. From this we proceed to a discussion of the generation or synthesis of code from the results of the analysis. Finally, we discuss several problems and limitations. Certain problems of syntactic specification and some modifications of the schemes we describe in the body of the paper have been discussed in an appendix.

SPECIFICATION OF SYNTAX

Several essentially equivalent formalisms for the representation of syntax have been developed. These include such things as

- Post Production Systems, developed by the logician Emil Post during the 1940's as a tool in the study of Symbolic Logic;
- Phrase Structure Grammars, developed by the linguist Noam Chomsky during the 1950's as a tool in the study of natural languages; and
- Backus Normal Form, developed by the programmer John Backus during the late 1950's as a tool in the description of programming languages.

We shall use here a formalism most similar to Backus's.

A syntactic specification of a language is a concise and compact representation of the structure of that language, but it is merely that—a description of structure—and does not by itself constitute a set of rules either for producing allowable strings in the language, or for recognizing whether or not a proffered string is, in fact, an allowable string.

However, rules can be formulated to produce, or recognize, strings according to the specification. In a "syntax-directed" compiler it is an algorithm which performs the recognition of allowable input strings, and it does this by using (an encoding of) the Syntax Specification as data. In this paper, we shall call such an algorithm an (or the) "Analyzer."

In order to discuss the structure of the language, we give names to classes of strings in the language—we call these names (or the classes they denote) "Syntactic Types." Some of the classes of interest consist of single characters of the source alphabet: these we call "Terminal Types," and specifically "Terminal Characters"; to talk about any particular one, we will merely display the character. Most of the classes, though, are more complicated in structure and are defined in terms of other classes; these we call "Defined Types," and to designate one, we choose a mnemonic name for the class and enclose it in the signs '<' and '>'.

Basic Syntax Specification

Rather than proceed immediately to a discussion of Backus Normal Form, we shall first define a simple form of Syntax Specification—the Basic Specification. This consists of a set of "Simple Type Definitions" (meaning, not that the Types are simple, but the Definitions are). A Simple Type Definition consists of the name of a Defined Type, followed by the curious sign '::=' followed by a sequence of Syntactic Types, Defined or Terminal. An example, taken from the Syntax I—soon to be discussed—would be:

<assignment>::=<variable>:=<arithexpr>

The Defined Type on the left of the '::=' is called The Defined Type of the Definition; and the Definition is said to be a Definition of its defined type. In general—even for the more complicated forms of Type Definitions yet to come—we shall call the right-hand side of the Definition the "Definiens." Any sequence of type designators appearing in a Definiens is called a "Construction," and each type designator within the Construction is a "Component" of the Construction. So, the above example is a Definition of the Defined Type <assignment>; its Definiens is a Construction with three components, which are, in the order of their appearance, the Defined Type <variable>, the Terminal Character '=' and the Defined Type <arithexpr>.

A Simple Type Definition of this sort states that, among the strings of the source language belonging to the Defined Type, are those which are concatenations of substrings—as many
substrings as there are components of its (Simple) Definiens—such that each substring (in order of concatenation) belongs to the Syntactic Type named by the corresponding Component (in order of appearance in the Definiens).

Applied to our example: A source string belongs to the class \texttt{<assignment>} (or, for short, \textit{is an <assignment>}) if it can be broken into three consecutive substrings, the first of which is a \texttt{<variable>}, the second of which is the single character \texttt{==}, and the third of which is an \texttt{<arith expr>}.

If we were interested in using Syntax Specifications as "generative grammars"—that is, if we were writing an algorithm to use a Syntax Specification to produce samples of strings of the language, we would write something which, applied to our example, would have the effect of: "if you wish to produce an \texttt{<assignment>}, then: first choose any "definition of \texttt{<variable>} and produce a string according to that definition, then, second write down the character \texttt{==}, then third produce an \texttt{<arith expr>} according to any definition of that type; then you have produced an \texttt{<assignment>}".

Thus, the use of a (Basic) Syntax Specification as a generative grammar is quite straightforward. The inverse problem—using the Syntax Specification as a "recognition grammar"—is, like many inverse problems, rather more involved. In our opinion, the fundamental idea—perhaps "germinal" would be a better word—which makes syntax-directed analysis by computer possible is that of \textit{goals}: a Syntactic Type is construed as a goal for the Analyzer to achieve, and the Definiens of a Defined Type is construed as a recipe for achieving the goal of the Type it defines. It is this use of goals which leads to another description of analysis techniques of this kind—"predictive analysis": setting up the recognition of a particular Syntactic Type as a goal amounts to predicting that an instance of that type will be found. Needless to say, this use of the term "goal" is not to be confused with the "goal-seeking behavior" of "artificial intelligence" programs or "self-organizing systems." However, when we come to specifying the algorithm for selecting particular goals in a particular order, we reach the point at which the several existing syntax-directed techniques diverge. Our purpose in this section on "Basic Syntax Specification" is to lay a foundation common to the principal different applications of the technique; hence, if we try to "picture" the use of a Syntax Specification as a recognition grammar, as we pictured its use as a generation grammar in the preceding paragraph, the most generally valid statement we can make is:

We can say that we have recognized an occurrence of a given Syntactic Type (at a given position in the source string) if one of the two following conditions obtains:

1. The Syntactic Type is a Terminal Character, and the character at the given position in the source string is exactly that Terminal Character;
2. The Syntactic Type is a Defined Type, and for some one of its (Simple) Definitions, we have already recognized concatenated occurrences of the Components of that Definiens, in the stated order, the first of which occurs at the given position.

In order for the set of Simple Type Definitions to constitute a useful Syntax Specification, it should satisfy some conditions.

(C1) Any Defined Type which occurs as a Component in any Definiens must also occur as the Defined Type of some definition.

The desirability of this "completeness condition" is fairly obvious—it will be very difficult to recognize a Defined Type if the Analyzer has no Definition of that Type. Of course, it is possible that the Analyzer may never be asked to find an instance of this Type, but then all the Definitions which included it as a Component would also be superfluous.

(C2) Every Defined Type must ultimately be constructible entirely out of Terminal Characters.

This "connectedness condition" is designed to prevent a cycle of definitions which it is impossible to break out of—that is, if a Defined Type is defined \textit{only} in terms of Defined Types,
each of which in turn is defined only in terms of Defined Types, etc. Of course, it will be true that there will be cycles within the act of definitions, and these cycles may be traversed arbitrarily many times; but there must exist some point in each cycle where an alternative definition of one of the types exists. It is probably sufficient to restate condition (C2) in the following fashion:

A Terminal Type will be said to be “grounded.” A Defined Type is grounded if it has at least one Definition, all of whose Components are grounded; then

(C2') Every Defined Type must be grounded.

(C3) There must exist exactly one Defined Type which does not appear as a Component in any Definiens (except possibly its own). This Type is called the “Starting Type” of the Syntax Specification.

The Starting Type represents the “largest” construction allowable under the Specification — e.g., “sentence,” or perhaps “paragraph,” in natural language applications, or usually “program” in compiler applications. If there is no Starting Type, the Analyzer, quite literally, will not know where to begin.

Let us note here in passing that there is a property of Syntax Specifications which is of great importance to theoreticians in this field, and to people who are designing new languages or trying to construct Specifications for existing complex languages, but which is irrelevant to the problem of programming a syntax-directed Analyzer. This is the question of “structural ambiguity”—does the Syntax Specification permit a particular source-language string to be correctly analyzed in two different fashions? A simple example, taken from natural language (with apologies to Oettinger) is: “Time flies incessantly.” This is certainly an English sentence—but is it a metaphorical declarative sentence, or a terse imperative? In the case of an Analyzer algorithm on a computer, only one thing is done at a time—if the Analyzer is asked to find an instance of an Ambiguous Syntactic Type, it must try one of the possible definitions first; if that definition succeeds, the Analyzer is satisfied, and the other definitions are not considered. This is not to say that an Analyzer, one of whose functions is to find all possible analyses, cannot be built; this has been done by Oettinger for natural language, and by Irons, for use in compiling.

Some Transformations of the Basic Specification

We shall now proceed to build up to the description of a particular simple Analyzer algorithm, and at this point, we must choose one among several different techniques. The differences between the various techniques stem from the following considerations:

—Given a Syntax Specification, there are different ways of using it to determine the next goal which the analyzer is to pursue. The two major approaches are called the “top-down” and the “bottom-up” techniques.

—There are different ways to use the output of the Analyzer, e.g., interpretation, immediate generation of output code, recording of the analyzed structure for later generation, etc.

The particular type of Analyzer we have chosen to describe here is, we believe, the easiest to explain, and is suitable for any of the three output-treatments mentioned above. It does not correspond, so far as we know, to any actually existing compiler system, although it bears a surprisingly strong resemblance to the algorithm used in some of the compilers that Computer Associates, Inc., has recently produced. (See Shapiro and Warshall).

The first step is to transform our Basic Syntax Specification into a Backus Normal Form. The description of a Syntactic Type Definition is now expanded so that the Definiens, instead of simply a Construction (which, remember, was a sequence of Components, which, in turn were Syntactic Types) can now be a sequence of Constructions, separated by the special sign ‘|’.

Any such sequence of Constructions, separated by ‘|’ and appearing in a Definiens is called an “Alternation,” and the individual Constructions in the sequence are called “Alternatives” of the Alternation. To transform a Basic Syntax Specification into Backus Normal Form, we must repeatedly apply the following transformation rule to the set of Definitions, until it can no longer be applied:
(T1) If any Defined Type has more than one Definition in the set, delete all such Definitions, and add to the set a new Definition whose left-hand side is the Defined Type in question, and whose Definiens is an Alternation of the original (Basic) Definiencies.

As an example, the Basic Syntax Specification for the simple language we are using for illustration in this paper would have contained three definitions for \(<factor>\):

\[
<factor> ::=<variable>
<factor> ::=<integer>
<factor> ::= (<arith expr>)
\]

After applying (T1) to the Basic Specification, these three Definitions would be replaced by the single Definition

\[
<factor> ::= <variable> | <integer> | (<arith expr>)
\]

This Definition, of course, should be read "a source string is a <factor> if it is either a <variable> or an <integer> or an <arith expr> enclosed in parentheses." This Backus Normal Form is exactly the form of Syntax Specification used in the defining documents for ALGOL 60 [8], and Table 1 presents a complete syntax for a simple arithmetic programming language in this form, which we shall refer to as "Syntax 1."

The Action of the Analyzer

We can now sketch out the action of the Analyzer: At the beginning of the process, it takes the Starting Type of the Specification as its first goal. Then at any point in the process it follows these steps when it has a Defined Type as its current goal:

The Analyzer consults the Definition of the Defined Type (in Backus Normal Form, of course, each Defined Type has a unique Definition), and specifically, it considers the first Alternative in that Definition. It then successively takes each Component of that Alternative as a sub-goal. (Of course, it must re-enter itself for each of these goals, and it must keep track of where it was at each level of re-entry.) If at any point it fails to find one of these sub-goals, it abandons that Alternative, and considers the next Alternative in that Definition, if there is one, and steps through the Components of that Alternative. If there is no next Alternative, it has failed to realize its current goal, and reports this fact "upstairs." If it succeeds in finding the sub-goals corresponding to each of the Components of any Alternative in the Definition of its current goal, it has found its goal, and reports that fact.

This rough sketch conveniently ignores a number of sticky points which we now have to consider. The first of these points is that we discussed the action of the Analyzer only when its current goal was a Defined Type. What if the goal is a Terminal Character?

When it comes to writing a compiler in practice, the question of recognizing Terminal Characters brings us face to face with the lovely problems of restricted character sets, input-output idiosyncracies of the particular computer, etc. Both in practice and in the remainder of this paper, we assume the presence of another routine, called the "Recognizer," which the Analyzer can call upon to deal with these problems. So far, we have also glossed over the problem of keeping track of where in the Input String the Analyzer is looking. Obviously, when the first Component of some Construction has been recognized, starting at a certain point in the Input String, then, when the Analyzer proceeds to look for the next Component, it must move its Input-String pointer past the substring which satisfied the first Component. Now, since a Type which has been successfully recognized consists, ultimately, of a sequence of Terminal Characters, and the recognition of Terminal Characters is the job of the Recognizer, we shall also leave the moving of the Input-String pointer to the Recognizer. The fundamental action of the Recognizer is then as follows:

The Recognizer is called by the Analyzer, and asked if a specified Terminal Character occurs at a stated character position in the Input String. The Recognizer then negotiates with the I/O system of the computer (if necessary) and examines the character-position in the input string. If the input character at that position is not the Terminal Character the Analyzer asked for, the Recognizer reports failure. However, if the input character is the desired Terminal Character,
the Recognizer reports success to the Analyzer, and advances the Input-string pointer by one character position.

Having the Recognizer at hand, it turns out to be convenient in practice to give it some further responsibilities. Consider the definitions of <variable> and <integer> in Syntax I. These amount to saying that an <integer> is any sequence of digits, and a <variable> is any sequence of letters or digits as long as the first one is a letter. If we relegate the recognition of these fundamental types to the Recognizer, rather than the Analyzer, we obtain several advantages.

— The Recognizer can be hand-tailored to perform these particular recognitions very efficiently on the particular machine, and this speeds up the analysis considerably.

— As far as the Analyzer is concerned, if the Syntax Specification calls for an <integer>, for instance, any old integer will do. But when we come to generating output code, we'll need to know the particular integer which occurred at that point. The Recognizer can perform the conversion from external number representation to machine representation, and either return the numerical value, or enter the number in a "Literal Table" and return its index value. Similarly, when it recognizes <variable>, it can look in a "Symbol Table" for previous occurrences of that particular variable, add it to the table if necessary, and return a line number.

— In practical applications the question of what constitutes a “blank” is often an involved one. In some languages, a long comment may function syntactically as a blank. When a compiler runs under the control of some operating systems, certain segments of the Input string (e.g., identification fields in cards) must be treated as blanks, or ignored entirely. Since the Recognizer constitutes the interface between the Analyzer and the outside world, it can take care of these matters.

To allow for this extended Recognizer in our Syntax Specification, we allow another sort of Terminal Type (up to now, we recall, the only Terminal Types have been Terminal Characters). We designate these new Terminal Types with script capital letters, and call them “Terminal Classes.” Thus, in Syntax I, we can delete the definitions of <variable>, <integer>, <letter>, and <digit>, and replace the Definientes of <variable> and <integer> by the Terminal Classes $V$ and $I$, respectively. This produces Syntax II, Table 1, which is the one we shall refer to throughout the rest of this paper.

But this technique could be carried further. A compiler-builder might decide that he prefers operator-precedence techniques for the analysis of arithmetic expressions, while keeping the flexibility of syntax-direction for analysis of the larger constructions. His arithmetic-expression scanner would then function as a Recognizer for the previous Defined Type `<arith expr>` and, for this simple language, the Syntactic Specification would take the form of Syntax III, Table 1.

To summarize: When the current goal is a Defined Type, the Analyzer calls upon itself to find it, but when the goal is a Terminal Type, it calls upon the Recognizer. When the Recognizer is called, it determines according to its own internal rules, whether the desired Terminal Type occurs in the Input string at the current pointer-position; if not, it reports failure; if so, it advances the pointer past the substring which constitutes the Terminal Type (single character, or member of a Terminal Class), and reports success.

Encoding the Syntax Specification

We are now almost ready to proceed to the encoding of the Syntax Specification for the use of the Analyzer, except for one embarrassing question:

Consider, as an example, the definition of <term> in Syntax II:

<term> ::= <factor> | <term> * <factor>

What if the Analyzer should find itself considering the second Alternative in this Definition? This would amount to the Analyzer saying to itself “in order to find my current goal, which is <term>, I must set out to find the first Component of this Alternative, which is <term>.” In order to find a term it must
TABLE 1
Alternative Syntax Specifications

Syntax I:
<program> ::= <assignment> | <assignment> ; <program>
<assignment> ::= <variable> = <arith expr>
<arith expr> ::= <term> | <arith expr> + <term>
<term> ::= <factor> | <term> * <factor>
<factor> ::= <variable> | <integer> | ( <arith expr> )
<variable> ::= <letter> | <variable> <letter> | <variable> <digit>
<integer> ::= <digit> | <integer> <digit>
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Syntax II:
<program> ::= <assignment> | <assignment> ; <program>
<assignment> ::= <variable> = <arith expr>
<arith expr> ::= <term> | <arith expr> + <term>
<term> ::= <factor> | <term> * <factor>
<factor> ::= <variable> | <integer> | ( <arith expr> )
<variable> ::= $'
<integer> ::= $'

Syntax III:
<program> ::= <assignment> | <assignment> ; <program>
<assignment> ::= <variable> = <arith expr>
<arith expr> ::= $'
<variable> ::= $'

be able to find a term first. This is called the 
"left recursion problem," and it has led some 
language designers to disallow Type Definitions 
which include Alternatives which mention the 
Defined Type of the Definition as their first 
Component. To a human analyst, of course, the 
intent of the Definition is plain; he should first 
look for a <factor>; if he finds one, he has 
indeed found a <term>, but he should con­
tinue looking to see if he can find a '*' followed 
by another <factor>; if he can, he has found 
a "longer" <term>, and should continue look­
ring for a still longer one; as soon as he fails 
to find a '*' following his latest <term>, he 
can stop looking, confident that he has found 
the longest <term> at that point in the string. 
This recognition process can be embodied in 
the encoding of the Syntax Specification, but it 
does require detecting the presence of these 
left-recursive alternatives, and giving them 
some special treatment. Keeping this in mind, 
we shall proceed to encode the Syntax Specifi­
cation.

The encoding consists of two tables, the Syn­
tax Type Table and the Syntax Structure Table. 
The Syntax Type Table will contain an entry 
for each Syntactic Type which occurs anywhere 
in the Syntax Specification, whether it be a De­
defined Type or a Terminal Type. Each entry i 
the Type Table consists of two items: a yes-no 
item TERMINAL, and an integer item LOOK­
FOR. When line $t$ in the Type Table corre­
sponds to a Terminal Type, TERMINAL [$t$] 
will be set to "yes," and LOOKFOR [$t$] will con­
tain an arbitrary code number which the Recog­
nizer will interpret as denoting the particular 
Terminal Character or Terminal Class it should 
try to recognize. When line $t$ in the Type Table
### TABLE 2
The Syntax Tables

<table>
<thead>
<tr>
<th>(GOAL)</th>
<th>(Index)</th>
<th>TERMINAL</th>
<th>LOOK-FOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;program&gt;</td>
<td>i</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>&lt;assignment&gt;</td>
<td>ii</td>
<td>No</td>
<td>4</td>
</tr>
<tr>
<td>&lt;arith expr&gt;</td>
<td>iii</td>
<td>No</td>
<td>7</td>
</tr>
<tr>
<td>&lt;term&gt;</td>
<td>iv</td>
<td>No</td>
<td>10</td>
</tr>
<tr>
<td>&lt;factor&gt;</td>
<td>v</td>
<td>No</td>
<td>13</td>
</tr>
<tr>
<td>&lt;variable&gt;</td>
<td>vi</td>
<td>No</td>
<td>18</td>
</tr>
<tr>
<td>&lt;integer&gt;</td>
<td>vii</td>
<td>No</td>
<td>19</td>
</tr>
<tr>
<td>/</td>
<td>viii</td>
<td>Yes</td>
<td>101</td>
</tr>
<tr>
<td>\</td>
<td>ix</td>
<td>Yes</td>
<td>102</td>
</tr>
<tr>
<td>;</td>
<td>x</td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>xi</td>
<td>Yes</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>xii</td>
<td>Yes</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>xiii</td>
<td>Yes</td>
<td>4</td>
</tr>
<tr>
<td>(</td>
<td>xiv</td>
<td>Yes</td>
<td>5</td>
</tr>
<tr>
<td>)</td>
<td>xv</td>
<td>Yes</td>
<td>6</td>
</tr>
</tbody>
</table>

2.1
Syntax Type Table

<table>
<thead>
<tr>
<th>SOURCE (Index)</th>
<th>TYPE-CODE</th>
<th>STRUCT</th>
<th>SUCCESSION</th>
<th>ALTERNATE</th>
<th>Corresponds to Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ii</td>
<td>Yes</td>
<td>2</td>
<td>FAIL</td>
<td>1.1</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>Yes</td>
<td>3</td>
<td>OK</td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>Yes</td>
<td>4</td>
<td>OK</td>
<td>1.2</td>
</tr>
<tr>
<td>4</td>
<td>vi</td>
<td>Yes</td>
<td>5</td>
<td>FAIL</td>
<td>3.1</td>
</tr>
<tr>
<td>5</td>
<td>xi</td>
<td>Yes</td>
<td>6</td>
<td>FAIL</td>
<td>3.2</td>
</tr>
<tr>
<td>6</td>
<td>iii</td>
<td>Yes</td>
<td>7</td>
<td>OK</td>
<td>3.2</td>
</tr>
<tr>
<td>7</td>
<td>iv</td>
<td>Yes</td>
<td>8</td>
<td>FAIL</td>
<td>4.1</td>
</tr>
<tr>
<td>8</td>
<td>xii</td>
<td>Yes</td>
<td>9</td>
<td>OK</td>
<td>4.1</td>
</tr>
<tr>
<td>9</td>
<td>xiv</td>
<td>Yes</td>
<td>11</td>
<td>FAIL</td>
<td>4.2</td>
</tr>
<tr>
<td>10</td>
<td>v</td>
<td>Yes</td>
<td>12</td>
<td>OK</td>
<td>4.2</td>
</tr>
<tr>
<td>11</td>
<td>xiii</td>
<td>Yes</td>
<td>13</td>
<td>OK</td>
<td>5.1</td>
</tr>
<tr>
<td>12</td>
<td>v</td>
<td>Yes</td>
<td>14</td>
<td>OK</td>
<td>5.1</td>
</tr>
<tr>
<td>13</td>
<td>vi</td>
<td>Yes</td>
<td>15</td>
<td>OK</td>
<td>5.2</td>
</tr>
<tr>
<td>14</td>
<td>vii</td>
<td>Yes</td>
<td>16</td>
<td>FAIL</td>
<td>5.2</td>
</tr>
<tr>
<td>15</td>
<td>xiv</td>
<td>Yes</td>
<td>17</td>
<td>FAIL</td>
<td>5.3</td>
</tr>
<tr>
<td>16</td>
<td>iii</td>
<td>Yes</td>
<td>18</td>
<td>FAIL</td>
<td>6.1</td>
</tr>
<tr>
<td>17</td>
<td>xv</td>
<td>Yes</td>
<td>19</td>
<td>FAIL</td>
<td>7.1</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
corresponds to a Defined Type, TERMINAL \([t]\) will be set to “No,” and LOOKFOR \([t]\) will contain some line-number in the Syntax Structure Table, to be filled in presently. We keep in mind that we can now use small integers as, in effect, names of Syntactic Types, by using them to index the Syntax Type Table.

The Syntax Structure Table will contain a line for each Component of each Alternative of each Definiens in the Syntax Specification. Each line of the Structure Table will consist of four items:

- **TYPECODE**, an integer item, will contain line numbers referring to the Type Table;
- **STRUCT**, a yes-no item;
- **SUCCESSOR** and **ALTERNATE**, integer items, will contain line numbers referring to other lines of the Structure Table, plus two special codes denoted by “OK” and “FAIL.”

The Syntax Structure Table is constructed according to the following rules:

Consider first a Defined Type which has no left-recursive Alternative in its Definiens. Reserve a block of entries in the Structure Table. Assign an entry in the Structure Table to each Component in each Alternative in the Definiens. In the Type Table line corresponding to this Defined Type—say, \(t\)—set LOOKFOR \([t]\) to the Structure-Table line number assigned to the first Component of the First Alternative of the Definiens. In each Component-line \(s\), set TYPECODE \([s]\) to the Type Table line number of the Syntactic Type which occurs as that Component in the Definiens. In each line corresponding to a Component which is not the last Component of an Alternative, set STRUCT to “No” and SUCCESSOR to the line corresponding to the next Component. In each line corresponding to a Component which is the last Component of an Alternative, set STRUCT to “Yes” and SUCCESSOR to “OK.” In each line corresponding to a first Component of any Alternative except the last Alternative of the Definiens, set ALTERNATE to the line corresponding to the first component of the next Alternative. Set all other ALTERNATE fields to “FAIL.”

If the Defined Type contains a left-recursive Alternative: (we shall here assume there is only one left-recursive Alternative—See Appendix). Set the left-recursive Alternative aside temporarily, and carry out the above process for the other Alternatives. Then:

Assign a Structure-Table line to each Component of the recursive Alternative except the recursive Component itself.

Set TYPECODE in each of these lines, as above.

Set SUCCESSOR and STRUCT in each of these lines, except for the last Component, as above.

Call the first of these lines (the one corresponding to the Component which immediately follows the recursive Component in the Definiens) the “handle.”

Set ALTERNATE in each of these lines, except the handle, to “FAIL.”

Set ALTERNATE in the handle to “OK.”

Set SUCCESSOR in the line for the last Component of this Alternative to the handle, and set STRUCT in this line to “Yes.”

Now, in the lines corresponding to last Components in all the other Alternatives in this Definiens, SUCCESSOR will have been set to “OK” by the nonrecursive treatment. Replace each of these “OK”s by the line number of the handle.

The Syntax Type Table and the Syntax Structure Table corresponding to Syntax II are shown together as Table 2. In the hope of
reducing the confusion involved in having entries in each table pointing to the other, we have indicated the indexing of the Syntax Type Table with Roman numerals, and the indexing of the Syntax Structure Table with Arabic numerals, and we have added the names of the syntactic types corresponding to the lines of the Syntax Type Table.

The Analyzer Algorithm

The flow chart, Figure 1, illustrates an Analyzer working from Syntax Tables of the sort we have just constructed. The following remarks will help to follow the flow chart.

—The global quantities GOAL, SOURCE, and CHAR are used as follows:

GOAL is the line number in the Syntax Type Table corresponding to the Syntactic Type currently being considered (initially, the Starting Type).

SOURCE is the line number in the Syntax Structure Table of the Component currently being considered (initially undefined, hence set to zero).

CHAR is the (ordinal) number of the character position in the Input String next to be considered (initially set to the beginning of the String, here 1).

—The operations of “Pushdown” and “Pop-up” are performed on these globals—this may be done by any appropriate mechanism. For definiteness, let us assume an index Y (initially zero) accessible to these two operations, and arrays GYOYO, SYOYO, and CYOYO. Then (in quasi ALGOL notation),

\[
\begin{align*}
\text{Pushdown: } & Y := Y + 1 \\
& \text{GYOYO}[Y] := \text{GOAL} \\
& \text{SYOYO}[Y] := \text{SOURCE} \\
& \text{CYOYO}[Y] := \text{CHAR};
\end{align*}
\]

\[
\begin{align*}
\text{Popup: } & \text{GOAL} := \text{GYOYO}[Y]; \\
& \text{SOURCE} := \text{SYOYO}[Y]; \\
& \text{if } \text{CHAR} \text{ is mentioned in the call then } \text{CHAR} := \text{CYOYO}[Y]; \quad Y := Y - 1;
\end{align*}
\]

Plausibly, CHAR is popped up when an Alternative has failed, and the Analyzer must back up to the beginning of that Construction and try another Alternative at the same place; and CHAR is not popped up—hence left as it has been set by the successful recognitions—when some Alternative has succeeded.

—Recognize is assumed as described earlier: It returns a success/failure indicator which is tested in the “Found?” box. For definiteness again, we shall assume that, when it succeeds in recognizing a Terminal Class, it places a Symbol-Table or Literal-Table line number in some global location, for the Generator to use.

—The following sections of this paper will discuss possible uses to be made of the Analyzer’s results. The routine which considers these results is named the “Generator,” and it is represented in this flow chart by a subroutine call box: “Generate.” When Generate is called, the value of SOURCE uniquely indicates the Syntactic Type which has been recognized and, moreover, the particular Alternative in the Definition of that Syntactic Type which has just succeeded. The column headed “Corresponds to Definitions” has been added to the Syntax Structure Table to indicate this relationship. The numbers in this column correspond to the Alternatives in the “semantics” tables, Tables 4 and 5.

DOING SOMETHING USEFUL WITH THE ANALYSIS

A syntactic analysis, such as that depicted verbally in the preceding section or via the flow chart in Figure 1, is an important part of the problem which must be solved by a compiler, but it is only a part. The goal of a compiler is, after all, to produce the coding required to carry out the procedure described in the programming language being compiled. This coding might be desired as actual machine instructions for some computer or it might be desired as instructions appropriate for some interpreter available on one or more machines or it might be desired in some other form. In any event, some further processing is required once the syntactic analysis is complete in order to “generate” and format the coding to be output.

Let us suppose that the syntactic analysis has proceeded to the point where some syntactic type has been recognized (in the flow chart, Figure 1, we have passed through the “GENERATE” box). The contents of SOURCE tells us
which syntactic type has been recognized as well as which alternative construction of the syntactic type was built. Thus, some action or set of actions could be initiated at this point to process this syntactic type in a variety of ways.

For example, Table 3 gives for each alternative construction of the syntactic types of Syntax II a verbal description of the computations to be performed upon recognition of that construction. Corresponding to this table, the analysis of the assignment statement

\[ X = NU^* (Y + 15) \]

could yield the following fragments of (slightly edited) verbal description:

<table>
<thead>
<tr>
<th>Syntax-Directed Compiling</th>
<th>41</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TABLE 3</strong></td>
<td></td>
</tr>
<tr>
<td>&quot;Interpretive Semantics&quot; for Syntax II</td>
<td></td>
</tr>
<tr>
<td>1.1 &lt;program&gt; ::= &lt;assignment&gt;</td>
<td>{Execute the &lt;assignment&gt; then halt}</td>
</tr>
<tr>
<td>1.2</td>
<td>&lt;assignment&gt; ; &lt;program&gt;</td>
</tr>
<tr>
<td>2.1 &lt;assignment&gt; ::= &lt;variable&gt; = &lt;arith expr&gt;</td>
<td>{&quot;Locate&quot; the &lt;variable&gt; (determine its address for later assignment of value); then evaluate the &lt;arith expr&gt;; then assign its value to the &lt;variable&gt;}</td>
</tr>
<tr>
<td>3.1 &lt;arith expr&gt; ::= &lt;term&gt;</td>
<td>{Evaluate the &lt;term&gt;; the value of the &lt;arith expr&gt; is this value}</td>
</tr>
<tr>
<td>3.2</td>
<td>&lt;arith expr&gt; + &lt;term&gt;</td>
</tr>
<tr>
<td>4.1 &lt;term&gt; ::= &lt;factor&gt;</td>
<td>{Evaluate the &lt;factor&gt;; the value of the &lt;term&gt; is this value}</td>
</tr>
<tr>
<td>4.2</td>
<td>&lt;term&gt; * &lt;factor&gt;</td>
</tr>
<tr>
<td>5.1 &lt;factor&gt; ::= &lt;variable&gt;</td>
<td>{The value of the &lt;factor&gt; is the value of the &lt;variable&gt;}</td>
</tr>
<tr>
<td>5.2</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>5.3</td>
<td>(&lt;arith expr&gt;)</td>
</tr>
<tr>
<td>6.1 &lt;variable&gt; ::= ( Q/ )</td>
<td>{The value of the &lt;variable&gt; is the value most recently assigned to the variable ( Q/ )}</td>
</tr>
<tr>
<td>7.1 &lt;integer&gt; ::= ( f )</td>
<td>{The value of the &lt;integer&gt; is the value of the integer ( f ) (according to the conventional representation of integers)}</td>
</tr>
</tbody>
</table>
4.1 <term> ::= <factor> <variable> NU

5.1 <factor> ::= <variable> Y

4.1 <term> ::= <factor> Y

3.1 <arith expr> ::= <term> Y

7.1 <integer> ::= \[ \]

5.1 <factor> ::= <integer> 15

4.1 <term> ::= <factor> 15

3.1 <arith expr> ::= <term> + <term> Y + 15

Thus, with a certain amount of editing, the recognition of \( X = NU \times (Y + 15) \) yields the verbal description:

"Let NU and Y represent the values most recently assigned to the variables NU and Y; then compute \( NU \times (Y + 15) \) and assign the resulting value to the variable X."

Table 4 illustrates a very simple approach to the problem of machine code synthesis. With each syntactic construction is associated a set of actions. These actions are of two types—output and set. The interpretation of the actions is reasonably obvious. The bracketed numerals under the components of a construc-
TABLE 4
Machine Code Semantics for Syntax II—Direct Generation

1.1 <program> ::= <assignment>  
1. Output END

1.2 | <assignment> ; <program>  
1. Output CLA (addr )  
2. Output STO addr  

2.1 <assignment> ::= <variable> = <arith expr>  
1. Output CLA (addr )  
2. Output STO addr  

3.1 <arith expr> ::= <term>  
1. Set addr = addr  

3.2 | <arith expr> + <term>  
1. Output CLA (addr )  
2. Output ADD (addr )  
3. Output STO (addr )  

4.1 <term> ::= <factor>  
1. Set addr = addr  

4.2 | <term> * <factor>  
1. Output LDQ (addr )  
2. Output MPY (addr )  
3. Output STQ (addr )  

5.1 <factor> ::= <variable>  
1. Set addr = addr

5.2 | <integer>  
1. Set addr = addr  

5.3 | (<arith expr>)  
1. Set addr = addr  

6.1 <variable> ::= 'f'  
1. Set addr = the variable name recognized at this point of the input string.

7.1 <integer> ::= 'f'  
1. Set addr = a symbolic name for the address in which will be kept the integer constant recognized at this point in the input string.
tion identify the components. The notation `addr` means the result address associated with the component identified by `[1]` in the syntax specification. A bracketed blank represents the syntactic type being defined and `addr` represents the result address of this type; when the `addr` appears in a machine instruction a temporary storage register is to be assigned. In the example below we use the notation `t_j` for the `j`th such temporary. Again consider the assignment

\[ X = \text{NU} \times (Y + 15). \]

If this assignment is analyzed and actions carried out as per Table 4, the following results:

<table>
<thead>
<tr>
<th>Line</th>
<th>Construct</th>
<th>Source</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td><code>&lt;variable&gt;</code> ::= ( V )</td>
<td>( X )</td>
<td><code>addr [&lt;variable&gt;] = X</code></td>
</tr>
<tr>
<td>6.1</td>
<td><code>&lt;variable&gt;</code> ::= ( V )</td>
<td>( \text{NU} )</td>
<td><code>addr [&lt;variable&gt;] = \text{NU}</code></td>
</tr>
<tr>
<td>5.1</td>
<td><code>&lt;factor&gt;</code> ::= <code>&lt;variable&gt;</code></td>
<td>( \text{NU} )</td>
<td><code>addr [&lt;factor&gt;] = \text{NU}</code></td>
</tr>
<tr>
<td>4.1</td>
<td><code>&lt;term&gt;</code> ::= <code>&lt;factor&gt;</code></td>
<td>( \text{NU} )</td>
<td><code>addr [&lt;term&gt;] = \text{NU}</code></td>
</tr>
<tr>
<td>6.1</td>
<td><code>&lt;variable&gt;</code> ::= ( V )</td>
<td>( Y )</td>
<td><code>addr [&lt;variable&gt;] = Y</code></td>
</tr>
<tr>
<td>5.1</td>
<td><code>&lt;factor&gt;</code> ::= <code>&lt;variable&gt;</code></td>
<td>( Y )</td>
<td><code>addr [&lt;factor&gt;] = Y</code></td>
</tr>
<tr>
<td>4.1</td>
<td><code>&lt;term&gt;</code> ::= <code>&lt;factor&gt;</code></td>
<td>( Y )</td>
<td><code>addr [&lt;term&gt;] = Y</code></td>
</tr>
<tr>
<td>3.1</td>
<td><code>&lt;arith expr&gt;</code> ::= <code>&lt;term&gt;</code></td>
<td>( Y )</td>
<td><code>addr [&lt;arith expr&gt;] = Y</code></td>
</tr>
<tr>
<td>7.1</td>
<td><code>&lt;integer&gt;</code> ::= ( j )</td>
<td>15</td>
<td><code>addr [&lt;integer&gt;] = 15</code></td>
</tr>
<tr>
<td>5.2</td>
<td><code>&lt;factor&gt;</code> ::= <code>&lt;integer&gt;</code></td>
<td>15</td>
<td><code>addr [&lt;factor&gt;] = 15</code></td>
</tr>
<tr>
<td>4.1</td>
<td><code>&lt;term&gt;</code> ::= <code>&lt;factor&gt;</code></td>
<td>15</td>
<td><code>addr [&lt;term&gt;] = 15</code></td>
</tr>
<tr>
<td>3.1</td>
<td><code>&lt;arith expr&gt;</code> ::= <code>&lt;arith expr&gt;</code> + <code>&lt;term&gt;</code></td>
<td>( Y + 15 )</td>
<td><code>CLA Y</code> ADD=15 STO ( t_1 )`</td>
</tr>
<tr>
<td>5.3</td>
<td><code>&lt;factor&gt;</code> ::= <code>( &lt;arith expr&gt; )</code></td>
<td>( Y + 15 )</td>
<td><code>addr [&lt;factor&gt;] = \( t_1 \)</code></td>
</tr>
<tr>
<td>4.1</td>
<td><code>&lt;term&gt;</code> ::= <code>&lt;term&gt;</code>* <code>&lt;factor&gt;</code></td>
<td>( \text{NU} \times (Y + 15) )</td>
<td><code>LDQ \text{NU} MPY \( t_1 \) STQ \( t_2 \)</code></td>
</tr>
<tr>
<td>3.1</td>
<td><code>&lt;arith expr&gt;</code> ::= <code>&lt;term&gt;</code></td>
<td>( \text{NU} \times (Y + 15) )</td>
<td><code>addr [&lt;arith expr&gt;] = \( t_2 \)</code></td>
</tr>
<tr>
<td>2.1</td>
<td><code>&lt;assignment&gt;</code> ::= <code>&lt;variable&gt;</code> = <code>&lt;arith expr&gt;</code></td>
<td>( X = \text{NU} \times (Y + 15) )</td>
<td><code>CLA \( t_2 \) STO \( X \)</code></td>
</tr>
</tbody>
</table>

Given this mechanism, which we shall refer to in the sequel as the "Direct Generation Mechanism," plus some mechanism for creating and housekeeping local or internal labels and a sufficiently sophisticated assembler (e.g., it can allocate memory for constants and variables) we have a rudimentary compiler.
## Table 5
Machine Code Semantics for Syntax II—Deferred Generation

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td><code>&lt;program&gt;</code> ::= <code>&lt;assignment&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1]</td>
</tr>
<tr>
<td>1.2</td>
<td><code>&lt;assignment&gt;</code> ::= <code>&lt;assignment&gt; ; &lt;program&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1] [2] [3]</td>
</tr>
<tr>
<td>2.1</td>
<td><code>&lt;assignment&gt;</code> ::= <code>&lt;variable&gt; = &lt;arith expr&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1] [2] [3]</td>
</tr>
<tr>
<td>3.1</td>
<td><code>&lt;arith expr&gt;</code> ::= <code>&lt;term&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1]</td>
</tr>
<tr>
<td>3.2</td>
<td><code>&lt;arith expr&gt;</code> ::= <code>&lt;arith expr&gt; + &lt;term&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1] [2] [3]</td>
</tr>
<tr>
<td>4.1</td>
<td><code>&lt;term&gt;</code> ::= <code>&lt;factor&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1]</td>
</tr>
<tr>
<td>4.2</td>
<td><code>&lt;term&gt;</code> ::= <code>&lt;term&gt; * &lt;factor&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1] [2] [3]</td>
</tr>
<tr>
<td>5.1</td>
<td><code>&lt;factor&gt;</code> ::= <code>&lt;variable&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1]</td>
</tr>
<tr>
<td>5.2</td>
<td><code>&lt;factor&gt;</code> ::= <code>&lt;integer&gt;</code></td>
</tr>
<tr>
<td></td>
<td>[1]</td>
</tr>
<tr>
<td>5.3</td>
<td><code>&lt;factor&gt;</code> ::= <code>( &lt;arith expr&gt; )</code></td>
</tr>
<tr>
<td></td>
<td>[1] [2] [3]</td>
</tr>
<tr>
<td>6.1</td>
<td><code>&lt;variable&gt;</code> ::= <code>Q</code></td>
</tr>
<tr>
<td>7.1</td>
<td><code>&lt;integer&gt;</code> ::= <code>f</code></td>
</tr>
</tbody>
</table>

1. Process [1]  
3. Output CLA addr  
4. Output STO addr  
5. Output ADD addr  
6. Output STO addr  
7. Output LDQ addr  
8. Output MPY addr  
9. Output STQ addr  
13. Set addr = addr  
14. Set addr = addr  
15. Set addr = addr  
16. Set addr = the variable name recognized at this point of the input string.  
17. Set addr = a symbolic name for the address in which will be kept the integer constant recognized at this point in the input string.
There is an interesting variation in this method of generating machine code. Let us suppose that, for one reason or another, it is desirable to complete the analysis for an entire assignment and produce the tree representation of its syntax, and then to generate the machine coding which is to correspond to the tree so constructed. Table 5 illustrates this approach. It is essentially Table 4 to which some further actions have been appended. In this table the action denoted by a bracketed numeral preceded by the word “process” is interpreted: “do the actions for the component indicated.” Thus, again given the assignment:

$$X = NU*(Y + 15)$$

an analysis of this assignment with respect to Syntax II could be carried out resulting in the tree of Figure 2. Given this tree, the actions indicated in Table 5 could result in the following:

<table>
<thead>
<tr>
<th>Tree Node</th>
<th>Line of Table 5</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>2.1</td>
<td>Process &lt;variable&gt;, node 2.</td>
</tr>
<tr>
<td>25</td>
<td>2.1</td>
<td>Process &lt;arith expr&gt;, node 24.</td>
</tr>
<tr>
<td>6</td>
<td>5.1</td>
<td>Process &lt;factor&gt;, node 6.</td>
</tr>
<tr>
<td>5</td>
<td>6.1</td>
<td>Set addr(5) = NU; actions for node 5 complete; return to node 6.</td>
</tr>
<tr>
<td>6</td>
<td>4.1</td>
<td>Set addr(6) = NU; actions for node 6 complete; return to node 7.</td>
</tr>
<tr>
<td>7</td>
<td>4.1</td>
<td>Set addr(7) = NU; actions for node 7 complete; return to node 23.</td>
</tr>
<tr>
<td>23</td>
<td>4.1</td>
<td>Process &lt;factor&gt;, node 22.</td>
</tr>
<tr>
<td>22</td>
<td>5.3</td>
<td>Process &lt;arith expr&gt;, node 20.</td>
</tr>
<tr>
<td>20</td>
<td>3.2</td>
<td>Process &lt;arith expr&gt;, node 14.</td>
</tr>
<tr>
<td>14</td>
<td>3.1</td>
<td>Process &lt;term&gt;, node 13.</td>
</tr>
</tbody>
</table>

Figure 2. Syntax Tree for the assignment

"X = NU* (Y + 15)."
SYNTAX-DIRECTED COMPILING

<table>
<thead>
<tr>
<th>Tree Node</th>
<th>Line of Table 5</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>4.1</td>
<td>Process &lt;factor&gt;, node 12.</td>
</tr>
<tr>
<td>12</td>
<td>5.1</td>
<td>Process &lt;variable&gt;, node 11.</td>
</tr>
<tr>
<td>11</td>
<td>6.1</td>
<td>Set addr_{11} = Y; actions for node 11 complete; return to node 12.</td>
</tr>
<tr>
<td>12</td>
<td>5.1</td>
<td>Set addr_{12} = Y; actions for node 12 complete; return to node 13.</td>
</tr>
<tr>
<td>13</td>
<td>4.1</td>
<td>Set addr_{13} = Y; actions for node 13 complete; return to node 14.</td>
</tr>
<tr>
<td>14</td>
<td>3.1</td>
<td>Set addr_{14} = Y; actions for node 14 complete; return to node 20.</td>
</tr>
<tr>
<td>20</td>
<td>3.2</td>
<td>Process &lt;term&gt;, node 19.</td>
</tr>
<tr>
<td>19</td>
<td>4.1</td>
<td>Process &lt;factor&gt;, node 18.</td>
</tr>
<tr>
<td>18</td>
<td>5.2</td>
<td>Process &lt;integer&gt;, node 17.</td>
</tr>
<tr>
<td>17</td>
<td>7.1</td>
<td>Set addr_{17} = 15; actions complete for node 17, return to node 18.</td>
</tr>
<tr>
<td>18</td>
<td>5.2</td>
<td>Set addr_{18} = 15; actions complete for node 18, return to node 19.</td>
</tr>
<tr>
<td>19</td>
<td>4.1</td>
<td>Set addr_{19} = 15; actions for node 19 complete; return to node 20.</td>
</tr>
</tbody>
</table>
| 20        | 3.2            | Output CLA Y 
Output ADD = 15 
Output STO t_1 
Set addr_{20} = t_1; actions for node 20 complete; return to node 22. |
| 22        | 5.3            | Set addr_{22} = t_1; actions for node 22 complete; return to node 23. |
| 23        | 4.2            | Output LDQ NU 
Output MPY t_1 
Output STQ t_2 
Set addr_{23} = t_2; actions for node 23 complete; return to node 24. |
| 24        | 3.1            | Set addr_{24} = t_2; actions for node 24 complete; return to node 25. |
| 25        | 2.1            | Output CLA t_2 
Output STO X 
Set addr_{25} = X; actions for node 25 complete; exit. |

Note that we have changed notation slightly and used the notation addr_{result}, for example, to indicate the (result) address which is associated with node 5.

The "code generation" mechanism implied by the above description is as follows: At any point in time some action of some node is to be performed; the actions and their interpretations are:
Interpretation Action  
Process a node  
Remember which action of the current node is being processed; perform the first action of the indicated node.

Output code  
Process [1]  
Example  
Output CLA addr [1]  
Procure the result address from the node indicated (the node corresponding to addr in the example) and output the instruction. If the address indication is blank (e.g., STO addr ) select the next available temporary register address; set that address as the result address of the current node.

Set address  
Set addr = [ ]  
Set the result address of the current node as the result address of the node indicated.

End of actions  
Return to the next action of the node which “called” for this node to be processed.

Thus, the mechanism has a “control” element which is, at any point in time, considering some node. When a new node is to be processed, the “control” remembers which action to proceed with when “control” returns to the current node and then initiates the first action of the node to be processed. When all actions for a node are processed, “control” returns to the node which called for the current node and takes up actions for that node where it left off. Further, the mechanism has the ability to output code, associate result addresses with the node being processed, and procure temporary register addresses.

Again given this mechanism, which we shall refer to in the sequel as the “Deferred Generation Mechanism,” plus some mechanism for creating and housekeeping local or internal labels and a sufficiently sophisticated assembler, we have a rudimentary compiler.

There are, of course, other kinds of “actions” one could associate with a node. For example, it would be quite straightforward to associate actions for producing a different tree structure than the complete syntax tree as depicted in Figure 2. This might then produce, from an analysis of the assignment

\[ X = NU^*(Y + 15) \]

the simple tree (or “Polish prefix” representation):

![Polish prefix tree](image)

It will be apparent that the Direct Generation Mechanism does not require that the complete syntactic tree actually be built as the analysis proceeds. Rather, it is sufficient that there be some means (for example, a push down store is adequate) for “remembering” the result addresses which have yet to be “used.” Further, while this technique appears, on the face of it, to be quite rapid and efficient (no tree need be kept, shorter “driving tables”—compare Table 4 with Table 5) it is subject to some serious limitations. In particular, since the coding for a syntactic type is actually being output as that type is recognized, there must (for most languages) be some mechanism for “erasing” a patch of code generated for a syntactic type recognized while the analyzer was attempting recognition of some larger construction when it turns out that the syntactic type in question does not enter into the construction of the larger syntactic type as it is finally built.
In the above example with the Deferred Code Mechanism we used \texttt{<assignment>} as the syntactic type over which (i.e., over the tree representation of which) generation was to occur. It is, of course, possible to generalize this to allow any syntactic type to be "tagged" as big enough to call the generation mechanism. Thus, at one extreme a complete program would have to be recognized before any generation was performed ("two pass" compiler) and, at the other extreme, each syntactic type would call for generation ("one pass" compiler) thus making the Deferred Generation Mechanism essentially the same as the Direct Generation Mechanism. It should be noted that, employing the Deferred Generation Mechanism, once the tree corresponding to some syntactic type has been processed ("generated over") it can be erased with the exception of its top-most or root node which may have to remain to supply a "result address" for the generation over some larger construction.

It must be emphasized that both these mechanisms are very rudimentary and for use within a compiler would require some embellishment in order to be practical. Thus, for example, it seems rather a shame to generate a complete syntax tree for some (perhaps fragment of some) program and then make essentially no use of the contextual information contained implicitly in the tree structure. Indeed, a rather simple addition to make some use of this information would be the following: consider that we add conditional actions of the following sort:

\begin{verbatim}
IF addr [a] = addr [b], SKIP n 
IF addr [a] \neq addr [b], SKIP n 
SKIP n
\end{verbatim}

where in the first two the truth of the relation causes the n actions following to be skipped and the SKIP n action causes the n actions following to be skipped. If we further add "AC" and "MQ" as special values for addr \[ \], then for a single address computer (say, like the IBM-7094), it would be possible to generate rather more efficient coding by placing results temporarily in the accumulator (AC) or multiplier quotient (MQ) registers and then checking for the use of these locations for operands before generating coding. Thus we might then associate with the construction \texttt{<arith expr> :- \texttt{<arith expr>} + <term>} in Table 5 the actions:

1. Process [1]
3. IF addr [3] = AC, SKIP 4
   [1]
4. IF addr [1] = AC, SKIP 1
5. Output CLA addr [1]
6. Output ADD addr [3]
7. SKIP 1
8. Output ADD addr [1]

These would have the effect of preserving results of additions in the accumulator and remembering that they were there in order to avoid redundant store-load instructions. In order to fully utilize such a facility, including keeping track of the MQ contents as well, some further mechanism for indicating the AC or MQ are empty or full and some mechanism for storing their contents would be required. The basic scheme is, however, reasonably clear from the example. The MAD Translator has facilities similar to these built into its code generation mechanism.

Even such a mechanism as this barely makes use of the rather rich store of contextual information available. In order to do so, however, we would require some means for talking about the nodes of the tree relative to any given node of interest (such as a node "father," "father's father," "father's right-brother's son's son," and so on). Further, it would probably be desirable to extend the "control" some what and allow more general "tree walks" than simply processing "sons" and returning to "father." Also, if contextual information were gathered, it would have to be "parked" somewhere and thus an addition of further information fields associated with each node would be useful plus, perhaps, some "working variables" which various actions could reference and set.
It is clear that one could extend the whole generation mechanism to include control sequencing and actions which were programs in a rather specialized programming language. The paper by Warshall and Shapiro [1] contains a brief sketch of one such extension which has been successfully tried on several computers and for several languages.

SOME PROBLEMS AND LIMITATIONS

The techniques and fragments of techniques which we have discussed above are all, in one way or another, relevant to the design and construction of compilers. Furthermore, these techniques, in the simplified form in which we have presented them, are of sufficient general utility that they should be in every "systems" programmer's bag of tricks. For example, the ANALYZE algorithm—coupled with either the Direct or Deferred Generation Mechanism discussed in the preceding sections—can be applied to a variety of programming tasks imbedding simple algebraic facilities in an assembly program, handling the "translation" of free format or formatted control cards, interpreting descriptions of formatted data, and so on. However, for the serious construction of compilers these techniques represent only a few of the techniques required and they are subject to some limitations.

Some of the considerations which must enter into any compiler design and which are affected to one degree or another by the choice of method of analysis and its linkage to the generation of code are the following:

—error analysis and recovery
—analysis of languages in which recognition order is not the same as generation order
—processing declarations
—generation of highly efficient coding

Let us consider these questions.

Error Analysis and Recovery

An error is detected, for example, in Syntax II, when the analyzer cannot recognize a <program>. Although the exact point in the input string past which recognition fails will be known, it is extremely difficult to determine exactly why the error occurred and to, in a general way, devise means for recovery.

Several schemes exist for dealing with this problem, notably:

1) A scheme which permits specification of "no back up" on certain constructs. For example, in Syntax II, no back up on recognition of "=" or "(" could help isolate the reasons for a failure.

2) A scheme due to E. T. Irons [5] which, in effect, carries along all possible parses of an input string.

3) Special "error" syntactic types which could be defined in the syntax.

At the present time there is no completely satisfactory scheme for dealing with syntactic errors discovered in the course of predictive analysis. If the programming language which is being analyzed has sufficiently simple structure that it is a precedence grammar, the technique of bounded context analysis is probably a better technique to utilize. A discussion of precedence grammars is given in Reference [6]; the use of bounded context analysis is described in Reference [7].

Recognition Order Differs from Generation Order

Some reasons why the order of generation might be different from the order of recognition are:

1) The detection of, and generation of coding for, sub-expressions which are common to two or more parts of the program is desired.

2) The detection of computations which are invariant in some larger computation (for example within loops) is desired.

3) Languages other than the usual programming languages are being translated, for example, data description languages or the computational fragments associated with TABSOL-like descriptions are to be processed.

Reference [1] describes some techniques for coping with these problems in a compiler which uses predictive analysis.
**Handing Declarations**

Here the problem is that the "actions" are not to generate coding (usually) but to change the syntax—normally through type coding information inserted into a symbol table. Formally, however, a declaration of type is really the appending of a syntax rule. Thus the ALGOL 60 declaration

```
"real X, Y;"
```

means that the two new syntax rules

\[
<\text{real var}> ::= X
\]

and

\[
<\text{real var}> ::= Y
\]

must be appended to the syntax.

Other declarations may cause changes to the full compiler—for example, debug mode declaration, and the like.

**Generation of Highly Efficient Coding**

This cannot be accomplished by generating code directly as the analysis is performed since common sub-expressions, invariant computations and the like couldn't be detected reasonably and special registers such as index registers certainly couldn't be allocated on any global basis, which is necessary if any really effective use is to be made of them.

Many of the manipulations required to collect the information pertinent to optimizing code are not particularly easily done (or, at least efficiently done) with the source material in a syntax tree form. Reference [1] describes a method by which such optimizations are performed over a set of "macro-instructions" which are "generated" by a technique similar to that depicted by Table 5.

**SUMMARY AND CONCLUSION**

In this paper we have tried to explain the workings of syntax-directed compiling techniques—or perhaps better, of those parts of a compiler in which the actions to be performed can reasonably be associated with the structure of the input string. A satisfying understanding of the operation of a syntax-directed analyzer can only be attained by actually playing through a few examples. We recommend this as a worth-while experience to anyone who is interested, and so we have given a sufficiently detailed description of a particular example to permit the reader to write statements of his own in the simple language, and play them through the Analyzer and any one of several code-generation techniques.

There remains the question of evaluating syntax directed compiler techniques in comparison to other approaches.

On the face of it, syntax directed analyzers cannot be as efficient as operator-precedence techniques for the simple task of recognizing input structures. This follows from the fact that, no matter how cleverly the language designer, or specifier, arranges the elements of his Syntax Specification, the Analyzer will necessarily spend some percentage of its time exploring blind alleys. Clever specifications can make the blind alleys less frequent and shorter, but even for the simplest of languages, there will be some.

Thus, in any situation where the primary consideration is speed of the compiler itself, syntax-directed techniques are not the most suitable. But this, we argue, is true only if the quality of the coding produced is also of relatively little importance. In our experience with attempts to generate highly efficient optimized coding for several different machines, we find that the time spent in analyzing is a small fraction of the total—even using very sloppy Syntax Specifications. The most important question in compiling system design today, we reiterate, is not the "understanding" of the source language—that is a solved problem—but rather the generation of really good object-language coding.

One of the principal arguments in favor of syntax-directed techniques is that it is very easy to change the specification of the language, or, indeed, to switch languages, merely by changing the Syntax Tables—no modifications of the algorithms are required. And this is in fact true, with some restrictions. But now that techniques exist for automatically producing operator-precedence tables from a Syntax Specification [6], the syntax-directed compilers no longer have a monopoly on this useful feature.
A further advantage of syntax-directed analysis remains, up to the present, only potential. These techniques are evidently not of the "bounded context" sort—a syntax directed Analyzer can take into account as large a context as required to perform its recognition (admittedly, at a definite cost in speed). Hence, when the day comes that we need to perform analysis of source languages of much less rigid structure, syntax-directed techniques will be more immediately applicable than the techniques which are designed to take advantage of the restrictive properties of present programming languages.

In summary, syntax-directed techniques have a definite place in today's computing world, and promise to play an even more important role in the future.

APPENDIX

Some Further Transformations of the Syntax Specification and the Syntax Tables

In constructing the Syntax Tables, we described a complicated operation for avoiding the problem of left-recursive Alternatives in a Syntactic Type Definition. We can describe this as a transformation within the Syntax Specification itself, and, at the same time, include some features which improve the efficiency of the encoding of the Syntax.

First, we extend the idea of "Component" to include two new forms:

1) An Alternation (of one or more Constructions), enclosed in square brackets [' and ']. These brackets are assumed to be different from any of the Terminal Characters (if they were not, we'd use some other signs).

2) An Alternation enclosed in braces '{' and '}', again assumed different from any of the Terminal Characters.

Second, we apply a left-distributive law:

(T2) Whenever, within a single definition, two (or more) Alternatives start with the same Component (or sequence of Components), replace all of these Alternatives with a single one, whose last Component is the bracketed Alternative of the non-common parts of the original Alternatives, and whose first Component(s) is (are) the one(s) common to the original Alternatives. It is also useful to introduce the idea of a "null" Syntactic Type—effectively a Terminal Type which is always recognized whenever it is called for—denoted here by a capital lambda. Then, for example:

\[
\begin{align*}
  \langle a \rangle & ::= \langle b \rangle \langle c \rangle \langle d \rangle \langle e \rangle | \\
  & \quad \langle b \rangle \langle c \rangle | \langle b \rangle \langle c \rangle \langle f \rangle | \langle g \rangle
\end{align*}
\]

would be transformed into:

\[
\begin{align*}
  \langle a \rangle & ::= \langle b \rangle \langle c \rangle [ \langle d \rangle \langle e \rangle | \\
  & \quad \langle f \rangle | \langle g \rangle]
\end{align*}
\]

(Obviously, if the Analyzer is going to consider Alternatives in the order in which they are written, a null Alternative should always appear last in an Alternation.)

Having applied (T2) to any Definition, there can be at most one left-recursive Alternative, and if there is one, we can rewrite the definition according to:

(T3) Put the left-recursive Alternative as the last Alternative in the definition; if there is more than one other Alternative, put square brackets around the Alternation consisting of the nonrecursive Alternatives; delete the sign ',' preceding the last Alternative, and delete the first Component of that Alternative (which will be the same as the Defined Type of that Definition); then enclose the remaining Components (of this formerly last Alternative) in braces.

Thus the ultimate transform of a left-recursive definition has as Definiens a single Construction, the last Component of which is "iterated" (enclosed in braces). As an example, a Definition which was originally:

\[
\begin{align*}
  \langle a \rangle & ::= \langle b \rangle \langle c \rangle | \langle a \rangle \langle d \rangle \langle e \rangle | \\
  & \quad \langle a \rangle \langle f \rangle | \langle g \rangle \langle h \rangle
\end{align*}
\]

would be transformed into:

\[
\begin{align*}
  \langle a \rangle & ::= [ \langle b \rangle \langle c \rangle | \langle g \rangle \langle h \rangle] \\
  & \quad (\langle d \rangle \langle e \rangle | \langle f \rangle)
\end{align*}
\]

The modifications to the rules for constructing the Syntax Tables to represent Definitions in this form is left as an exercise for the reader.
The Analyzer flowcharted in Figure 1 should work on the resulting tables.

Three of the Definitions in Syntax II would be changed by application of (T2) and (T3):

\[
\langle \text{program} \rangle ::= \langle \text{assignment} \rangle \\
[ ; \langle \text{program} \rangle | \lambda ]
\]

\[
\langle \text{arith expr} \rangle ::= \langle \text{term} \rangle \\
( + \langle \text{term} \rangle )
\]

\[
\langle \text{term} \rangle ::= \langle \text{factor} \rangle \{ \ast \langle \text{factor} \rangle \}
\]

Now, an analogous pair of transformations—a right-distributive law, and the elimination of right-recursive Constructions—could be applied, and this would render the Syntax Specification still more compact. The language used by Brooker and Morris [10] for specifying syntax is essentially one of this sort, although the notation used is rather different.

More “Groundedness” Problems

The treatment we have described takes care of left-recursive definitions, as long as the recursion occurs within the definition of a single Syntactic Type. It will not handle the infinite-loop problem engendered by, as an example:

\[
\langle a \rangle ::= \langle b \rangle Z | X \\
\langle b \rangle ::= \langle a \rangle Z | Y
\]

and it is in general true that, for an Analysis technique of the “top down” sort, as presented here, a Syntax Specification with such left-recursive loops will not be adequate. This leads to the requirement of an additional condition on Syntax Specifications: If we say that a Construction is “firmly grounded” when its first Component is either a Terminal Type or a firmly grounded Defined Type, and a Defined Type is firmly grounded when all of its non-left-recursive Defnientes (in the Basic Syntax Specification) are firmly grounded, then:

(C4) Every Defined Type must be firmly grounded.

In practice, this is not a serious restriction. The simplest test for this condition is to try to run the Analyzer—it stops requesting input and goes into a loop. It is usually a simple matter to rewrite the Syntax Specification to eliminate the difficulty. In the above example, this could be done in several ways, one of which is:

\[
\langle a \rangle ::= \langle b \rangle \\
\langle b \rangle ::= [X | Y ] [Z ]
\]

A Modified Analyzer

The Analyzer algorithm of Figure 1 is designed to call the Generator upon recognition of every instance of a Syntactic Type, even if it is not the “longest” instance of that type present at the given position of the Input string. It turns out to be the case that, for all the standard programming languages, when the Analyzer needs to recognize a recursively defined Syntactic Type, it wants the longest string which is a member of that Type—that is, it should keep re-entering the iterated Component of the Definition (in our latest transformed form) until it meets a failure. The Syntax Tables and Analyzer described in this paper will find the “longest” instance of a type but this Analyzer does report each partial recognition also.

Now, a slight change in the Analyzer algorithm allows it to avoid reporting partial recognitions to the Generator, and call it only when it has completed recognition of the longest instance of a Syntactic Type. For those who might be interested in exploring this point, the changes to be made are:

1) Eliminate the boxes \(\text{GENERATE}\) and \(\text{STRUCT}[\text{SOURCE}]\) from the flow chart.

2) Insert a \(\text{GENERATE}\) box between the boxes \(\text{GOAL}, \text{SOURCE}\) and \(\text{SOURCE}=0?\)

The entire Syntax Structure Table entry \(\text{STRUCT}\) can also be eliminated.

In order to correctly record the recognitions, the Generator must construct a slightly different tree (we are here assuming operation in the “deferred generation” mode), the form of which is best illustrated by an example:

For the Input (sub-)string,

\[
A + B + C + D
\]

the Generator discussed in this appendix will produce a (sub-) tree:
If the modifications mentioned above are made in the Analyzer, each of the terms in the input string will cause the Analyzer to signal recognition of a \textit{<term>}, but only one recognition of \textit{<arith expr>} will be reported, and that after all four \textit{<terms>}s. The Generator will then tree this recognition as follows:

To use a tree of this form for the generation of output code, the Generator language must be extended, for example, to refer to "siblings" instead of just "sons" and "fathers" and also to "count" sons (the number per node is not fixed, but depends upon the actual input) or to recognize the non-existence of a "sibling," etc. Typically, the processing sequence for the above tree is for a \textit{<term>} node first to send control to its first son, to evaluate the sub-tree, then when control returns, to send control "horizontally"—to its siblings—for evaluation of the other \textit{<term>}s; only the last \textit{<term>} node would send control back to the father. References [1] and especially, [2], discuss this in more detail.

\textbf{A More Subtle Problem}

The Analyzer algorithm given in this paper has the following property: Assume that a given Syntactic Type \textit{<x>} has been recognized at character-position \textit{c} in the input string, and the Analyzer fails to recognize any of the possible successors of \textit{<x>} within the Definition of its current goal. The Analyzer will report failure for this Alternative of the Definition. It is possible to formulate a Syntax Specification in such a way that this reported failure would be erroneous: if there were another, different, sub-string starting at \textit{c} and which was also an \textit{<x>} (this other substring would correspond to an Alternative occurring later in the Definition of \textit{<x>} than the Alternative which was first successfully recognized). It is certainly possible to design an Analyzer which will keep track of the information necessary to allow this kind of "back-up" (see References [5] and [13]), but for the present purposes it would have encumbered the description with a good deal of additional mechanism—essentially, the Syntax Structure Table would have another item, encoding the converse of the relation represented by SUCCESSOR, and push-down storage would be required to keep track of the SOURCE lines of the Types successfully recognized, instead of just those which are currently being worked on. Using the Analyzer and Generator described in this appendix, it becomes much easier to ac-
commodate this feature, since the required additional information can easily be kept in the tree while it is being built. Reference [2] discusses this question in detail and with examples.

"Top Down" vs. "Bottom Up"

The Analyzer described in this paper is of the sort known as "top down," the appellation referring to the order in which the Analyzer sets its goals. The present Analyzer will always set as its next goal some Syntactic Type which appears as a Component in the Definition of its present goal — any time the Recognizer is called to find a Terminal Type, the pushdown storage of the Analyzer will contain a record for each Syntactic Type in a chain reaching down from the Starting Type to the Terminal Type. The order in which a "bottom up" Analyzer sets its goals is much more difficult to describe, but the actions of the two types can be impressionistically sketched as follows:

The "top down" Analyzer sets a goal and tries all possible ways of achieving that goal before giving up and replacing the goal with an alternative.

The "bottom up" Analyzer, having recognized a Syntactic Type, checks whether it has "gone astray" in trying to reach its goal or whether that Type is indeed a possible first Component of a first Component of . . . of the goal. If the latter, it continues processing input until it has built another Type of which the previous one is a first Component, and goes back to the checking. If it has gone astray, it backs "down" and tries to see if it can construe the input differently, to approach its goal along a different chain of intermediate types.

E. T. Irons' original syntax-directed Analyzer design was of this type (Reference [4]). It might be interesting to characterize an Analyzer similar to Irons' within the terminology of this paper.

We start with the Basic Syntax Specification, and first build a magical matrix which will answer the question "Can \( \alpha \) start with \( \beta \)" where \( \alpha \) and \( \beta \) are Syntactic Types. The relation "can start with" is defined recursively as follows:

\[
\text{A (Defined) Syntactic Type } \alpha \text{ can start with the Syntactic Type } \beta \text{ either }
\]

1) if \( \beta \) appears as the first Component of some Definiens of \( \alpha \), or

2) if there exists a \( \gamma \) which can start with \( \beta \), and \( \gamma \) occurs as the first Component of some Definiens of \( \alpha \). Irons' paper [4] gives an elegant technique for constructing this matrix. Note that \( \alpha \) can start with \( \alpha \), if it is left-recursively defined, but not otherwise.

The next step is to transform the Basic Specification:

First, remove the Defined Type and the sign '\:=' from the left-hand end of the Definition, and place '\:=' followed by the Defined Type at the right-hand end. (In effect, when a definition is considered from left to right, the Type which it defines is not known until all the Components have been recognized.) Hereafter, the Defined Type will be called the "Result" of the Definition. For example, the Definition of \(<\text{assignment}>\) becomes:

\[
<\text{variable}> = <\text{arith expr}> = :: <\text{assignment}>
\]

Second, apply the left-distributive law to the set of definitions, introducing Alternation signs '| ' as required. To illustrate, the following two interesting lines would result, in the example language of this paper:

\[
<\text{term}> [*:<\text{factor}> = ::<\text{term}> | \\
= ::<\text{arith expr}>]
\]

\[
<\text{variable}> [= <\text{arith expr}> = ::<\text{assignment}> | = ::<\text{factor}>]
\]

The effect of this transformation is to reduce the set of definitions to one line for each Syntactic Type which occurs as a first Component of one of the original Simple Definitions.

From the resulting set of "definitions" syntax tables are constructed, analogous to the ones in this paper. But the analogue of the Syntax Type Table is now a directory of first Components, each entry of which points to the first of a block of structure-table entries which encode the remainder of the "definition," now including a mention of the Result of the original Simple Definition (suitably flagged to avoid interpreting it as just another successor).
For use with a "bottom up" Analyzer of this sort, the Terminal Types of the language (or at least those which appear as first Components in any Definition) must be unambiguously recognizable independently of context—that is, the Recognizer may be told merely to "find something," and it will return with an indication of the particular Terminal Type it recognized.

To start the analysis, the goal is set to the Starting Type, and the Analyzer proceeds as follows:

**Step 1** Call the Recognizer; the Terminal Type it reports is placed in Type In Hand.

**Step 2** Can the goal start with the Type In Hand? If not, go to Step 4. If so, proceed to Step 3.

**Step 3** Consult the Structure Table at the point indicated in the Type Table for the Type In Hand. Push down the current goal and its source and set up as new goal the Component mentioned in this entry in the Structure Table. (This structure-table entry is the "source of this goal"). Go to Step 1.

**Step 4** Is the Type In Hand the same as the goal? If not, go to Step 7. If so, proceed to Step 5.

**Step 5** (We have attained a goal) Consider the source of this goal. Is the successor of that entry in the Structure Table flagged as a Result? If so, go to Step 6. If not, replace the goal with the (Syntactic Type mentioned in the) successor of the source, reset the source to point to this successor, and go to Step 3.

**Step 6** (We have recognized all the Components of a Definition.) Place the name of the Type mentioned in the Result entry into Type In Hand, pop up the goal (and source), and go to Step 2.

**Step 7** (We have "gone astray.") Consider the Structure Table entry for the source of the current goal. Does it have an alternate? If not, go to Step 8. If so, restore the input-string pointer to the value it had when the current goal was first set up, replace the current goal with the alternate (adjust source), and go to Step 1.

**Step 8** Pop up the goal and source, and go to Step 4.

For programming languages of the current sort, there is no clear advantage in favor of either the top down or bottom up analysis techniques, insofar as efficiency of the Analyzer is concerned. For either technique, it is possible to design a language and Syntax Specification on which the technique will perform very poorly, while the other one will not be nearly as bad. The choice between the techniques is generally made on the basis of considerations other than raw speed of the analysis, such as the kind of output desired from the analysis, the possibility of error detection and correction, or personal taste.

"Boostrapping"

As a final comment, we merely point out the fact that the language of the Syntax Specification is itself a rather straightforward, well-behaved language, easily susceptible of being described by a Syntax Specification. A version of the compiler can be written which uses a Specification of the Syntax-Specification-Language to "drive" it, and produces, instead of output code in a machine language, a set of Syntax Tables which encode the Syntax Specification it receives as input. This has, in fact, been done (References [1], [2]).

**BIBLIOGRAPHY**


