Teaching Compiler Design - Outline of a suggested course

Professor N. Wirth

Zurich.

These notes outline an approach to a course in compiler design through the study of a series of models which exhibit the essential features of current compilers. In such a course, we should be able to teach people to develop systematically new compilers. The primary objective of a system designer should be to retain an overall view of his proposed system, developing a model of the hard core of his compiler to which the details may be added later. No attempt at mathematical rigour will be made in this short outline.

The pre-requisite knowledge expected of a student attending this course could be provided by :

(i) an introductory course on basic programming (approximately 60 hours).

(ii) courses in data-structures, advanced programming techniques, analysis of algorithms, and machine organisation (total of approximately 90 hours).

(iii) a course on automata and formal languages (30 hours).

It will also be assumed that a student will have a working knowledge of set theory, logic and combinatorics.

The first idea that we should introduce is that of a language as a set of strings (and a string as a sequence of symbols). Here we introduce the basic notions of set theory, and consider sets A, B, C, ... whose elements are strings. We then define set union and set product (using conventional set notation), as

(union) $A \cup B = \{\alpha \mid \alpha \in A \text{ or } \alpha \in B\}$,

(product) $A\,B = \{\alpha\beta \mid \alpha \in A \text{ and } \beta \in B\}$

Later we shall use the abbreviated notations

$A = s$, meaning $A = \{s\}$,

and As , meaning $\{\alpha s \mid \alpha \in A\}$, for s being a single string.

A language can be defined through a system of set equations; and it is very easy to show that if you allow recursive definitions you can, with a finite set of equations, define an infinite set of strings and therefore an infinite language. We say at this point that a set of equations using only the operations of set union and set product define a type of language called "context-free".

The second notion to be introduced is that of a finite-state automaton (or f.s.a.), which can generate or conversely accept strings which belong to such sets. A f.s.a. can assume a certain finite set of states, and its operation consists of reading a symbol and, as a result, changing its state. We can immediately make an analogy between the definition of a language by a set of equations and an automaton. We write a transition rule for this f.s.a. as,

$$X\ y \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv Z$$

which means that if the automaton is in state X and reads symbol y it goes into state Z.

To illustrate, take a simple example :

$$\begin{array}{llll}
A\ 0 \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv B & \qquad & B\ + \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv C \\
A\ 1 \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv B & & B\ \times \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv C \\
C\ 0 \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv B & & B\ . \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv D \\
C\ 1 \relbar\joinrel\relbar\joinrel\relbar\joinrel\dashv B & &
\end{array}$$

Here we have a set of terminal symbols,

$$T = \{0 \;\; 1 \;\; + \;\; \times, \;\; . \} \quad ,$$

and a set of states,

$$N = \{A, \; B \;\; C \;\; D\} \quad .$$

We assume that the automaton starts in state A, and reads a finite sequence of symbols, "accepting" this string if it finishes in state D (the final state).

We can now demonstrate our analogy by transliterating the rules for the f.s.a. to :

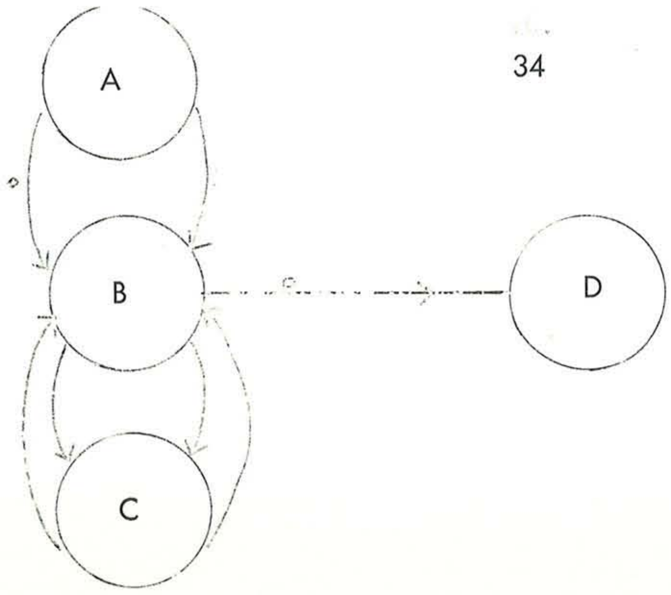
$$\begin{aligned}
B &= A \quad \cup A\,1 \cup C \quad \cup C\,1 \\
C &= B + \cup B \times \\
D &= B\ . \\
A &= \lambda \text{ (the empty string)}
\end{aligned}$$

We can make the correspondence between the language definition and this simple mechanism only if the equations have the form,

$$A = B_1\ s_1 \cup B_2\ s_2 \cup \ldots \cup B_n\ s_n$$

where the B_i are sets, and the s_i terminal symbols.

A third equivalent notation is a finite graph, whose nodes represent the states or sets and whose directed paths are labelled with the input symbols for the corresponding transitions. For our example, we have :



The concept of determinism can now be introduced; we say that our automaton is deterministic if for each state a given input symbol uniquely determines the action to be taken.

With such an automaton, we have both a generator and an acceptor for the language D (the notion of determinism is only essential for acceptors). At this stage, we can also introduce the concept of regular expression.

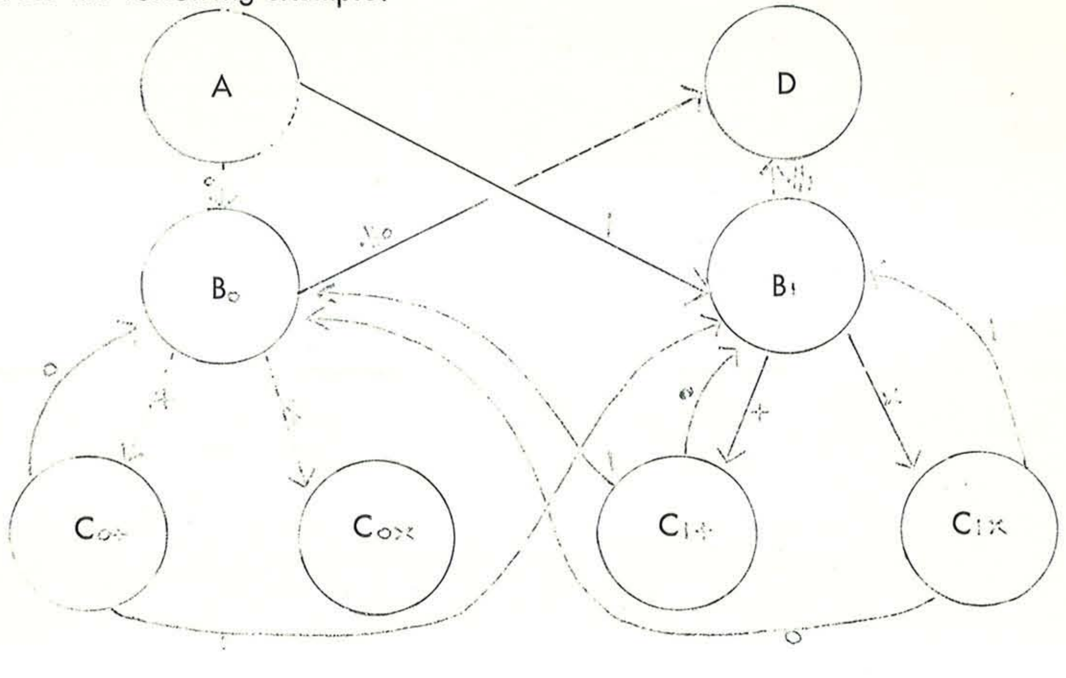
As an acceptor, our automaton either reads the whole string and ends in state D - in which case we say it "accepts" the string - or it reads the whole string, ending in some other state - in which case it "rejects" the string - or it ends in state D before reaching the end of the string - in which case it accepts a "head" of the string. We are still far from having a compiler model; we have not even a translator.

We now develop our model into one that writes as well as reads, by associating with each transition an output element. The rules for this expanded model are written as,

$Xy \longrightarrow Z \,||\, \zeta$

where ζ is some string (possibly empty) over a certain output alphabet. (We continue to write $Xy \longrightarrow Z$ for the case $X\,y \longrightarrow Z \,||\,$ [illegible]).

Consider the following example:



input symbols : O 1 + x .

output symbols: O 1

This deterministic model will produce as output a single symbol, O or 1, depending on the form of the input string.

It is quite easy to show that both this and the previous automaton accept the same language, which in regular expression form is given by :

$$(O \cup 1) \quad ((+ \cup x) \quad (O \cup 1))^*$$

An example of an acceptable string is :

$$O + 1 \times 1 + O \ .$$

The acceptable strings can be interpreted as expressions with the operators denoting addition and multiplication modulo two. Our second automaton gives this interpretation by virtue of the fact that for acceptable strings its output is just the value we would obtain through this interpretation. To see this, we note that whenever in state B_0, for example, the automaton has accepted a string with associated value O; similarly, in state C_{1+} it has accepted a substring with value 1 followed by a + symbol. If in the latter state it reads a 1 then it goes into state B_0 (1 + 1 = 2 mod 2 = O), and into state B_1 on reading a O (1 + O = 1 mod 2). Such an analysis can be applied to all states.

We now provide the additional notation required to introduce the output specifications into our set equations, which for the second automaton are :

$$A = \lambda$$

$$B_0 = A\,O \cup C_{0+}\,O \cup C_{0x}\,O \cup C_{0x}\,1 \cup C_{1+}\,1 \cup C_{1x}\,O$$

$$B_1 = A\,1 \cup C_{0+}\,1 \cup C_{1+}\,O \cup C_{1x}\,1$$

$$C_{0+} = B_0 \, +$$

$$C_{0x} = B_0 \, x$$

$$C_{1+} = B_1 \, +$$

$$C_{1x} = B_1 \, x$$

$$D = B_0 \,.\, [O] \cup B_1 \,.\, [1] \ .$$

In the last equation, the output strings are enclosed in brackets immediately after the corresponding term.

Introducing some notational abbreviations, we write V_o, V_1 for O, 1 respectively and write a combined rule for B_o and B_i as,

$$B_i = A\ V_i \cup C_{jk}\ V_1 \qquad \begin{array}{l} i = j + 1 \ \underline{\text{mod}}\ 2, \text{ if } k = + \\ i = j \times 1 \ \underline{\text{mod}}\ 2, \text{ if } k = \times \end{array}$$

where the second term is a union of terms whose subscripts satisfy the given constraints. The constraints express the interpretation we previously gave to the states.

It is important to distinguish between the use of the symbols O, 1, +, x where they occur as symbols in the transition rules and where they occur in their normal arithmetic sense in the constraints.

The final equations for our model have the concise form,

$$A = \lambda$$

$$B_i = A\ V_i \cup C_{jk}\ V_1 \ \big|\ j\ \Theta_k\ 1 = i$$

$$C_{ij} = B_i\ \Theta_j$$

$$D = B_i\ .\ V_i$$

where $V_o = O$, $V_1 = 1$,

$\Theta_o = +$, $\Theta_i = \times$.

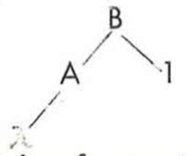
We have now shown that a language can be accepted by a simple model, and in a sense be evaluated by a model which is able to produce output and we observe that what is usually called "semantics" appears embedded in the syntax. If we try to do ordinary integer arithmetic in this way, the set of states for a corresponding "evaluating automaton" would become infinite; however, the same condensed notation could be used.

Let us now introduce, as an interlude, the notions of <u>top-down</u> and <u>bottom-up</u> analysis. So far, our set equations have formed a so-called right-linear system, each term consisting of a set symbol followed by an input symbol. This we can call a "source-oriented" notation, since the sets involved in a transition rule always contain as an element the part of the string which was <u>already analysed</u>. Equivalently, we can build up a so-called left-linear system which we can call "goal-oriented", wherein the sets will denote that part of the string which <u>remains to be analysed</u>.

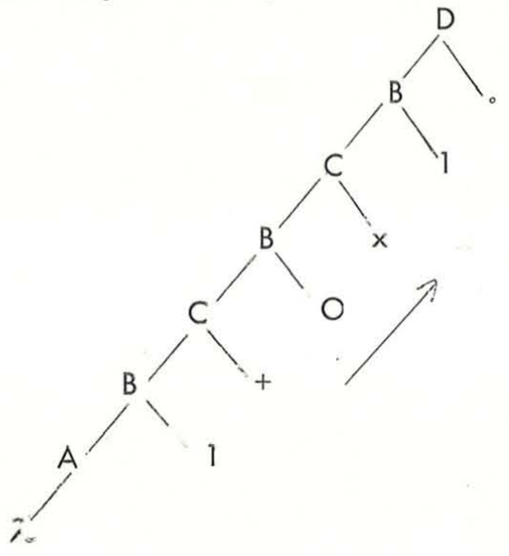
The two systems, left and right-linear, for the first example are shown below :

<u>left-linear</u>	<u>right-linear</u>
A = O B U 1 B	A = λ
B = + C U x C U . D	B = A O U A 1 U C O U C
C = O B U 1 B	C = B + U B x
D = λ	D = B .

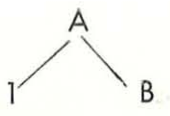
As an example, let us analyse the string 1 + O x 1 . with respect to both left- and right-linear systems. With the right-linear system, our analysis proceeds in the following manner: we start with set A (i.e. the empty string) and on receiving symbol 1 accept a string from set B :



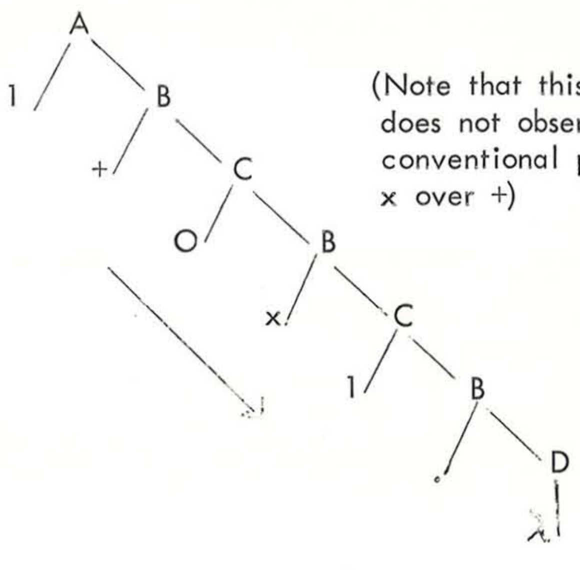
Our next input is a +, giving a string from set C; proceeding in this manner from left to right, we finally construct, from the bottom-up the analysis represented by the diagram below,



With the left-linear or goal-oriented system, we again start with set A and input symbol 1. From the equations the remaining string must belong to set B if it is to be accepted :



the next symbol + determines that the now remaining string O x 1. must be in set C, and in this way the following analysis is constructed from the top-down :



(Note that this example does not observe the conventional priority of x over +)

With a more complex language model, the distinction between top-down and bottom-up analyses is obscured It is for this reason that we choose to elucidate these concepts with the aid of a finite-state model. Although we can define with a finite-state model perfectly sensible languages, in practice more complex systems are required (e.g to deal with such features as nested structure).

We now introduce the <u>stack acceptor</u> (s.a.) which we describe by rules very similar to those of the f.s.a. We note that the principal property of the s.a. is its ability to store an ordered set of states on a stack, and thus we have the effect of having many f.s.a's active at a given instant. There are two types of transition rule, which we write,

type 1. $y \longrightarrow Y$ where $y \in T$, $Y \in N$

type 2. $\xi \longrightarrow Z$ where $\xi \in N^*$, $Z \in N$

T = set of terminal (or input) symbols
N = set of states

The effect of these rules on the s.a. is shown by the following diagrams ,

		input	stack	
type 1.	before	$y\ \alpha$	β	$y \longrightarrow Y$
	after	α	$\beta\ Y$	
type 2.	before	input α	stack $\beta\ \xi$	$\xi \longrightarrow Z$
	after	α	$\beta\ Z$	

To each symbol $y_i \in T$ we have a corresponding type 1 rule $y_i \longrightarrow Y_i$. In the following we make no distinction between y_i and Y_i, denoting the state Y_i by the symbol y_i. Thus we ignore the fact that type 1 rules exist and modify type 2 rules to refer to input symbols on the stack. However the occurrence of the symbol y_i on the stack denotes the state Y_i. We now give a simple example of a stack acceptor :

We have $T = \{0\ 1\ +\ x\ .\ (\ )\}$

$N = \{B,\ C,\ D\}$,

where the stack initially contains only the state $I = \lambda$ and accepts an input string only if by applying a finite sequence of the following transition rules it terminates with state D as the only item on the stack. The type 1 rules are implicit as described above. The type 2 rules are as follows ,

$0 \longrightarrow B$
$1 \longrightarrow B$
$B \longrightarrow C$
$(C) \longrightarrow B$
$C+B \longrightarrow C$
$Cx.B \longrightarrow C$
$C. \longrightarrow D$

The sequence of transitions through which this machine passes in accepting the string 0 + (1 x 1) + 0 is given by the following table,

input	stack	rule applied
O + (1 x 1) + O.	λ	type 1
+ (1 x 1) + O.	O	$O \dashv B$
+ (1 x 1) + O.	B	$B \dashv C$
+ (1 x 1) + O.	C	type 1
(1 x 1) + O.	C+	type 1
1 x 1) + O.	C+(	type 1
x 1) + O.	C+(1	$1 \dashv B$
x 1) + O.	C+(B	$B \dashv C$
1) + O.	C+(Cx	type 1
) + O.	C+(Cx1	$1 \dashv B$
) + O.	C+(CxB	$CxB \dashv C$
) + O.	C+(C	type 1
+ O.	C+(C)	$(C) \dashv B$
+ O.	C+B	$C+B \dashv C$
+ O.	C	type 1
O.	C+	type 1
.	C+O	$O \dashv B$
.	C+B	$C+B \dashv C$
.	C	type 1
λ	C.	$C. \dashv D$
λ	D	

The corresponding system of set equations which we obtain from the above acceptor are :

$$B = O \cup 1 \cup (C)$$
$$C = B \cup C + B \cup C \times B$$
$$D = C.$$

In a practical implementation we require a decision technique which selects a unique transition rule at each stage (e.g. precedence syntax analysis).
A sufficient condition for our s.a. to be deterministic is that the state on top of the stack and the current input symbol select a unique transition rule.

This last example has demonstrated a language which cannot be accepted by the finite state model. The s.a. of the example is source oriented; it is possible to describe a goal oriented s.a. which accepts the same language, although based on rules of a slightly different form. By adding output facilities to the s.a. we can describe a stack transducer to evaluate or translate the parenthesised expressions of this example. Suppose we wish to add the familiar notions of variables and of assignment through a language specification of the type,

$$
\begin{aligned}
T &= \{O\ 1 + \times\ (\ )\ ,\ \circ\ a\ b\} \\
N &= \{B\ C\ D\ E\ F\} \\
B &= O \cup 1 \cup a \cup b \cup (C) \\
C &= B \cup C + B \cup C \times B \\
D &= C \cup a \leftarrow C \cup b \leftarrow C \\
E &= D \cup E, D \\
F &= E.
\end{aligned}
$$

For output to be produced corresponding to the value of expressions from this language we must carry along the current value of the variables by some means. We do this by adding a value table T, and call the resulting model a <u>table acceptor</u>.

If we replace a, b by W_o, W_i, the transition rules corresponding to the assignments in the above language are :

$$W_i \longrightarrow B$$
$$W_i \leftarrow C \longrightarrow D$$

The value-table T is used in the constraint conditions accompanying the transition rules :

$$W_i \longrightarrow B_k\ ,\ k := T_i$$
$$W_i \longleftarrow C_k \longrightarrow D,\ T_i := k$$

where the assignment to W_i is mirrored by an assignment of the value of the expression (expressed by the suffix of C) to the table element T_i, and the use of the variable W_i in an expression is expressed by a transition to B according to the entry in the table T_i.

Thus we have a model which can evaluate languages which are not context free.

From now on, in the second part of the course, we are discussing the construction of actual compilers. Two questions are important here : how to make our analysis processes deterministic, and (more pragmatically) how to design languages not only from the user's point of view but so as to make their processors simple in structure.

In the case of finite state automata, we can show intuitively with the aid of examples that an equivalent deterministic automaton can be derived from a non-deterministic one - usually with the addition of extra rules. Unfortunately this is not the case for stack automata, and we find already that to obtain a deterministic process we have to restrict our language in some ways. However, these restrictions are usually avoidable for those features of programming languages we would like to have.

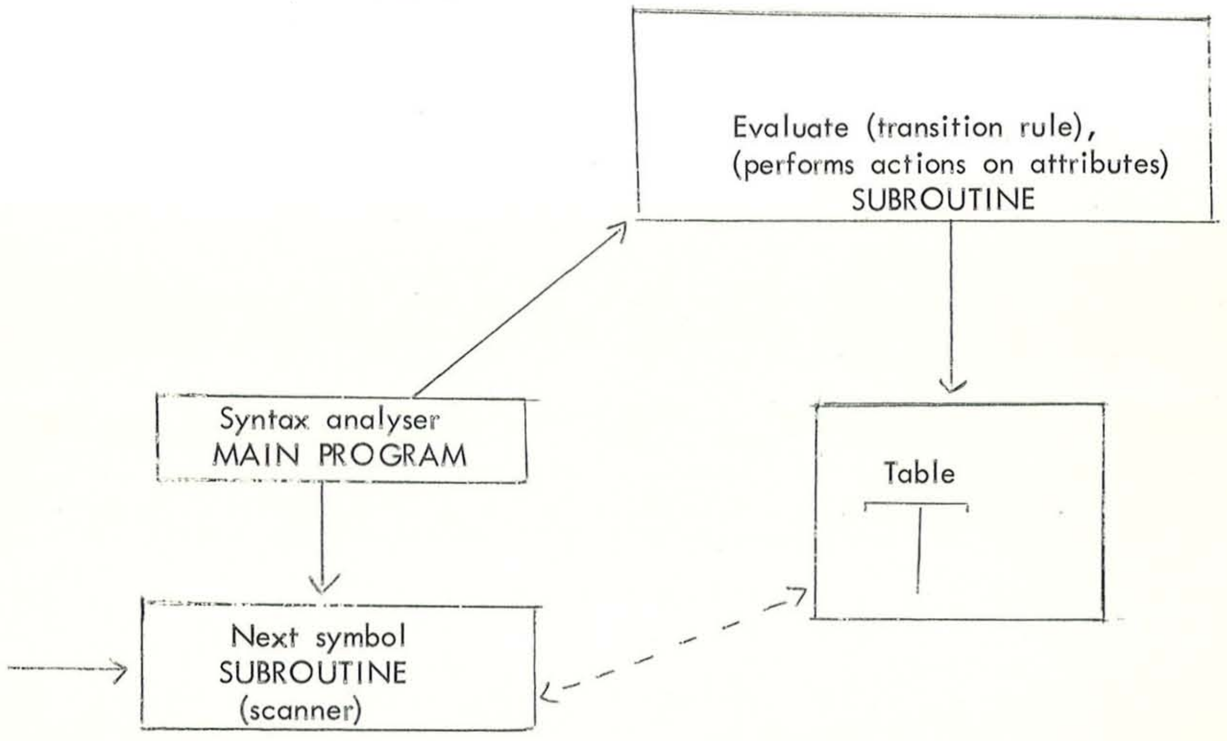
Again, in the case of an f.s.a. we have an efficient mechanism for the selection of applicable transition rules in the transition matrix. The combination of one state with one input symbol identifies directly the transition matrix element representing the new state. In the case of stack automata, we can introduce at this point the idea of precedence grammars analysis as an example of an efficient table-lookup mechanism in a rather similar way - the additional feature being that we must not only select the rule but also delineate how deep into the stack we must go to make the reduction. In effect, several "states" combined with the input symbol determine the new state. Although the precedence analysis technique does this efficiently, the price is paid in further restrictions on the language. However, the restrictions do in practice not turn out to be essential ones. A syntax processing program will check a given grammar, listing any precedence conflicts.

If we can use the precedence analysis algorithm to guarantee a deterministic process, then our remaining problem is the design of the evaluation or translation sections.

Here another program serves as a clerical aid, listing the productions in one column of a table, so that the language designer may enter in parallel columns the computations to be carried out on the attributes (associated with each symbol in the stack) for each production applied. This forces the designer to specify what the compiler has to do along with the syntax. We also obtain a listing of all symbols and their attributes; this enables e.g. an assistant to check the correct use of attributes

The compiler schema developed is shown overleaf:

The compiler schema.



The approach used is first to write and debug the syntax analyser and scanner. Then having designed and checked the language, the precedence tables are constructed, each action is translated into a statement in the 'Evaluate' procedure, and the compiler is now ready for testing. It is then necessary to write a test program that will cause each transition rule to be used at least once - if successful, the compiler is now debugged! An exhaustive testing of the compiler is essential.

Into what area have we now pushed the difficulties of compiler writing? (We cannot expect the whole process to become trivial!) The answer is : into the field of language design rather than compiler design. Ideally the same man should be concerned with both, developing grammars which are efficiently parsable and which also allow their "meaning" to be attached naturally to the production rules (e.g. via the 'indices' or 'attributes' introduced above). This task is not a trivial one, but we have found that the compiler schema described forces the definition of a language in a systematic way. In many cases, we have found that our difficulties with concepts and ideas which were initially unclear have been resolved under the pressure of this systematic approach. The solutions found (not only for the compiling system but also for the description of the language itself) have been altogether neater and more satisfactory than anticipated. We could say that this schema is a very good teacher!

In the discussion period following Professor Wirth's talk, questions concentrated on particular aspects of his existing systems and possible extensions. Professor Wirth mentioned that he was preparing a paper for publication on this subject.

Reference accumulated from this discussion and Professor Wirth's notes are :

Conway, M.E., "Design of Separable transition - diagram compiler", Comm. ACM 6 (July 1963), p. 396.

Floyd, R.W., "The Syntax of Programming languages - a Survey", IEEE Trans. EC13, 4 (August 1964), pp. 346-353.

Ginsburg, S., "The Mathematical Theory of Context - Free Languages", McGraw Hill, 1965.

Kurki-Suonio, R., "On some sets of Formal Grammars", Annales Academiae Scientiarum Fennicae A.I. 349.

Wirth, N. and Hoare, C.A.R., "A Contribution to the Development of ALGOL", Comm. ACM 9, 6 (June, 1966) , pp. 413-432.

van Wijngaarden, A. et al., "ALGOL 68", Tech. Rep. MR93 (April 1968). Mathematisch Centrum, Amsterdam.

Wirth, N. and Weber, H., "EULER, a generalisation of ALGOL and its formal definition. Part I", Comm. ACM 9, 1 (January 1966), p.13.