Automating the teaching of artificial language using production systems

Cornelia Boldyreff

Automating the Teaching of Artificial Languages Using Production Systems

by

Cornelia Boldyreff, B.A. (Leeds)

of the

South West Universities Regional Computer Centre

Bath BA2 7AY

ABSTRACT

The work to be described here is an investigation into the means whereby the learning of programming languages may be made easier. The role of formal definitions of programming languages is studied and a system is described which utilises production systems as the basis for generating an environment in which students may test their understanding of programming languages.

This system for automating the teaching of programming languages provides an experimental testbed for carrying out further investigations into programming behaviour.

Automating the Teaching of Artificial Languages Using Production Systems

by

Cornelia Boldyreff, B.A. (Leeds)

of the

South West Universities Regional Computer Centre

Bath BA2 7AY

A Thesis submitted to the University of Durham for the Degree of Master of Philosophy.

The research described in this thesis was conducted in the Department of Computing at the University of Durham from 1975 to 1978.

Thesis submitted 1983.

CONTENTS

DECLARATION

I hereby declare that none of the material contained in this thesis has previously been submitted for a degree in this or any other university, nor is any of it based on joint research.

STATEMENT OF COPYRIGHT

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. P. G. Barker, for his help in this work, and also my colleagues at Durham, in particular, Mahes Visvalingam and Malcolm Munro who gave me much advice and encouragement. Tim O'Shea has made several helpful suggestions to me concerning this work and I would like to thank him for his continued interest.

CHAPTER 1 The Role of Computer-Aided Learning in the Teaching of Programming Languages

1. Conventional Approaches to the Teaching of Programming Languages

Extending a classification used by John Barnes (1981), programming languages may be classified as being suitable for the amateur or professional programmer, or they may be largely of interest to the academic programmer. These classifications are not exclusively applicable. Some programming languages notably both BASIC (Kemeny and Kurtz, 1967) and Pascal (Wirth, 1971) were developed by academics for teaching and are now used by both amateur and professional programmers. These classifications are useful when consideration is given to how programming languages are taught; the methods of acquainting professionals with a tool they will use in serious software production vary from those appropriate for the amateur or student user of a programming language. This work is concerned primarily with the latter groups although many of the conclusions will be generally applicable to the professional programmer as well.

In the rest of this chapter, conventional approaches to teaching programming languages will be examined. Emphasis will be on methods which the student may use with a minimum of assistance. It is difficult to compete with the tuition supplied by a dedicated able individual teacher; however, as individual tuition in the field of programming languages is

uncommon, by considering those possibilities by which many students actually learn programming languages - textbooks, programmed learning material, and computer aided instruction systems - I hope to survey current approaches to teaching programming languages.

2. Programming Language Textbooks: Two authors' approaches to teaching BASIC

The two BASIC textbook that will be considered from the pedagogical view point are:

Illustrating BASIC by D. Alcock published by Cambridge University Press,

and

The BASIC Idea by R. Forsyth published by Chapman and Hall.

Although both of these introductory books are written for students without any prior knowledge of Computing, the approaches they follow are diametrically opposed. While both authors acknowledge the lack of standardisation in BASIC, Alcock accordingly treats the "need for portability as axiom" and describes the language in such a way that programs may be written without dependence on any particular version of BASIC. In contrast, Forsyth adopts the policy of sticking to one representative implementation of BASIC, namely DEC BASIC. This latter approach severely limits the utility of the book for students. Forsyth does give them

some indication of those features which are specific to DEC BASIC and unlikely to be generally available; nevertheless, whole chapters devoted to such features are irrelevant to students not using DEC BASIC on a DECsystem-10. To be fair to Forsyth, he does give students some guidelines to be followed when portability is required, but these occur almost as an afterthought in the last chapter. On the other hand, Alcock's approach is to enumerate several possible forms a statement may take and their associated effects. Given the various dialects of BASIC in use, Alcock's thoughtful analysis should prove a source of aid to students.

Both books contain numerous examples. Alcock makes good use of diagrams throughout. He switches between diagrammatic description and prose freely in a manner which is effective. Forsyth relies exclusively on the flow chart as his only diagrammatic aid to program understanding. As both authors offer an example which involves number conversion, their styles may be compared and contrasted with respect to their rendering of the solution. Forsyth begins with an abbreviated trace through the program for a given three digit number, and explains how this forms the basis of a more general solution. His example is concerned with converting from digital representation to the English prose description of the number. He makes no attempt to explain how the techniques used in this program may be usefully employed in other programs. Alcock's conversion program converts Roman numerals to their decimal equivalents. He uses this example

to demonstate the use of a symbol-state table emphasising that this technique is of general applicability and sketching a more complex extension. He also summarises the possibilities with respect to inputting a string of characters and extracting individual characters so that a capable student would be able to appreciate how the solution given could be modified if desired.

There is flexiblity in Alcock's attitude missing from the letter of Forsyth's text. Early on Alcock introduces recursion and gives an example illustrating the use of a stack along with the exhortation to try "playing computers" using a pencil, paper and a pocket calculator. Forsyth categorically states in his chapter on functions and subroutines that subroutines may be nested as long as they are not recursive. Although the spirit of Forsyth's last chapter is less rigid; his advice generally remains unadventuresome.

There are minor errors in the program listings in both books, students tutored in Alcock's critical approach will suffer less from these than those following Forsyth.

Generally the major difference between these two textbooks is that one attempts to illustrate BASIC in general while the other introduces BASIC through one particular implementation.

3. A Programmed Learning Textbook Approach to PL/1

The idea behind programmed learning textbooks is that the

students working through the material presented are able to progress through the lessons at their own individual rate. Depending on their self- assessed understanding, they can progress from concept to concept either directly or via additional explanatory material. The programmed learning textbook thus allows the more able student to work through material without being bored by unnecessary explanations, while the less able student is assisted by having a fuller account of the material.

The book I would like to examine here uses the programmed learning approach to teach a subset of the programming language PL/1 (Roper, 1973). This book, PL/1 in Easy Stages, progressively develops the notion of a PL/1 program for the student. The PL/1 language was designed with the aim of having uniformity in its syntax (Radin and Rogoway, 1967). Thus the language employs a general form for all PL/1 statements, and a uniform definition of procedures. Roper's book is good in that it emphasises this uniformity by giving rules throughout the text. However, the rules are simply given in English; and their use is not related to the exercises set. The book is also lacking in structure with regards to the presentation of the rules. The table below indicates the distribution of rules in the text.

Chapter	Pages	Rules
1	40	27
2	24	17
3	38	27
4	52	38
5	34	40
6	24	33
Totals	239	182

Throughout 239 pages, there are 182 separate rules. The student learning these rules would be better helped if they were related by a concise notation and more generally related to one another. Thus a student may successfully employ the rules given in the text and yet be left without an over-all understanding of PL/1 programs in general.

4. The LEARN Program and its Approach to Teaching C

4.1. The UNIX LEARN program

The LEARN program (Kernighan & Lesk, 1978) as provided with Version 7 UNIX is an interpreter based system which is driven by scripts. One set of scripts provides an introduction to the C programming language and is relevant as an automated approach to teaching a programming language. The interpreter implements a common strategy independent of the material being learnt. First, the LEARN program will be described simply as a method of computer aided instruction and then details of its approach to teaching C will be

given.

4.2. LEARN's approach to CAI

The LEARN program is based on the assumption that the way to teach people how to do something is to get them to do it. The student is shown examples and then required either to replicate the example or to produce a variant of it.

The LEARN scripts implementing lessons in a particular topic do not attempt to deal with incorrectness. They simply offer simpler examples on the assumption that by breaking the material down into smaller chunks of information eventually a point will be reached where the student can grasp the material successfully. Able students need never enter a remedial track, and can make speedy progress through the lessons. The LEARN developers acknowledge that the practice of subdividing material may be impossible and emphasise that the LEARN program should be seen as an ancilliary aid to be used in addition to reference manuals.

4.3. LEARN's approach to teaching C

Some of the lessons in the C script of the LEARN program are loosely based on material in the book, The C Programming Language (Kernighan & Richie, 1978), and refer the student to the relevant section on which the lesson is based. These lessons cover material found in Sections 1.1 to 1.9 i.e. the first chapter. The majority of the lessons are from an older C script, and are prefaced with a warning regarding their

poor quality.

Some scripts simply require a straight-forward answer, for example:

(Lesson 1.1d)

```
printf("\"£@\"");
```

Type "answer XXX", where XXX is the set of characters that will be printed.'

Most require that the student write, compile and execute a C program. The LEARN program through its C scripts is able to determine whether or not the students' programs have produced the correct results, but it is incapable of determining how the student has achieved these results.

These lessons reinforce a method of programming which achieves results by taking a program which nearly does what is required and by slightly modifying it achieving the required result. While this may be an expedient way of achieving results, it can hardly be said to give students much insight into the C programming language.

The fact that the C scripts used in LEARN have not been developed to cover the whole of the C language points to the difficulty of using the LEARN method to teach a programming language. While the LEARN program is adequate to illustrate by examples the usage of the C language, it does not have the potential for developing into a generalised system for describing programming languages for beginners.

5. Conclusion: The Area of Investigation and Scope of this Work

Two approaches to familiarising students with a programming language have been isolated. One which has been characterised above as the generalist approach aims to give the student a general form of the programming language from which the student may deduce particular programs. The other, the particularist approach, employs particular examples of programs and expects the student to form a general model of the language by the process of induction.

In this work, the efficacy of the former approach will be investigated. In order that the system may be used by unassisted students, a computer based system will be constructed. I will show how the general form of a programming language given by its grammar may be analysed by computer programs and form the basis of a system of subprograms which allow students to test their understanding of it. It is not proposed that the system developed should be a replacement for a human teacher, but it is intended to show that such a system is of utility in that it gives students an environment in which they may familiarise themselves with the grammar of a programming language and practice using it.

CHAPTER 2 Processing Grammars

371. Essence is expressed by grammar.

Philosophical Investigations

L. Wittgenstein.

1. Introduction

1.1. Scope of the Work

In this chapter, the design and implementation of two systems for analysing Context Free Grammars (CFGs) are described. The grammars to which this study is restricted are a subclass of Phrase Structure Grammars (PSGs) which Chomsky has described (1956). Chomsky's work will be reviewed briefly relating it to the characterisation of formal languages. CFGs have been used to define programming languages. Inadequacies of CFGs for this purpose have led to extensions. A new grammatical form, W-Grammars, developed by Van Wijngaarden will be discussed (1976). W-Grammars are double level CFGs which have proved very powerful in defining programming languages.

In connection with the processing of grammars, consideration is given to the problems of representing a grammar from the standpoint of choosing a formal representation which is easily understood, and choosing an appropriate data structure for representing the grammar which will facilitate its analysis and use by computer programs.

The purpose in analysing the grammars is to check firstly that they are well defined and secondly to check for various properties of the grammars.

This analysis is preliminary to using the grammars to produce automatically recognisers and generators for the programming languages described by the grammars. In the final section of this chapter, some results of work in this area are related.

1.2. Historical Review

In John Lyons' popular book on Chomsky (1970), he says that Chomsky drew on the branch of mathematics or logic which is concerned with formal properties and generative capacities of various grammars; and he notes that Chomsky made an independent and original contribution to the study of formal systems. Chomsky's chief contribution was to provide a definition and hierarchical classification of Phrase Structure Grammars (PSGs).

Informally, PSGs may be used to describe languages constituted of phrases, for example, English. A simple PSG for English sentences might be stated as follows:

```
sentence      -> noun phrase + verb phrase
noun phrase   -> adjective + noun phrase
noun phrase   -> noun phrase + connective + noun phrase
noun phrase   -> noun
verb phrase   -> verb
```

adjective -> old

adjective -> young

connective -> and

noun -> women

noun -> men

verb -> laughed

verb -> sang

From this grammar, the following sentences can be derived:

old women and men laughed

young women sang

old men laughed

(Note that already ambiguity has arisen. With the use of parentheses as phrase markers, this could be overcome, viz.

(old women) and men laughed

Note also that the noun phrase is recursively defined. This allows for the embedding of one noun phrase within another.)

Formally, a PSG is a system G, such that

$$G=(Vn,Vt,P,S)$$

where

Vn is the nonterminal vocabulary

Vt is the terminal vocabulary

P is a finite set $V+ \times V^*$

S is a member of Vn called the distinguished symbol

(Pairs from V+ x V* are written x -> y and read "the string x is rewritten as the string y", where x ϵ V+ and y ϵ V*:

$$V = Vn \cup Vt$$

(the empty set) $\emptyset = Vn \cap Vt$

$$V^* = \bigcup_{k=0}^{\infty} Vk$$

$V+ = V^* - \Lambda$ (the set containing the null string

* is Kleen closure; a special operation which yields all possible strings using elements of the set. {0,1}* is the null string, 0, 1, 00, 01, 10, 11,....).

Chomsky defined a hierarchy of PSGs by placing restrictions on the elements of P. (Martin, 1972)

In the most general case, there are in fact no restrictions; the grammar is of type 0. The restriction that the string on the left hand side of any production rule must be less than or equal in length to the string on the right hand side gives grammars of Type 1. A further restriction is to limit the string on the left hand side to a single element of the nonterminal vocabulary; this gives Type 2 grammars. The stipulation that all rules must be of the form: single nonterminal goes to single terminal element, or single nonterminal element goes to single terminal element followed by nonterminal element gives Type 3 grammars.

The languages described by Type 3 grammars are referred to as Regular or Finite State Languages. Chomsky has demonstrated in Syntactic Structures (pp. 21-24) (1957) that Type 3 grammars are inadequate to describe the generation of

sentences where there are relations between nonadjacent words, that is, where one phrase is embedded within another. An example of such a sentence is: 'Any sentence which contains an embedded clause cannot be described by a Regular grammar.'.

Context Free Languages have this property and they are described by Type 2 grammars; the rewriting of the nonterminal symbols using a Type 2 grammar takes place without any consideration of the context in which they occur.

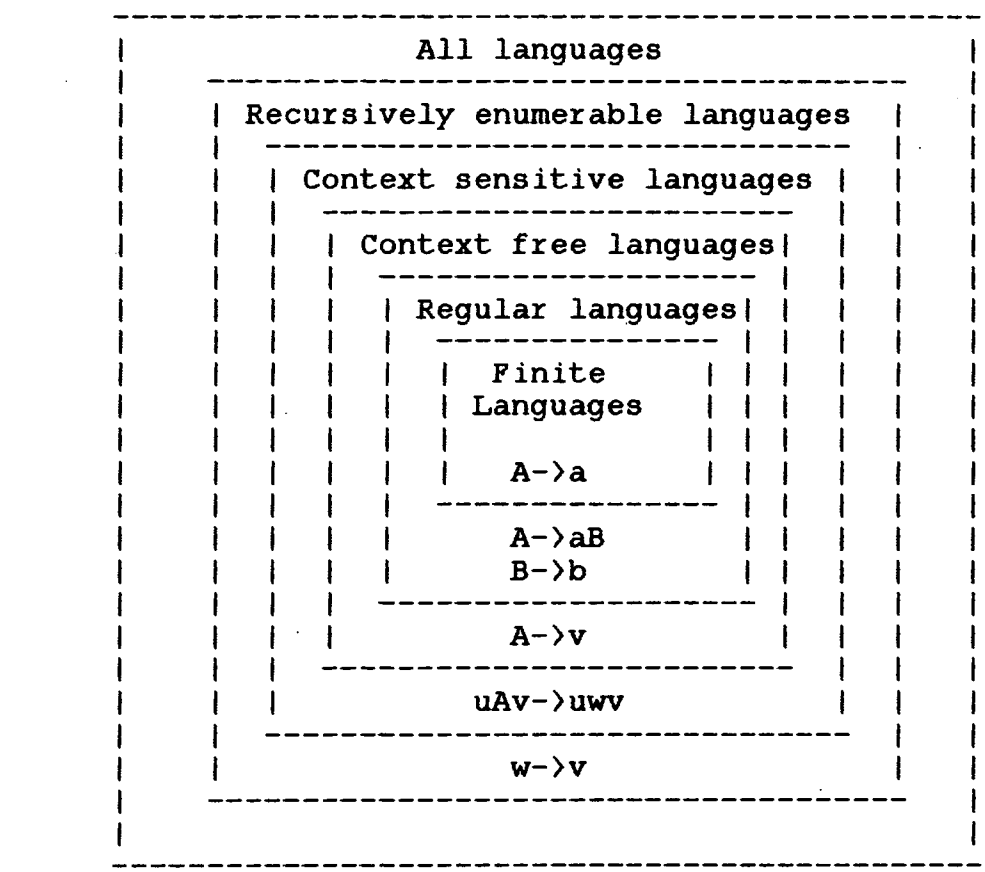
The contextual rewriting rules of Type 1 grammars allow Context Sensitive Languages to be described. Agreement in number between parts of speech is a familiar English language construct which we could employ a context sensitive grammar to describe. For example, such a grammar might include the following rules:

the noun flies -> the crow flies
the noun fly -> the crows fly

The languages described by Type 0 grammars are called recursively enumerable languages. Informally, this means that the elements of the language can be generated by a recursive procedure. The problem of determining for a given string whether or not it has been generated by a Type 0 grammar is undecidable; this is only the case for Type 0 grammars. For all other types, it is possible to decide this problem.

The relationship between languages is summarised in the

following diagram which also indicates the nature of the production set in their respective grammars:
(Adapted from Cleaveland and Uzgalis, 1973)



Where a,b are elements of Vt
A,B are elements of Vn
w is an element of V+
u,v are elements of V*

2. Using Grammars to Describe Programming Languages

In 1959, John Backus, a designer of the programming language Algol 60, developed a grammatical form equivalent to Chomsky's Type 2 grammar which is still widely used to describe the context free syntax of programming languages. Backus notation, BNF, has been used in some form for the

definition of every major programming language since Algol 60 (Cleaveland and Uzgalis, 1973). It has proved useful not only as a tool for defining and teaching programming languages but also as an aid to the machine analysis of programming languages. CFGs have been classified according to the method of parsing they require. Certain classes of CFGs may be used to automatically generate parsing programs. DeReemer (1969) described an early system which worked from a BNF description of a programming language.

There are two kinds of restrictions which CFGs in general cannot handle: static context conditions and dynamic context conditions (Cleaveland and Uzgalis, 1973). An example of a static context condition is the restriction that each identifier (i.e. name in a program) must be unique; a dynamic context condition is the restriction that identifiers occurring in an expression to be evaluated must refer to variables which have previously been assigned values (i.e. the reference of names must be fixed before their use).

To describe the static context conditions, a context sensitive grammar could be employed. This would not be satisfactory primarily for the same reasons which Chomsky gave when he rejected PSGs as a descriptive model of natural language; the grammars needed would be "extremely complex, ad hoc and unrevealing". (Syntactic Structures quoted by Lyons)

Only a formal definition of the semantics or meaning of programs written in a particular programming language would

spell out the dynamic context conditions. Usually, the semantics of a programming language are described in prose although limitations of CFGs have led to the development of formal techniques capable of defining the semantics as well as the syntax including any context sensitive aspects. One such formal technique is the grammatical form, W-Grammars, which Van Wijngaarden developed for defining Algol 68.

W-Grammars are two level CFGs. At the first level, the W-Grammar consists of what are known as "metaproductions" and "hyperrules" which are models for the production rules of the language; employing a "uniform replacement rule", the second level CFG is produced from hyperrules and metaproductions.

A simple example of a W-Grammar (taken from Cleaveland and Uzgalis) is the grammar which describes the language {anbncn} which is a type 1 or context sensitive language. The grammar is as follows:

```
(metaproductions)     N :: n;N,n.
                    ABC :: a;b;c.
(hyperrules)          S : Na,Nb,Nc.
                   nN ABC : letter ABC symbol, N ABC.
                    n ABC : letter ABC symbol.
```

By uniform replacement, the following production rules may be derived:

```
na : letter a symbol.
```

nb : letter b symbol.

nc : letter c symbol.

nNa : letter a symbol, Na.

nNb : letter b symbol, Nb.

nNc : letter c symbol, Nc.

The metaproduction rule 'N :: n;N,n.' produces arbitrary length strings of n. By uniform replacement in the first hyperrule, an infinite set of production rules may be derived:

s : na,nb,nc.

s : nna,nnb,nnc.

s : nnna,nnnb,nnnc.

and so on.

By convention, the typographical representation of any element ending in symbol is given by a table; these elements are the elements of the terminal vocabulary.

This example illustrates how a W-Grammar can deal with static context conditions. There may be a requirement to add dynamic context conditions. This is achieved in a W-Grammar by the introduction of predicates. Predicates are nonterminal elements which may be rewritten as the null string. Predicates may be generated from general predicates given as metaproductions.

A predicate may be used to express the requirement that every name in a list of names is unique; for example, this

might be achieved by the following predicate:

```
name list : name; namelist,comma,name,
                unless namelist contains name.
```

where 'unless false :.', i.e. 'unless false' is rewritten as the null string. Further rules omitted from this example would be required to spell out what it is for one notion to contain another; the predicate 'contains' could be rewritten in terms of the predicate 'begins with', and 'begins with' rewritten in terms of 'coincides with' which could be rewritten finally as either true or false, thus allowing the above predicate to be rewritten as the null string if it has been rewritten as 'unless false'. Note that no rule is given for the predicate 'unless true'; it is simply a blind alley and is a predicate which cannot be eliminated.

A full description of W-Grammars is beyond the scope of this chapter. W-Grammars have been demonstrated to be powerful enough to describe completely both the syntax and semantics of the programming language, Algol 68. J.E.L. Peck, one of the authors of the Revised Report on Algol 68 in which such a definition is undertaken, has produced an excellent short tutorial paper on W-Grammars which demonstrates their capabilities more completely than this text does (Peck, 1974).

3. Representation of Grammars

These remarks are confined to the notational systems employed to represent CFGs. The system with the arrow as the

production symbol and plus as the concatenation symbol is from Chomsky. In BNF, '::=' is the production symbol; concatenation is implicitly represented by writing elements of the vocabulary next to one another in a rule; rules for the same nonterminal element are condensed into one rule with '|' as the or symbol; and nonterminals are enclosed in angle brackets. Thus, a rule defining a number might be written in BNF as follows:

<number>::=<digit>|<number><digit>

BNF notation was extended to give an alternative form to some simple recursive rules, and to distinguish more clearly alternatives and options within a rule. The extension was the result of marrying BNF with the metalanguage developed to describe the programming language COBOL; it has been praised for its "utility and cleanliness" (Cleaveland and Uzgalis, 1973). An example of a rule in this notation is as follows:

NUMBER::=[SIGN] DIGIT...

where the dots mean the occurrence of the immediately preceding element one or more times and the square brackets indicate an optional element.

The CFGs of W-Grammars are notationally equivalent to BNF grammars with the exception that a symbol to indicate the end of a rule, a full stop, has been added. In W-Grammars, a semicolon is used for the or symbol; a comma is the explicit

concatenation symbol; there are two production symbols colon and double colon; and by convention terminals end in 'symbol'. Thus, in W-Grammar notation, the following rules would define numbers:

```
     NOTION::digit.
NOTION sequence:NOTION;NOTION sequence,NOTION.
         number:digit sequence.
          digit:zero symbol;one symbol;.....;nine symbol.
```

with a table showing the particular representation of each symbol in the language.

The notations described above for CFGs with the exception of Chomsky's were developed by the designers of various programming languages as an aid in the specification of the language for both future implementors and to describe the language to future users.

Implementors of a programming language are concerned with the implementation of the language on a machine which as Marcotty et al point out is "after all a kind of formal definition" (1976). Unfortunately, it is sometimes the only definition to which users may appeal. In implementing a language, system programmers construct compilers; these are programs which consist of recognisers for that language and specify what actions are to result for all recognised programs.

While W-Grammars have the advantage that they can be

employed to completely define a programming language, the defintions resulting are somewhat incomprehensible to the uninitiated. Cleaveland and Uzgalis have attempted an introduction to W-Grammars in the hope that more programmers will come to appreciate their power. They also express the hope that work will begin on "automatic parsing techniques which could automate W-Grammar definitions and provide giant advances in automatic compiler construction and in the development of far more responsive and facile computer languages" (Cleaveland and Uzgalis, 1973).

Addressing this problem of automation of formal definition allows the vexed area of the human engineering of the definition to be left behind as consideration is given to the problem of how best to represent a definition so that it may be automatically processed with ease. Here the crux of the problem is to choose a data structure and/or data type which will reflect rather than obscure the form and content of the data which in this case will be the rules themselves. Data represented within a computer is given form and content by the programming language structures and data types which allow for the interpretation of the data in various ways.

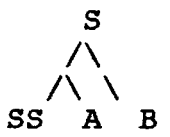
Perhaps the most straightforward method of represention is to process the rules as strings of characters using a linear data structure. Analysis is facilitated by functions for indexing and forming substrings. In my first program which was written in the programming language PL/1 to process the

rules of a CFG in W-Grammar notation, the data was represented as character strings declared to have the form of variable length character strings. Character strings in PL/1 are one dimensional, and characters are distinguished by their rank or index within a string, such that the first character has index 1 and so on. The built-in functions, INDEX and SUBSTR, were used for determining the index of one character string within another, and forming substrings respectively (PL/1 (F) Language Reference Manual, 1969).

This method of representation was not found to be particularly satisfactory because of the effort involved in extracting information from the rules. For instance, when indexing the characters to the right of the colon in the string, 'S:SS,A,B.', for an occurrence of 'S' indicative of recursion, the INDEX function will return the index of the the first 'S' in the element 'SS' which is a distinct element from 'S'. Thus, it is necessary to include delimiters round the string being indexed for, that is to say, in the case of a search for a generally recursive rule, index for the string, ',S,'.

By rewriting the program in the LISP programming language, it was possible to overcome these difficulties by representing each rule as a list of atoms which were the elements of the nonterminal and terminal vocabulary. In LISP (McCarthy et al, 1965), an atom is considered to be an indivisible item of data, so that the above problem does not arise. In

order to do the analysis in LISP, each rule in the W-Grammar had to be converted into the form of a LISP list. The following illustrates the conversion process, which was in fact carried out by a translation program written to transform rules already represented as character strings for the first PL/1 program. Consider each rule in W-Grammar as a tree structure with the nonterminal on the left hand side as the root and the alternative on the right hand side as branches where a branch consists of elements of the vocabulary. Thus, the rule,



In nested parenthesis tree notation, this is the tree (S (SS A) (B)) which coincidentally is the way this tree is expressed as a list in LISP. The branches of the tree may now be processed as sublists. A set of LISP functions may be defined which allow testing for properties of the rules directly by treating the sublists as sets of vocabulary elements. This overcomes the indexing problems mentioned above. As many automatic parsing programs employ trees as a data structure to represent production rules, this method is not original.

The notation of LISP has been criticised as consisting of Lots of Insignificant Silly Parentheses. In this application, the notation was straightforward to use and use of the translation program ensured the lists were well formed. LISP

is a functional calculus with "programs" taking the form of functions which are themselves lists written in the list notation. This use of the same structure for both "programs" and "data" has an attractive simplicity.

In summary, representation of a grammar in a notation which facilitates its use as a source of information to programmers is problematic although it is undoubtedly true that a good formal definition is an important factor in determining the ease of learning and using a programming language; the following chapter will return to this point and give it fuller consideration. Choosing a machine representation is a less vexed question and related to the sort of functions which will be applied to the data when it is processed. The processing of CFGs as linear character strings does not exploit the tree structure inherent in the rules which a list structure is capable of revealing.

4. Analysis of Grammars

The CFGs may be completely defined by the production rules with the convention that the rule for the distinguished symbol is given first. The elements of the nonterminal vocabulary in the W-Grammar form of the rules will all appear only once on the left hand side of the rules. The terminal vocabulary is the set which is the difference between the set of all vocabulary elements and the nonterminal elements.

As the rules are individually processed, the analysis pro-

gram builds up these sets. Rules are analysed branch by branch; each branch is checked for left recursion, self-embedding, right recursion, circularity, ambiguity and uniqueness. These properties are all illustrated in the following rule:

```
  S : S, A; A, S, A; A, S; S, A, S; S; A; A.
      |        |        |     |     |     |
      | self-embedded    |  ambiguous|  not unique
      |                  |           |
left recursive   right recursive  circular
```

A check must be made that there is at least one nonrecursive branch in each rule, so as each rule is processed a count is kept of the nonrecursive branches.

After all the rules have been processed individually, relations between the rules may be analysed. A check is made for all the properties mentioned above occurring indirectly. For example, the following might occur:

S:T,A;A. and T:S,B.

In such a case, S is said to be indirectly left recursive.

Any branches which do not ultimately end in terminal elements are marked as incomplete. Any rules for nonterminals which do not occur on the right hand side of some rule with the possible exception of the distinguished symbol are marked as superfluous. If a reduced form of the grammar is required, all superfluous rules and incomplete branches would need to be deleted (Gries, 1977).

The manner in which these checks are carried out is as I have indicated in the discussion on representation partially a function of the programming language used to write the analysis program. It was in my case also a matter of experimentation; at first, I chose the simple linear structure in preference to a more complex nonlinear structure because it seemed sufficient to represent the grammar for my purpose of analysis. Later, I experienced difficulties extracting information from the strings and decided to try a data structure which allowed the structure of the rules to be represented directly.

To illustrate some differences resulting from the linear and nonlinear representations, a brief discussion on how each program checks a rule follows. The PL/1 program relies heavily on the use of the built-in functions, INDEX and SUBSTR. Each rule identified by its terminating full-stop is read in by the program which indexes the rule for a colon. All characters preceding the colon are taken as an element of the nonterminal vocabulary; adopting Van Wijngaarden's terminology, this is referred to as the notion defined by the rule. The rest of the rule following the colon is processed as follows: first, the full-stop at the end is replaced by a semicolon and then indexing to the next semicolon, each branch can be differentiated. Branches representing each alternative are formed as substrings and they are processed in turn. The checks for recursion, circularity and ambiguity are carried out by indexing for

occurrences of the notion suitably delimited within the sub-string.

In the LISP program, the same analysis is carried out by rather different means. A rule in list notation is read in by the LISP READ function which reads in complete lists. By applying the primitive LISP function CAR to the rule, the notion at the head of the rule is obtained. Application of the function CDR to the rule results in the formation of a list which is the list of all the branches associated with the notion. By applying CAR to this list, the first branch is obtained. By applying CAR to the first branch, the first element of the branch is obtained, and so on. The checks for recursion, circularity and ambiguity are carried out using the functions MEMBER and DIFFERENCE with their usual meaning over the lists as sets.

The analysis of relations between the rules was only implemented in the LISP version of the analysis program. It was considered that the most effective way to do this in PL/1 would have been to introduce a list structure for each rule using pointers and as this is readily available in LISP, the switch to LISP was deemed sensible.

To complete this phase of the analysis, the LISP program works through possible derivations starting with the rule for the distinguished symbol and ensures that each nonterminal eventually goes to a set of terminal elements. Indirect occurrences of the properties checked for in the branches

are also checked for at the same time. These properties can be recast in terms of set relations which must hold between various lists which are constructed from the grammar. In the analysis, rules and branches which would be deleted to reduce the grammar are merely marked out; they may be indicative of an incomplete or tentative definition, or they may be simply superfluous.

These analysis programs were tested on various simple grammars which were devised for testing the programs and on the grammar of the programming language SEQUEL (Chamberlin and Boyce, 1974). By analysing the SEQUEL grammar, it was possible to identify construction patterns within the language which enabled the grammar to be modified so that common construction rules after the fashion of Van Wijngaarden's hyperrules could be employed. Further details are given in the following chapter.

5. Production of Generators and Recognisers from Grammars

My interest in analysing grammars grew out of programs I developed to produce generators and recognisers from CFGs. These programs were written for use within a tutoring system which was designed to instruct students in the application of grammar rules. Chapter 3 gives the background of this work and a full description of the resulting system and its use.

The method of generating expressions from a CFG is straight-

forward. Starting with the distinguished symbol, the program must work through rewriting nonterminal elements on the right hand side of the rule until a string containing only terminal elements is generated. Problems can arise where the language is defined in a highly recursive manner as this allows for the generation of very long complex expressions. To overcome this the generator may include restrictions on the length of the expressions to be generated. In my programs, usually a limit on the length of expressions has been imposed although experiments were conducted in which the selection of the next branch to be employed in the derivation was completely random.

My program for producing generators forms functions for each nonterminal element; these functions return character strings corresponding to a derivation for a particular nonterminal. Both the program which produces the generators from a grammar and the generators produced are in PL/1.

I incorporate hyperrules into the generator as functions with functions as arguments. For example, I have defined a function, SEQUENCE, which requires as its argument a function for a nonterminal and returns a character string corresponding to a sequence of the derivations of that nonterminal.

The production of recognisers is a much more interesting project. It is bound up with the wider area of interpretation and compilation of programming languages. In discussing

the execution of a program by a conventional computer, Dana Scott described a compiler as something which is applied to a program to produce by syntactical manipulation the machine code equivalent (Scott, 1974). In order to carry out this transformation, the compiler must possess the capacity to recognise expressions in the language in which the program is written; recognition is effected by parsing the program in symbolic form. If our aim is compile the program, the parsing given by one grammar may be more useful than that given by another; for example, an unambiguous grammar allows the parser to identify grammatical errors more clearly than an ambiguous grammar.

Assuming the grammar as given, a choice remains as to the method of parsing. while various methods may be used, I shall only describe a method which I find particularly elegant. In passing, I should mention that a more critical defense of this method can be found in a paper by Turner (Turner, 1975). Its relevance here is that the tutoring program described in Chapter 3 uses this technique. The method is known as "recursive descent", and I will show how it can be implemented automatically working from a machine readable description of the rules.

From the following example, it should become clear why this method is called "recursive descent". Consider the grammar given by the following rules:

sentence : subject, verb; not, sentence.

subject : all, cats.

verb : are, greedy.

Two derivations are:

all cats are greedy, not all cats are greedy

To recognise a sentence, we look for a subject. To recognise a subect, we look for an "all" followed by "cats". Once a subject has been found, it must be followed by a verb. To recognise a verb, we look for an "are" followed by "greedy". If the search for a subject fails, we look for a "not" which must be followed by a sentence. To recognise a sentence, we look for for ..... as above.

To implement recursive descent, the recogniser requires a procedure for recognising every nonterminal element of the vocabulary. The recogniser may contain separate procedures for recognising each nonterminal element; these procedures will reflect in a clear way the right hand side of the rules and if a nonterminal is recursively defined, then its recognising procedure will be recursive. Alternatively, the recogniser may consist of a single recognising procedure which directly utilises the grammar rules as data. In the method of parsing by recursive descent, recognition proceeds strictly from left to right through the expression, looking at one symbol at a time without having to look ahead or back; such a recogniser is decscribed as deterministic (Wirth, 1976). It is not possible to construct a deter-

ministic recogniser without look-ahead working from an ambiguous grammar (Rifkin, 1978). Wirth has formulated the properties which a CFG must have if a deterministic recogniser is to be constructed for the language it describes. He formulated the properties in terms of set relations which must hold over the elements of the vocabulary. These relations may be checked out by an automatic analyser such as the LISP program already described. If they do not hold, it is often possible to transform the grammar without affecting the language, although in some cases it may be necessary to alter the language. With the exception of the following example, no attempt will be made to describe in detail any of the well known methods of transforming grammars; these are described in Foster's Automatic Syntax Analysis (1970) and elsewhere (Gries, 1977; and Wirth, 1976).

An example of one transformation which a CFG may require is factoring. Consider the grammar given by the following rules:

S : A, B.

A : x, A; y.

B : x, B; z.

This grammar as it stands is unsuitable for direct production of a deterministic recogniser without look-ahead because both "A" and "B" may begin with the same terminal element, namely "x" and we cannot recognise either until the last terminal element has been recognised. By factoring, the

grammar is transformed into the following equivalent form:

S : C; x, S.

C : y; z.

Using these rules, the recogniser either successfully recognises a "C" or recurses past "x"s until it recognises a "C".

The program which produces recognisers does so in a method similar to the one by which generators are produced. It forms functions which look for elements of the vocabulary returning true if the search is successful and false if it is not. The functions all require one argument, the current element under consideration from the expression to be recognised. this is supplied by a function common to all the recognisers, NEXT, which returns the next element from the input expression. All these functions and the program producing them are written in PL/1.

PL/1 was used because the core of the original tutoring program was written in PL/1 and by writing all the subprograms in PL/1, many common procedures could be re-employed. This also enabled the frame of the original tutor program to be used in a more general way with other grammar rules.

Automating the production of recognisers from grammars is a first step towards the fully automatic implementation of programming languages. This modest claim can be made without committing the "fallacy of the first step" of which Dreyfus (1972) accused the early machine translation workers.

6. Conclusions

I have described the role of grammars in programming language definition showing how these grammars may be represented to reflect their structure and analysed to determine their properties. The analysed grammar already represented as a data structure may then be transformed if analysis indicates this is necessary and used directly to produce generators and recognisers for the language described by the grammar. The latter are especially important because of their use in compilers.

A system which undertakes the analysis of grammars, while interesting in its own right, also provides a diagnostic aid to the propounders of grammars. It is obvious that careful formulation of the grammar of a programming language from the start is desirable. This avoids nonstandard implementations based on arbitrary decisions where the definition is ambiguous, and prevents the development of incompatible dialects.

The day of a universal programming language is unlikely; however, communication about a particular programming language is made difficult if the language is not defined in a manner which is, "complete, clear, natural, and realistic", to paraphrase Marcotty et al (1976). I hope to have demonstrated that the role of CFGs and in particular W-Grammars, in the definition of programming languages both from the standpoint of describing the language for potential

users and with an eye to facilitating the implementation of the language.

We may hope for the day when there is a generally accepted formal technique for defining programming languages. I am reminded of Leibniz's remarks On Method (in Leibniz Selections, Weiner(ed), 1951); he envisaged the time when by applying a universal method, disputes could be settled by calculation. All that is required is the settling on a universal language for use in formal definitions; of course, ideally our calculations in the universal language should be capable of computerisation.

CHAPTER 3 Generating Programming Environments for Learners

1. Overview

In the mid-1960s, in order to study 'rule learning', Professor G. A. Miller at the Harvard Center for Cognitive Studies in a project entitled Grammarama programmed a computer to conduct experiments with artificial grammars. (Miller, 1970). In 1965 Donald Norman and John Schneider (referred to by Miller, pp. 169-173) used computer programs to study the most effective way to decompose a grammar into rules so that it might be learned by identifying correct and incorrect productions. The work to be described in this chapter has been concerned with teaching students how to apply the syntactic rules of the class of artificial languages known as "programming languages"; it is not expressly concerned with the problem of how they might discover these rules, although in my work, it has been useful to draw on the methodology of Miller's Grammarama.

The prototype system was a program to teach first year Computing students how to form logical expressions using a notation for these derived from the PL/1 programming language. The program introduced itself as a game in which the student could opt to be either the producer of expressions for checking by the program or the checker of program-generated expressions. A score was kept by the program; this was summarised when the student decided to finish. The rules for forming logical expressions were

available throughout each session. The program was capable of entering an explanatory mode in which it explained how it generated expressions from the rules and how it recognised or checked expressions, if the student's responses suggested this would be helpful. At this stage, the program was simply an extended version of Miller's Polish notation program used in the Grammarama Project. It was tested with first year undergraduate Computing students, and found satisfactory within its limited goals.

During these studies, it became apparent that such a system could be used to teach almost any formal language syntax provided that the sub-programs for rule presentation, expression generation or production, and expression recognition or syntax checking were slotted into the original program. To automate this process, a program has been developed which works from a definition of the syntax of a programming language and generates appropriate sub-programs to be used in the framework of the original syntax game program.

The sub-program generator was tested with the SEQUEL language as described by Chamberlain and Boyce (1973). The resulting program to teach SEQUEL brought out the need to clarify exactly how expressions of specified difficulty can be produced, and from these studies I developed a use of production systems to describe the general teaching strategy. Thus the method of syntax game programs has been generalised, so that both the programming language to be taught

and the teaching strategy to be used in the production of examples are both expressed by sets of production rules. The use of production rules to describe the teaching strategy allows the syntax game programs to act as test-beds for different teaching strategies.

THe production rules which form the input to the sub-program generator are expressed in Van Wijngaarden's notation, W-Grammar, already discussed in Chapter 2. The sub-program generated to present the rules may translate the rules if desired into another notation, BNF or syntax diagrams; this flexibility allows for experiments to be carried out using different notations, for example to determine notation preferences if any exist.

The purpose of these studies has been to provide systems for generating programs which can be adapted easily to individual student's needs. Some students may benefit from the elegant richness of the full two-level W-Grammar notation while others only require the modest economy of BNF notation.

While it is not particularly worthwhile to attempt to teach the complete syntax of a programming language in this way, the complexity of expressions presented and checked would not be suitable for presentation and checking in an interactive mode. For simple languages and for sub-sets of more complex languages, a syntax game is a useful learning vehicle. It not only introduces students to the syntax of a

particular language, it also familiarises them with the use of a formal definition.

2. Historial Background

Miller's interest in artifical languages arose out of work undertaken in the late 1950s with Noam Chomsky. In 1957, they collaborated in a study of algebraic systems which Chomsky then called "finite state grammars". Chomsky's work in this field is described in Chapter 2 of this work.

Miller assumed that when people learn a natural language, they do not memorise all the particular sentences that comprise it; rather they learn rules for producing and interpreting any sentence. In order to investigate "rule learning", he began to experiment with artificial languages. He described his method as inductive in that the subject could only obtain information about which sentences were part of the language and from this, by induction, had to learn the rules. In Miller's case, these were PSG production rules.

Miller was quick to see the advantages of automating his experiments. It was found from the start, for example, that human experimenters were simply not fast enough nor accurate enough to run the experiments if grammars of any complexity were used. In addition to speed, Miller noted that the subjects had great faith in the computer and appeared to believe it would not trick or cheat them:

> "I find it remarkable that an intelligent college student will let a machine tell him repeatedly that he is wrong without losing heart or face; if a human experimenter told him the same thing, he would seethe with indignation."
>
> (Miller, 1970, p.159)

In evaluating his automated experiments, Miller had the insight to distinguish between people learning a language and people learning to make the machine respond in a certain way. It is, of course, possible to do the latter without a complete understanding of the language; and it is important to bear this distinction in mind when assessing the claims of any automated teaching system.

As the complexity of the grammar increased to the point where it became impossible to learn (inductively) the whole grammar at once, Miller considered using the strategy of teaching rules one by one and combining them later. Some work was done along these lines by Norman and Schneider, who used a context free grammar and found that Polish notation was more easily learned when the rules were learned individually. The three rules they taught were:

(P1) S->P

(P2) S->NS

(P3) S->ASS

or in BNF:

<S>::=P|N<S>|A<S><S>

Miller postulated that decomposing the grammar to be learned into a regular grammar with infinite rules would be of lit-

tle help to learners; to Miller, a grammar with infinite rules was ridiculuous. At almost the same time, such a grammar, W-Grammar, was being developed and used to describe the then new programming language, Algol 68 (Van Wijngaarden, 1976).

As indicated in Chapter 2, grammars have been used to describe programming languages since the late 1950s. Since their inception, context free grammars describing programming languages have provided a useful teaching aid. Because of their similarity in form to dictionary definitions, most people find the use of a context free grammar almost intuitive, and so refer to it as naturally as they would to a dictionary to settle exactly how any particular notion in the language has been defined.

3. Scope of this Work

Miller's inductive method of rule learning may be compared with the way in which many people learn a programming language. For the most part, beginning programmers have no understanding of explicit grammatical rules for describing the languages in which they are programming. Like Miller's subjects, they submit their attempts at program production to the compiler and it responds by identifying correct productions and signalling errors if any occur.

Beginning programmers may be concerned only with getting results and may not wish to gain any more of an understand-

ing of the language than is necessary for their immediate goals. This attitude is acceptable for "one-off" programmers but encourages a dangerous dependency if maintained over a programming career of any length. The following slogan appeared on a Christian Aid collection envelope: "Give a hungry man a fish, and you feed him for a day. Teach him to fish and you feed him for a life". In the context of programming, a distinction might equally be made between the benefits of imparting specific information of limited utility and those which accrue from imparting more general information applicable in a wide range of cases. Where possible, specific information should be derived as an instance of a more general principle; such an approach enables students to gain a more systematic understanding of the programming language. In contrast to the beginners, experienced programmers learning a language use the definition as an independent source of information, deriving programs from it. While experienced programmers may use the compiler to check their understanding of the definition, they also make use of the definition as an independent check on the compiler.

THe work to be described involved setting up an environment in which beginning programmers could be presented with a simple programming language definition and be allowed to test their understanding of it. The environment took the form of a syntax game program. The production rules of the language were first made explicit, the student then being

encouraged to apply them in forming particular statements in the language. As in Miller's system, the fast and accurate computer was retained to check that the student had applied the rules correctly. Moreover, an automated system like Miller's has the advantage that it is trusted by students to perform objectively. The work rests on an adapted form of Miller's thesis, concerning the learning of natural langauges, discussed above: when people learn a programming langauge, they do not need to memorise all the particular programs which comprise it; rather they need to learn rules for producing and interpreting any program in the particular programming language.

In designing the program, it has been useful to draw on the ideas proposed by Jonathan D. Wexler in a report entitled "A Design for Describing (Elementary) Programming Problem Generators in an Automatic Teaching System" (1973). In this report, Wexler outlined a grammar for describing programming problems which he used in a program to teach machine-code programming. The sub-programs in the syntax game program operate in two modes; one in which expressions are generated and presented to the student for checking and one in which the student submits expressions to the appropriate sub-program for checking. In the former mode, ideas from Wexler's system have been developed; while in the latter mode, the work of compiler theorists in automatic syntax analysis has been drawn upon (Gries, 1971).

A generative sytem was chosen because of a desire to get away from the drill-and-practice type of computer-aided instruction reviewed in Chapter 1. Such systems which merely present pre-stored sequences of problems are unnecessarily inflexible in their mode of presentation. Inflexible drilling is harmful because it is not adaptive to the needs of the student and it does not provide the student with a framework in which particular examples can be related to general models. There is no reason why a computer should be used to perpetuate one of the worst possible teaching techniques. If a computer-aided instruction program emulates a programmed learning textbook, then the computer merely becomes an expensive substitute for a book.

4. Prototype System

The prototype system was a program which simply gave students practice in forming logical expressions and checking them. The program can be run interactively from a terminal, and the way in which it functions is described below. The rules for producing logical expressions are presented. These may be reviewed at any time during a session if the student wishes. The program can then either present randomly generated examples of expressions to the student for checking, or the student may input expressions to the program for checking, in which case the program will determine whether or not the input is well formed and reply appropriately. The mode of operation is flexible and chosen by the student, who

may alter it at any point. In both modes the program is capable of error reporting. Where the student's replies are correct this is not strictly necessary, and the program gives the student the option of having this information. Because the program allows the student to enter expressions for checking, it must be capable of doing the checking; it also checks expressions which have been program-generated as this enables errors to be pinpointed in context for the student.

The level of difficulty at which the program presents material is either determined by the sort of productions entered by the student, or in the case of program-generated examples is started arbitrarily low and increased if the student's responses suggest a readiness for more difficult examples. The level of difficulty is proportional to the complexity of the expression. The complexity of the expression is determined by the number of recursive calls of the syntax checking procedure required when checking the example. Syntax checking is accomplished using the method known as recursive descent which has been extended in the program to a funtional form described in Chapter 2.

In the prototype system, example formulae are generated as follows:

(i) Start with a proposition letter.

(ii) Add a negation sign in half the cases.

(iii) Add an operator and letter either on the left or the right in half the cases.

(iv) Put brackets around the whole in half the cases.

(v) Repeat from (ii) until the formulae has the required length (where this is simply a measure of the number of symbols).

The generator starts with a branch which results in the least number of symbols. As all branches except one are recursive, obviously the non-recursive branch must be chosen first. The next branch chosen is the one which will result in the next least number of symbols being added, and so on using the remaining branches in order of their generative power until the formula of the required complexity has been built up.

The generative power (g.p.) is a measure of how many symbols a branch will add to the expression under construction. In the grammar for well-formed formulae (wffs) used by the program, the generative powers of the branches are as follows:

(Branch 1)⟨ppn letter⟩ has g.p. of 1;

(Branch 2)⟨not⟩⟨wff⟩ has g.p. of ⟩=2; and

(Branch 3)⟨wff⟩⟨connective⟩⟨wff⟩ has g.p. ⟩=3.

In this grammar, simple inspection of the grammar is enough to enable the generative power of a branch to be determined.

If the student continues to make mistakes in either checking or production mode, the program enters a remedial mode and begins to explain how it checks an example as well as indicating how it generates examples according to the rules. As mentioned above, the student may request to view the rules at any time.

The student may terminate a session at any point, and the program will give a summary of the session so far. The program then administers a brief interview which allows the student to give an evaluation of the effectiveness of the program.

All input and output which takes place during a session with the program is monitored. The information recorded includes the date of usage, the start and finish times of a session, and the student's name. These files have proved useful in determining how effectively the program functions as a teaching aid.

A short extract from a student's session with the program is recorded below:

(C=program, S=student)

C: THE DATE IS 02-12-78

C: THE TIME IS 17:53:48

C: I AM A LOGIC TUTOR PROGRAM. PLEASE TYPE IN YOUR NAME:

S: GEORGE

C:

C: *WELCOME TO THE FORMULA GAME*

C: ---

C: THIS GAME WILL ALLOW YOU TO PRACTICE PRODUCING AND

C: CHECKING OF FORMULAS. THE OBJECT OF THE GAME IS TO

C: PRODUCE ONLY WELL-FORMED FORMULAS IF I'M CHECKER

C: AND TO CHECK CORRECTLY EACH EXAMPLE I PRESENT IF YOU

C: ARE THE CHECKER.

C: I'LL KEEP SCORE ON HOW WELL YOU GET ON.

C: WOULD YOU LIKE TO KNOW THE RULES OF THE GAME?

C: JUST TYPE Y FOR YES, N FOR NO:

S: Y

C: WELL-FORMED FORMULAS (WFFS FOR SHORT) ARE PRODUCED

C: USING THE FOLLOWING RULE:

C: <WFF>::=<PPN LETTER>|~<WFF>|

C: <WFF><CONNECTIVE><WFF>

C: <PPN LETTER>::=A|B|C|D|E|F|G|H...ETC

C: <CONNECTIVE>::=&|| AND OR

C: N.B. ANY WFF MAY BE ENCLOSED BY BRACKETS, FOR EXAMPLE:

C: A|B IS A WFF AND SO IS (A|B)

C: WOULD YOU LIKE MORE INFORMATION?

S: Y

C: THE RULES TELL YOU HOW A WFF IS PRODUCED:

C: A WFF IS EITHER A PROPOSITION LETTER WHICH IS ANY LETTER

C: OF THE ALPHABET (E.G. A) OR A WFF IS A NEGATION SIGN (~)

C: FOLLOWED BY A WFF (E.G. ~A) OR A WFF IS A WFF FOLLOWED

C: BY A CONNECTIVE (& OR |) AND ANOTHER WFF (E.G. A&~A).

C: ANYTIME WE PRODUCE A WFF, WE CAN PUT BRACKETS ROUND IT,

C: FOR EXAMPLE, (A&~A) IS A WFF.

C:

C: IF AT POINT, YOU'D LIKE TO REVIEW THE RULES,

C: JUST ENTER THE MESSAGE: RULES

C:

C: IF AT ANY POINT, YOU'D LIKE TO QUIT WHILE YOU'RE AHEAD

C: OR OTHERWISE, ENTER THE MESSAGE: QUITS

C: AND I'LL GIVE YOU A RUN DOWN OF THE GAME SO FAR BEFORE

C: QUITTING.

C:

C: EITHER I CAN ACT AS CHECKER OF WFFS OR YOU CAN. YOU CAN

C: CHANGE PLACES BY ENTERING THE MESSAGE: CHANGE

C: DO YOU WANT TO BE CHECKER?

C: JUST TYPE Y FOR YES, N FOR NO:

S: Y

C: AN INTERPRETATION OF THE FORMULA IN ENGLISH IS:

C: DOGS LIKE BANANAS AND PIGLETS HAVE TAILS

C: INTERPRETATION KEY:-

C: ~=NOT &=AND |=OR

C: H=DOGS LIKE BANANAS

C: V=PIGLETS HAVE TAILS

C: IS THE FOLLOWING FORMULA WELL FORMED?

C: H&V ?

C:

S: Y

C: THAT'S CORRECT

C: AN INTERPRETATION OF THE FORMULA IN ENGLISH IS:

C: NOT

C: INTERPRETATION KEY:-

C: ~=NOT &=AND |=OR

C: IS THE FOLLOWING FORMULA WELL FORMED?

C: ~ ?

S: Y

C: THAT'S INCORRECT. DO YOU WISH TO KNOW WHY?

S: Y

C: ~

C:)

C: PPN LETTER REQUIRED AT THIS POINT.

C: AN INTERPRETATION OF THE FORMULA IN ENGLISH IS:

C: PEOPLE ARE MILK AND (NOT COWS HAVE BANANAS AND (PEOPLE

C: HATE FLEAS OR PIGS LOVE PEOPLE)).

C: INTERPRETATION KEY:-

C: ~=NOT &=AND |=OR

C: D=PEOPLE ARE MILK

C: M=COWS HAVE BANANAS

C: V=PEOPLE HATE FLEAS

C: U=PIGS LOVE PEOPLE

C: IS THE FOLLOWING FORMULA WELL FORMED?

C: D&(~M&(V|U))?

C:

S: Y

C: THAT'S CORRECT.

S: CHANGE

C: ENTER A WFF FOR CHECKING:

S: D&----------D

C: THAT'S CORRECT.

C: ENTER A WFF FOR CHECKING:

S: QUITS

C: FINAL STATE OF PLAY

C: I WAS CHECKER FOR

C: 4 WFFS

C: 3 OF THESE WERE CORRECT.

C: YOU WERE CHECKER FOR

C: 18 WFFS

C: 11 OF THESE WERE CORRECT AND YOU CHECKED

C: 9 OF THEM OUT PROPERLY. OF THE

C: 7 DUFF WFFS, YOU LET

C: 2 OF THEM SLIP BY.

C: THAT WASN'T A BAD GAME. BETTER LUCK NEXT TIME.

C: THE TIME IS 18:16:22

C: NOW THAT THE GAME IS OVER, I HOPE YOU WON'T MIND

C: TAKING PART IN A SHORT INTERVIEW. AFTER EACH QUESTION,

C: ENTER AS MANY LINES AS YOU WISH IN REPLY AND WHEN

C: YOU HAVE COMPLETED YOUR REPLY, ENTER: £FINISH

C:

C: DO YOU THINK YOU'VE LEARNED ANYTHING BY PLAYING THIS GAME?

S: YES

S: £FINISH

C: DID YOU ENJOY PLAYING THE GAME?

S: YES THANKS

S: £FINISH

C: ANY COMMENTS, CRITICISMS, OR SUGGESTIONS?

S: MORE INCORRECT EXAMPLES PLEASE

S: £FINISH

The program was nicknamed the "Logictutor" and proved popular among first-year students; many of whom were addicted to interactive computing in any form. An element of novelty was introduced by including an interpretation of the expression in "English" which greatly amused the students. Although Miller found that undergraduates could not learn Polish notation for logical expressions by his purely inductive method, students using the program had little difficulty mastering the notation it gave for logical expressions.

It has been difficult to abstract any recurrent themes from the scripts of student sessions. It was certainly very rare for the program to enter its remedial mode; possibly because the student needed to make three consecutive mistakes to trigger the remedial mode of the program. Students were more likely to make mistakes in the checking mode than in the production mode, although checking was usually the mode in which the students chose to begin playing and so inexperience rather than any inherent difficulty in checking may be the reason for this difference.

The Logictutor was not developed any further because it did not provide enough complexity in either the production set or the language described by the rules to test this method

of teaching. With the Logictutor, it was clear that if students failed to understand the notation of the production rules, they could not approach the tasks set except by trial and error. Their induction was not as crude as that employed by Miller's subjects; they explained their strategy as determining exactly what the rules meant. This emphasis reinforced the view central to this work about the importance of the rules in providing a general model of the language.

5. Generalised System

In order to investigate further this method of teaching an artificial language, it was generalised so that it could be used to teach the syntax of any language which could be specified using production rules. One object of this generalisation was to determine how complex a language could effectively be presented in this way, and another was to experiment with various notations for the production rules themselves. In particular, the generalised program was designed to enable some ideas from Van Wijngaarden's two level W-Grammars to be incorporated into the rules.

As the set of production rules becomes larger, it is more difficult to grasp easily as a whole. Two-level grammars provide a means of generalising the production rules. As explained in Chapter 2, in a two-level grammar there is at the top level a context free system for defining metanotions in the language; these metanotions may be substituted for

hypernotions in the hyperrules which are models of the production rules, thus the rules of the lower level context free grammar describing the language are derived.

Van Wijngaarden's notation for the context free grammar may be used to present the rules to the student. This notation is more compact than BNF and has the advantage of including a rule terminator.

The following extracts from rules giving a two-level definition of SEQUEL illustrate the form of input to the subprogram generator:

```
Metanotions
ALPHA::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.
NOTION::ALPHA;NOTION,ALPHA.
EMPTY::.

General Hyperrules
NOTION list:NOTION;NOTION,comma symbol,NOTION list.
NOTION sequence:NOTION;NOTION,NOTION sequence.
NOTION option:NOTION;EMPTY.
NOTION expression:NOTION term;
        NOTION term,NOTION operator,NOTION expression;
        left par symbol,NOTION expression,right par symbol.

Hyperrules
Statement:Basic Query expression.
Basic Query term:Label option,selection list,
```

where clause option.

Basic Query operator:union symbol;

intersection symbol;

difference symbol.

Label:string,colon symbol.

string:letter sequence.

letter:letter ALPHA symbol.

selection:select from option,table name,

group by option, dupl option.

The generation of a recursive descent syntax checker from the rules turns out to be quite simple as explained in Chapter 2. A function which returns "True" or "False" according to whether or not it recognises a notion is generated for each notion in the language. The general hyper-rules are dealt with by functions of functions which utilise the simple functions and return "True" or "False" as each hypernotion is recognised. By retaining recursive descent as the checking method, the final program can still obtain a measure of the complexity from the depth of recursion and can pinpoint with ease the cause of errors in a production.

The programming of the sub-program generator, to produce the sub-program which presented examples to the student, brought out the need to examine how a context free grammar (CFG) may be used to generate expressions with a specified level of difficulty. In theory, a CFG is specifically a generative grammar. A CFG generates an expression in the language it

defines as follows:

(i) Start with the string (called the "string in hand") consisting only of the distinguished symbol.

(ii) Apply productions from the grammar's set of production rules to the string in hand until it consists only of terminal symbols (i.e. members of the terminal vocabulary).

Such a string is said to be a member or expression in the language generated by the grammar (Martin, 1972).

Depending on the replacement alternative chosen from any particular rule when it is applied, different statements are generated. A systematic method of application is required for generating statements with specific properties. For any given grammar, it may be possible to outline a strategy which enables statements with a desired property to be generated. Wexler (1973) brought up the problem in his report without attempting a solution:

> "There are two important difficulties that arise with problem generators that are not dealt with in this current design. One involves the need to generate problems of a particular level or degree of difficulty. The other difficulty of problem generators is more subtle: how to generate problems that have particular kinds of features or properties."

The next section discusses work which addresses these issues.

5.A A More Detailed SEQUEL Example

The following example of text generated from the W-Grammar for SEQUEL illustrates how explanations are automatically dealt with during parsing and generation of examples by sub-programs. It also illustrates some of the difficulties encountered.

The rule is

```
ATOM_TERM:
    TABLE_NAME_OR_LABEL_OPTION, COL_NAME;
    SET_FN;
    RRB_SYMB, COL_NAME, LRB_SYMB.
```

The procedure for parsing this generated from the above rule by the subprogram generator is as follows:

```
ATOM_TERM : PROC RECURSIVE
      RETURNS(BIT(1));
DCL R BIT(1);
IF EXPLAIN THEN CALL MM('ATOM_TERM?');
IF OPTION(TABLE_NAME_OR_LABEL) THEN
DO; CALL READSYM;
IF COL_NAME THEN
R='1'B;
ELSE R='0'B;
END;
ELSE
IF SET_FN THEN
```

```
R='1'B;
ELSE R='0'B;
ELSE
IF RRB_SYMB THEN
DO; CALL READSYM;
IF COL_NAME THEN
DO; CALL READSYM;
IF LRB_SYMB THEN
R='1'B;
ELSE R='0'B;
END;
ELSE R='0'B;
END;
ELSE DO; R='0'B;
IF EXPLAIN THEN CALL MM('ATOM_TERM NOT FOUND');
END;
LEVEL=LEVEL+1;
RETURN(R);
END ATOM_TERM;
```

As a recognising procedure for each non-terminal element in the grammar is generated, correct error messages are generated for use in EXPLAIN mode. For each recognising procedure, two statements are included at the beginning and end:

```
IF EXPLAIN THEN CALL MM('<element>?');
IF EXPLAIN THEN CALL MM('<element> NOT FOUND');
```

These also occur in the functions for the hyper-rules as the following procedure illustrates:

```
SEQUENCE: PROC(NOTION) RECURSIVE RETURNS(BIT(1));
DCL NOTION ENTRY RETURNS(BIT(1));
DCL (RESULT,FOUND) BIT(1);
     IF EXPLAIN THEN CALL MM('SEQUENCE-');
     FOUND,RESULT=NOTION;
     DO WHILE(RESULT);
     CALL READSYM;
     RESULT=NOTION;
     END;
     LEVEL=LEVEL+1;
RETURN(FOUND);
END SEQUENCE;
```

Executing the recognising procudure in the example given above, a successful parse of an atom term consisting of

PARTS BOLTS

would give the following explanation in EXPLAIN mode:

```
ATOM_TERM?
OPTION-
TABLE_NAME_OR_LABEL?
STRING?
SEQUENCE-
LETTER?
A_SYMB?
A_SYMB NOT FOUND
B_SYMB?
B_SYMB NOT FOUND
```

and so on...

P_SYMB?

LETTER?

A_SYMB?

LETTER?

A_SYMB?

A_SYMB NOT FOUND

and so on...

While this explanation is correct at a low level, it is rather long winded. It does have the advantage of reflecting the action of the parser in checking an example.

The problem of generating a helpful explanation for the subprogram to use when explaining the generation of correct statements is equally difficult. These also can be automatically generated from the grammar by producing the following statements at the beginning and end respectively of the generating procedure for each element in the grammar:

```
IF EXPLAIN THEN CALL MM('ADDING <element> USING <rule>');
IF EXPLAIN THEN CALL MM('<element> ADDED');
```

Thus, from the rule for TABLE_NAME_OR_LABEL, the explanation generated is as follows:

ADDING TABLE_NAME_OR_LABEL USING TABLE_NAME_OR_LABEL:STRING.

ADDING STRING USING STRING:LETTER_SEQUENCE.

and so on.

These explanations are equally long winded, and not particularly illuminating.

In both cases, recognising and generating examples, EXPLAIN mode is automatically activated by the student making repeated errors. It may also be entered at the request of the student to explain a particular example.

The statements incrementing the variable, LEVEL, in the procedures given above illustrate the simple measure of complexity used in the early versions of the software to gauge the depth of recursion and number of procedure calls. The LEVEL variable is local to both the parser subprogram and to the generator subprogram; in both it is initialised to zero and incremented by each subprocedure call within the respective subprograms, thus giving a measure of the number of calls to either parse or generate an example. This measure was improved by calculating an associated generative power for each alternative within a rule.

This general system can improved by importing more appropriate explanatory text into the grammar. Illustrations of this improvement and other improvements are given in the following section.

5.B Illustrations of Difficulties Generating Examples and Solutions Employed

Algorithms for generating examples are summarised in Section 6. This sections illustrates with examples some of the specific difficulties and solutions employed.

An alternative approach which allows for the inclusion of more appropriate explanation is to extend the grammar which drives example generation to include a teaching strategy with associated explanations. Importing explanatory text and the teaching strategy into the grammar allows a finer level of control to be exercised in the generation of examples. The disadvantage of this approach is that the grammar of the language is compromised by that addition of these rules. The subprogram for recognising, ie checking, examples is generated as before from the unalterated syntax.

The grammar below illustrates how lesson on SEQUEL SELECTIONS is generated beginning with a simple example followed by an explanation and finally a complex example.

```
SELECTION_LESSON:SIMPLE_SELECTION_EXAMPLE,
                 SELECT1_EXPLANATION,
                 COMPLEX_SELECTION_EXAMPLE.
SIMPLE_SELECTION:SELECTION.
SELECT1:'THE PREVIOUS EXAMPLE CONSISTED OF A SINGLE SELECTION.',
'IT IS POSSIBLE TO CONSTRUCT A SELECTION WHICH IS A LIST OF',
'SINGLE SELECTIONS AS THE NEXT EXAMPLE WILL ILLUSTRATE.'.
COMPLEX_SELECTION:SELECTION_LIST.
```

An Example of a Simple Strategy.

The strategy is quite simple: progress from an non-recursive alternative, SIMPLE_SELECTION, to the recursive alternative via the explanation given.

The code generated for this lesson is as follows:

```
CALL EXAMPLE(SIMPLE_SELECTION);
CALL EXPLANATION(SELECT1);
CALL EXAMPLE(COMPLEX_SELECTION);
```

A further SEQUEL example is given below to illustrate the problem of generating examples with semantically consistent variable names. In the general system, the subprogram for generating examples is driven by purely syntactic rules. Even in a simple language such as SEQUEL where a production is essentially a single statement, randomly generated strings while correct detract from the comprehensibility of the example. The first of the following examples with randomly generated names is less comprehensible than the second

in which the names refer to components of a database. Both statements have the same correct syntactic form.

SELECT AXYD, SUM(NPEK) FROM JSLT GROUP BY IVOB

SELECT DEPT, SUM(UNITCOST) FROM PARTS GROUP BY DEPT

This problem can be overcome by introducing a consistent set of variable names into the example grammar used for generating examples. This restricting the generality of the grammar so that only semantically meaningful names appear in examples need not be reflected in the rules used to generate the recognising subprogram.

The generation of negative, that is incorrect, examples in the earlier versions of the software was accomplished by wrecking correct examples by randomly removing elements. While not guaranteed to succeed, this proved adequate for simple grammars. It has the disadvantage of producing randomly incorrect examples. A more satifactory solution is to incorporate rules for generating examples which illustrate common mistakes into the example grammar.

5.C Prospects of Practical Application of this Work

The ideas developed in this thesis and their implementation discussed here could form the basis of practical applications of this method for other programming languages, but further refinement of the methods and re-implementation of the software in a portable language is recommended. The current implementation in the PL/1 language is restricted to machines supporting PL/1, typically IBM or IBM compatible mainframes; a more widely available language such as C or Pascal would be better employed in any further development to achieve portability.

Many of the difficulties discussed in Section 5.B above are the result of the software developed being restricted to handling syntactic rules rather than being driven by rules handling both syntactic and semantic aspects of programming languages. The solution employed with SEQUEL of restricting names to consistent database model is not generally applicable with other languages. In procedural languages, the requirement for consistency between declaration of variables and the scope of their usage could be addressed by using W-Grammars predicates.

From an aesthetic standpoint, the formatting used by the software is merely adequate. It would be desirable to introduce additional notation into the grammar for indicating how generated expressions are to be displayed. Such developments could benefit the results of work in information display in the field of Graphics.

6. Algorithms for Generating Examples

In the more general teaching system, a "top-down" approach to generation was attempted. This took two forms which might be characterised as explicitly recursive and explicitly non-recursive; the main distinction was the way in which recursively defined notions were handled. Using these two methods, generators for the SEQUEL language were produced and an evaluation of these generators now follows.

In the SEQUEL generator (version 1), the branches are merely chosen at random. This method of generation has been recommended by Neil Rowe (1978). It is only adequate for simple grammars; in particular, if there are several recursively defined notions in the language, this method cannot be ensured to terminate in a reasonable time. Using this method of generation with the SEQUEL grammar, it was not possible to generate SEQUEL statements. More importantly, it offered no control over the complexity of the statements generated. It must be concluded that if it is desired to have some mechanism whereby statements with specified levels of difficulty are generated, mere random replacement is not adequate.

In a second generator, statements are generated by a random replacement scheme only where the notion is not recursively defined. In this modified form, all explicit recursion during the generation of examples is removed; all recursive notions are dealt with by iterative generation of limited

length. If an easy example is required then all options are omitted and the minimum number of symbols are returned from functions generating any recursively defined notion. This method, while resulting in productions for most grammars, may not terminate in a reasonable time if the grammar has several indirectly recursive notions. It does not allow for the specification of a very exact measure of difficulty of the individual examples either. The generation of hard and easy expressions is adequate for some teaching systems but is rather unsatisfactory for those where the teaching strategy requires a gradual progression from very easy to more difficult examples.

The third generator uses a set of rules to guide generation. These take into account the generative power of each alternative and allow for a finer discrimination to be made between alternative branches.

The knowledge of how examples with the required properties are to be generated is imported into the production rules. The trade-off is that the production set loses generality. Two sets of production rules are required: one which gives a general model of the language and which is presented to the student for reference; and another which embodies a teaching strategy and is used to generate examples for the student to check.

The system outlined above would seem preferable to that of Koffman (1972), who employed a "probabilistic grammar" to

generate logical expressions for use in a computer-aided instruction program. A probabilistic grammar is formal language in which every rewrite rule is assigned a probability of being applied. The teacher must specify separately the method for initialising and updating the probabilities, and there is the overhead of recalculating the probabilities after any change in the student's level of competence.

Allowing the teaching strategy to be expressed entirely in the production rules enables the teacher in effect to program using the grammar only as an author language (Barker and Singh, 1982), and has the advantage that no other specification is necessary.

7. Conclusion

A grammar only comes alive when it is used, so in further work on a more generalised system it must be recognised that the language most effectively being taught is the notation which describes the grammar, for it is that notation which the student must first come to understand. The syntax game programs described here are most effective at testing a student's understanding of the grammar or production rules notation. The ultimate productions are in a sense disembodied and do not have any honest employment in the syntax game program; it is the production rules that are actually given a sense by their use in producing exprssions in the language. Nevertheless, this does not detract from the merit of the programs. They provide an introduction to particular

languages through their syntax, while at the same time giving the student practice at understanding the notation of the formal definition.

The advantages of defining a programming language formally are obvious (Zemanek, 1974). A formal definition of a programming language enables a student to grasp the language as a whole rather than by piecemeal induction. If the notation of the formal definition is not easily understood, these advantages cannot be realised to their full potential. While students should not be encouraged to neglect writing programs when getting to grips with a programming language, a familiarity with the syntax of the language is a helpful preliminary which will cut down the occurrance of syntactical teething troubles and will better equip the student to use the language to its full power. As George W. Cherry noted in the Preface to his textbook on Pascal:

> "I have taken very seriously the careful explication of Pascal's syntax. It's gratuitous frustration for a student to wrestle with a malfunctioning program because his textbook failed to elucidate some syntactical banana peel it's easy to slip on." (Cherry, 1980).

Where the production set is large, decomposing the rules for separate presentation is of value provided the rules are linked together in a wider context of usage, preferably in actually writing programs.

Just as Miller distinguished between people learning to make the machine behave in a certain way and those gaining an understanding of the language, familiarity with the formal

definition of a programming language gives the programmer a means of generally understanding a program as opposed to understanding the particular meaning it may exhibit when it is run. We must clearly differentiate between concrete implementations of languages and their abstract definitions; it is knowledge of the latter which enables programmers to gain an understanding of the meaning of their programs and to rise above the ability to simply make the machine do things. As programming languages move further away from their machine-code origins and become more fully abstract (Geurts and Meertens, 1978), it is imperative for programmers to acquire this understanding so that they may benefit from these conceptual advances.

Ideally the grammar of a programming language should reflect its usage, so that its application becomes transparent in the formation of the problem solution. This implies a grammar of problem-solving. In programming, analysis of the problem is often followed by two separate steps: construction of the solution and translation of the solution into a program. We should be thinking of grammars which will bring these two steps together.

CHAPTER 4 Conclusions

1. The Results of this Work

The fundamental importance of grammar and its role in describing programming languages has been established, and this has been shown to be the basis of a successful method of teaching programming languages.

This work has also demonstrated the adequacy of production systems for specifying not only the grammar of programming languages but also the teaching strategies to be employed in teaching a particular programming language through its formal definition.

2. Applications

2.1. The Design of Structured Editors and Teaching Compilers

In recent years, there has been a trend in microcomputer software for the compiling and editing modes to be linked so that errors detected by syntax analysis can be easily corrected. The UCSD Pascal System (Bowles,1980) has a configurable STUDENT option switch. If this switch is set to true during compilation, the first syntax error will cause the system to enter the editor; the syntax error message will be displayed on the top of the screen and the cursor will be positioned at the point in the program where the error was detected.

The BASIC interpreter incorporated in Sinclair microcomputers (Boldyreff, 1980) has a similar facilty; it will not allow the user to enter in syntactically incorrect BASIC statements. These are signalled by an inverse S on the line being entered as soon as an error is detected. Economy does not allow for any more helpful error messages.

As more and more people untrained in programming are purchasing personal computers and teaching themselves programming, these trends towards self-explanatory error detection are becoming increasingly important. The methods used by the UCSD Pascal System and the Sinclair BASIC interpreter are only a beginning in the right direction. Using the methods outlined in this thesis, it would be possible to construct a system incorporating a full explanation of its working. Such an explanatory mode would not necessarily be of interest to every user of the system and would obviously need to be optional, but it would enable the adventuresome users attempting to teach themselves programming to gain an insight into and a better understanding of the programming language being used.

2.2. Studies in Programming Behaviour

Recent empirical studies (Green, 1980) have shown that criticism of one syntactical form, the nested conditional, was unfounded. Green and his colleagues investigated programmers' understanding of programs written in both un-nested and nested forms, and concluded that programmers found that

there was little to choose between the forms in straightforward application and that where the application was not straightforward, the un-nested form was much more difficult to understand. Green speculates on the 'ideal' construct for nested conditionals and urges readers to try out more real-life studies.

The system I have developed could be easily modified to provide an experimental testbed into the suitability of various syntactic forms, as well as understandability of various programming styles. The experimentor would simply need to specify the syntax of the forms to be investigated; from these a subprogram to generate examples could be produced. The tutor program could be easily modified to administer experiments and monitor and time the subjects responces.

3. Recommendations for Future Work

The above sections on applications give examples of how production rule and grammar based systems have an immediate role in programmer education, and provide the basis for creating an experimental testbed for carrying out investigations into programming behaviour.

Programming languages are the primary vehicle used for programming today; future languages may be directed more towards specifying the solution required rather than describing the step-by-step method for achieving the solution. The evolution of programming languages and their

associated grammars will present new challenges to teachers; however, given the fundamental importance of grammar, the results established here will remain relevant.

The adequacy of production rules for the specification of teaching strategies outside the field of programming languages and their use more generally as a specification language remains for future investigation.

REFERENCES

Alcock, D. (1977). Illustrating BASIC (A Simple Programming Language). Cambridge: Cambridge University Press.

Barker, P.G. and Singh R. (1982). Author Languages for Computer-Based Learning. British Journal of Educational Technology, No. 3, Vol. 13, 167-196.

Barnes, J.G.P. (1980). An Overview of Ada. Software-Practice and Experience, Vol. 10, 851-887.

Boldyreff, C. (1980). An Evaluation of the ZX80. Microprocessor Software Unit Report No. 7, South West Universities Regional Computer Centre, Bath.

Bowles, K. (1980). BEGINNER'S GUIDE TO THE UCSD PASCAL SYSTEM. Peterborough, NH: BYTE BOOKS (Subsidiary of McGraw-Hill).

Chamberlin, D.D. and Boyce, R.F. (1973). SEQUEL: A structured English Language Query Language. Research Report, IBM Research Laboratory, San Jose, California.

Cherry, G.W. (1980). Pascal Programming Structures: an introduction to systematic programming. Reston, Virginia: Reston Publishing Company (A Prentice-Hall Company).

Chomsky, N. (1957). Syntactic Structures. The Hague: Mouton.

Cleaveland, J.C. and Uzgalis, R.C. (1973). What Every Programmer Should Know About Grammar. Research Report,

University of California at Los Angles.

Cleaveland, J.C. and Uzgalis, R.C. (1978). Grammars for Programming Languages. New York: Elsevier North-Holland.

De Reemer, F.L. (1969). Generating Parsers for BNF Grammars. Spring Joint Computer Conference.

Dreyfus, H.L. (1972). What Computers Can't Do: A Critique of Artificial Intelligence. New York: Harper & Row.

Forsyth, R. (1977). The Basic Idea. London: Chapman and Hall.

Foster, J.M. (1970). Automatic Syntax Analysis. New York: MacDonald/Elsevier.

Geurts, L. and Meertens, L.G. (1978). Remarks on Abstracto. Algol Bulletin, 42, 56-63. Gries, D. (1971). Compiler Construction for Digital Computers. New York: Wiley.

Green, T.R.G. (1980). Ifs and Thens: Is Nesting just for the Birds? Software-Practice and Experience, Vol. 10, 373-381.

Kemeny, J.G. and Kurtz, T.E. (1967). BASIC Programming. New York: Wiley.

Kernighan B. and Lesk M.E. (1979). LEARN - Computer-Aided Instruction on UNIX. UNIX PROGRAMMER'S MANUAL. Seventh Edition, Volume 2A. Murray Hill, New Jersey: Bell Telephone Laboratories, Inc.

Kernighan B. and Ritchie D. M. (1978). The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall.

Koffman, E.B. (1972). A Generative CAI Tutor for Computer Science Concepts. Proceedings of the AFIPS Spring Joint Computer Conference, 40, 379-389.

Leibniz, G.W.F. (1674). On Method. In P.P. Weiner (ed.) Leibniz Selections (1951). New York: Scribner's.

Lyons, J. (1970). Chomsky. Fontana Modern Masters Series. London: Fontana.

Marcotty, M., Ledgard, H.F., and Bochmann, G.V. (1976). A Sampler of Formal Definitions. ACM Computing Surveys, Vol. 8, No. 2.

Martin, D.F. (1972). Formal languages and their related automata. In A.F. Cardenas (ed.) Computer Science. New York: Wiley-Interscience.

McCarthy J., Abrahams, P.W., Edwards, D.J., Hart, T. and Levin M.I. (1965). LISP 1.5 Programmer's Manual. Cambridge, Massachusetts: The M.I.T. Press.

Miller, G.A. (1970). The Psychology of Communication. London: Pelican Books.

Peck J.E.L. (1974). Two-level Grammars in Action. Proc. IFIP Congress, 317-321.

PL/1 (F) Language Reference Manual (1969). File No. S360-29.

Form C28-8201-2. Third Edition. International Business Machines Corporation.

Radin, G. and Rogoway, H.P. (1967). Highlights of a New Programming Language. In S. Rosen (ed.) Programming Systems and Languages. McGraw-Hill Computer Science Series. New York: McGraw-Hill.

Rifkin, S. (1975). CERN Lecture Notes, Meeting 2, CERN, Geneva.

Roper, J.S. (1973). PL/1 in Easy Stages. London: Paul Elek (Scientific Books).

Rowe, N. (1978). Grammars as Programming Languages. Creative Computing, 4, 80-86.

Scott, D. (1974). Mathematical Semantics. In B. Shaw (ed.) Formal Aspects of Computer Science. Newcastle: University of Newcastle.

Turner, D. (1975). An Implementation of SASL. TR/75/4, University of St. Andrews, Scotland.

Van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., Koster, C.H.A., Sintzoff, M., Lindsey, C.H., Meertens, L.G. and Fisker, R.G. (1976). Revised Report on the Algorithmic Language Algol 68. Berlin: Springer-Verlag.

Wexler, J.D. (1973). A design for describing (elementary) programming problem generators in an automatic teaching sys-

tem. Technical Report No. 66, Dept. of Computer Science, State University of New York, Buffalo, New York.

Wirth, N. (1971). The programming language PASCAL. Acta Informatica, 1, 35-63.

Wirth, N. (1976). Algorithms + Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice-Hall.

Zemanek, H. (1974). Formalization: Past, Present, and Future. In B. Shaw (ed.) Formal Aspects of Computing Science. Newcastle: University of Newcastle.

BIBLIOGRAPHY

Aho, A.V. and Ullman, J.D. (1972, 1973). The Theory of Parsing, Translation, and Compiling. Volumes I and II. Prentice-Hall Series in Automatic Computation. Englewood Cliffs, New Jersey: Prentice-Hall.

Brown, J.S, Burton, R.R. and Zdybel, F. (1973). A Model-Driven Question-Answering System for Mixed-Initiative Computer-Assisted Construction. IEEE Transactions on Systems, Man, and Cybernetics, Vol SMC-3, No. 3, 248-257.

Carbonell, J.R. (1970a). Mixed-Initiative Man-Computer Instructional Dialogues. BBN Report No. 1971, Job No. 11399. Cambridge, Massachusetts: Bolt Beranek and Newman Inc.

Carbonell, J.R. (1970b). AI in CAI: An Artificial-Intelligence Approach to Computer-Assisted Instruction. IEEE Transactions on Man-Machine Systems, Vol. MMS-11, No. 4, 190-202.

Coombs, M.J. and Alty, J.L. (editors) (1981). Computing Skills and the User Interface. London: Academic Press.

Davis, R. and King, J. (1975). An Overview of Production Systems. Stanford Artificial Intelligence Laboratory Memo AIM-271/ Computer Science Department Report No. STAN-CS-75-524.

Dewar, R. and Schwartz, J. (1977). 'Abstracto' Project for an Algorithm Specification Language. Algol Bulletin, No. 42,

64-73.

Fenchel, R.S. (1981). Self-Describing Systems Using Integral Help. Paper from the author at the University of California, Los Angeles.

Green, T.R.G. (1977). The Necessity of Syntax Markers: Two Experiments with Artificial Languages. MRC Memo No. 145, MRC Social and Applied Psychology Unit, Department of Psychology, The University, Sheffield.

Hartley, J.R. (1976). Computer Assisted Learning in the Sciences: some progress and some prospects. Studies in Science Education, 3 (1976), 69-96.

Holt, R.C., Wortman, D.B., Barnard, D.T. and Cordy, J.R. (1977). SP/k: A System for Teaching Computer Programming. Communications of the ACM, Vol. 20, No. 5, 301-309.

Kettle-Williams, J.M. (1975). Computer Aided Learning Program: The Reverse Polish Program. CSP/Cl/1, Departemt of Computer Science, Prtsmouth Polytechnic.

Krueger, M. (1977). Responsive Environments. Proceedings of the National Computer Conference, 1977, 423-433.

Koffman, E.B. and Blount, S.E. (1973). Artificial Intelligence and Automatic Programming in CAI. Proceedings of the Third International Joint Conference on Artificial Intelligence, 86-94.

Kramer B. and Schmidt, H.W. (1977). On the Implementation of van Wijngaarden Grammars. Institut fur Software-Technologie Internal Report 3/77. Gesellschaft fur Mathematik und Datenverarbeitung MBH Bonn.

Kurki-Suonio, R. (1971). Computability and Formal Languages: A Programmer's Introduction to Computability and Formal Languages. Sweden: Studentlitteratur, Auerbach.

Lauer, P.E. (1975). An Automated Programming and Certification Aid for the Systems Programmer. MRM/90. Computing Laboratory, University of Newcastle upon Tyne.

Lauer, P.E. (1976). Abstract Tree Processors with Networks of State Machines as Control: Their use in Programming Language Definition. University of Newcastle upon Tyne Technical Report Series No. 87.

Ledgard, H.F. (1977). Production Systems: A Notation for Defining Syntax and Translation. IEEE Transactions on Software Engineering, Vol. SE-3, No. 2, 105-124.

Martin, J. (1973). Design of Man-Computer Dialogues. Especially Chapter 24: Computer-Assisted Instruction. Englewood Cliffs, New Jersey: Prentice-Hall.

Oettinger A.G. and Marks S. (1969). Run, Computer, Run: The Mythology of Educational Innovation. Cambridge, Massachusetts: Harvard University Press.

O'Shea, T. and Sleeman, D.H. (1973). A Design for an

Adaptive Self-Improving Teaching System. In J. Rose (ed.) Advances in Cybernetics and Systems. London: Gordon and Breach.

Papert, S. (1972). Teaching Children Thinking. Programmed Learning and Educational Technology, Vol. 9, No. 5, 245-255.

Pask, G. and Scott, C.E. (1972). Learning Strategies and Individual Competence. Int. J. Man-Machine Studies (1972) 4, 217-253.

Rumelhart, D.E. (1977). Introduction to Human Information Processing. New York: Wiley.

Self, J.A. (1974). Student Models in Computer-aided Instruction. Int. J. Man-Machine Studies (1974) 6, 261-276.

Sime, M.E. and Green, T.R.G. (1974). Psychology and the Syntax of Programming. MRC Memo No. 52, MCR Social and Applied Psychology Unit, Department of Psychology, The University, Sheffield (Private Circulation).

Sime, M.E., Arblaster, A.T. and Green, T.R.G. (1977). Structuring the Programmer's Task. J. occup. Psychol. 1977, 50, 205-216.

Sussman, G.J. (1975). A Computer Model of Skill Acquisition. London: American Elsevier Publishing Company.

Taylor, E.F. (1968). Automated Tutoring and Its Discontents. American Journal of Physics, Vol. 36, No. 6.

Turski, W.M. (ed.) (1973). Programming Teaching Techniques: Proceedings of the IFIP TC-2 Working Conference on Programming Teaching Techniques, Zakapane, Poland, September 18-22, 1972. Amsterdam: North-Holland Publishing Company.

Weinberg, G.M. (1971). The Psychology of Computer Programming. Computer Science Series. New York: Van Nostrand Reinhold Company.

Wittgenstein, L. (1953). Philosophical Investigations. Oxford: Basil Blackwell & Mott, Ltd.