IBM

IBM United Kingdom
Laboratories Limited

c B J
WORK

Dynamic Syntax: A Concept for the Definition of the Syntax of Programming Languages

K. V. Hanford
C. B. Jones

Technical Report T.R.12.090

consider relation to Earley work

Unrestricted

Technical Report T. R. 12.090

Dynamic Syntax: A Concept for the Definition of the Syntax of Programming Languages

K. V. Hanford
C. B. Jones

Unrestricted

June 1971

215-8032-0

IBM United Kingdom Laboratories Limited
Hursley Park
Winchester Hampshire

CONTENTS

ILLUSTRATIONS

ABSTRACT

It is well known that the syntax of declarative programming languages is not context-free, and that this is due to their ability to declare names which may then occur only in specific contexts. This report explores the idea that declarations modify the context-free grammar of any program in which they appear. The name *dynamic syntax* has been given to this concept. The report presents a functional formulation of dynamic syntax and applies the resulting metalanguage to the description of the syntax of Algol 60.

1. INTRODUCTION

The subject of this report is the formal specification of the syntactically valid programs in a declarative programming language. Conventionally, such a specification takes the form of a context-free grammar. However, the context-free grammar always specifies some superset of the actual set of valid programs. The members of this superset which are truly valid programs are those which satisfy a number of additional syntactic constraints which the context-free grammar is unable to express. Current practice is to express these additional constraints informally as a set of rules which require that certain relationships exist between the declarations of names and their use elsewhere in the program.

The report explores the notion that the declarative statements of a program construct a context-free grammar for the imperative statements of that particular program, and that the required relationships can be represented by the rules for constructing this grammar. The concept has been termed *dynamic syntax* since it implies a dynamic context-free grammar.

Using the language of the lambda-calculus, the report gives a functional realisation of dynamic syntax, allowing the construction of an expression which denotes the set of language strings of a given programming language. In the Appendix the method is applied to the description of a large subset of Algol 60†.

† All subsequent, unqualified references to Algol should be taken as references to Algol 60.

2. CONTEXT-FREE LANGUAGES

A context-free language is defined in the following way:

A *vocabulary* V is a finite set of symbols. A *string* over the vocabulary is a finite sequence of these symbols. The set of all strings over V (where the set includes the empty string) will be denoted by V^*. Arbitrary symbols will be denoted by upper-case Latin letters A, B, ... and strings by lower-case Latin letters a, b, ...

A *context-free grammar* G consists of:

1. A vocabulary V.
2. A non-empty subset of the cross-product set $V \otimes V^*$. Let (A,a) be a pair belonging to this subset. Then (A,a) is called a *production rule* and the binary relation between the symbol A and the string a is denoted by $A \rightarrow a$. A symbol of V which occurs as the left-hand element of some production rule is called a *nonterminal symbol.* A symbol for which there is no such production rule is called a *terminal symbol.*
3. One of the nonterminal symbols, S, called the *sentence symbol.*

Define the binary relation $\dot{\rightarrow}$ between strings by:

$$a \dot{\rightarrow} b = (\exists x, y, z, W)\ (a = x\,W\,y, b = x\,z\,y, W \rightarrow z)$$

If $a \dot{\rightarrow} b$, we say that a *directly produces* b. Thus, this means that b can be obtained from a by the application of some production rule.

Now define the binary relation $\overset{*}{\rightarrow}$ between strings to be the transitive closure of $\dot{\rightarrow}$, i.e.

$$a \overset{*}{\rightarrow} b = (a = b) \vee (a \dot{\rightarrow} b) \vee (\exists c)\ (a \overset{*}{\rightarrow} c \wedge c \dot{\rightarrow} b)$$

If $a \overset{*}{\rightarrow} b$, we say that a *produces* b. This means that b can be obtained from a by a sequence (possibly empty) of applications of production rules.

Then, the *context-free language* L with grammar G is defined to be the set of strings, over the terminal symbols, which are produced by S. That is, if T is the set of terminal symbols, then:

$$L = \left\{ x \in T^* \mid S \overset{*}{\rightarrow} x \right\}$$

As an example of a context-free language, the set of strings of a's and b's of the form:

a . . . . a b . . . . b

where the a and b sequences are of arbitrary non-zero length, is defined by the grammar:

vocabulary:	a, b, A, B, S
production rules:	S → AB
	A → a
	A → Aa
	B → b
	B → Bb
sentence symbol:	S

This grammar has terminal symbols a, b and nonterminal symbols A, B, S.

Some further terminology: A *terminal string* is a string over the terminal symbols. A *phrase* is a terminal string produced by a nonterminal symbol. Suppose A is a nonterminal symbol. Then a phrase produced by A is called an *A-phrase* and the set of all phrases produced by A is called the *A-phrase class.*

We now introduce a variant of the above notation for context-free grammars which will be more convenient for development in the sequel. A nonterminal symbol is written as a descriptive multi-character identifier, possibly hyphenated, e.g. simple-arithmetic-expression. A terminal symbol is either a single character, e.g. q, or an identifier in bold type, e.g. **then**. We generalise the idea of a terminal symbol and use it to denote a string which we choose to leave undefined. This allows us to shorten grammars by using symbols like **letter**, **letter-or-digit**. The operation of concatenation is indicated explicitly by the infix operator $\hat{}$. The right-hand side of a production rule can contain any number of direct productions for the left-hand nonterminal, separated by the operator | ('or').

As an example, the definition of a simple arithmetic expression in Algol may be written:

simple-arithmetic-expression → term | **adding-operator** $\hat{}$ term |
simple-arithmetic-expression $\hat{}$ **adding-operator** $\hat{}$ term

We shall refer to such a multi-production rule simply as a production rule (or rule).

The affinity with the BNF[1] *metalinguistic formula:*

<simple arithmetic expression> : : = <term> | <adding operator> <term> |
<simple arithmetic expression> <adding operator> <term>

is evident.

We postulate the existence of strings **emptystring** and **nullstring** which satisfy:

emptystring ^ x = x ^ **emptystring** = x
nullstring ^ x = x ^ **nullstring** = **nullstring**

for all strings x.

THE SYNTAX OF PROGRAMMING LANGUAGES

This and the subsequent section suggest that a method for the complete description of the syntax of declarative programming languages can be based on a consideration of the syntactic role of declarations.

Section 1 of the Algol 60 report[2] says "...statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements...". The use of declarations characterises many present-day programming languages.

An Algol declaration associates a set of properties with a computational object. These properties can be represented by three items:

1. A set of values which can be taken by the object.
2. A specification of the contexts in which the name of the object may occur in statements.
3. A specification of the meaning (effect) of the object in the contexts specified under 2.

This bundle of information constitutes the total interface between the object and the statements which may use it. Item 2 is the syntactic part of this interface and represents the syntactic character given to the object by the declaration.

Consider a type declaration for example. The type given to a variable defines the class of numbers which it may take as values. But the type also determines the kind of expression in which the variable may occur as a primary. The Algol report (Section 3.3) says, for example ". . . . the constituents of simple arithmetic expressions must be of type **real** or **integer** . . . . ".

Because it permits declarations, Algol 60 is not a context-free language. Floyd's[3] proof of this concentrates attention on a program of the form:

begin real x^n; x^n : = x^n **end**

where x^n stands for the identifier x x . . . . x consisting of n x's. This is a valid program only if all the x^n sequences have equal length. The plausibility of Floyd's result can be seen intuitively as follows. A derivation of a program of this form from a context-free grammar can be considered to involve the intermediate string:

begin ^ **real** ^ identifier ^ ; ^ identifier ^ : = ^ identifier ^ **end**

At this point we need to specify that all three occurrences of identifier must produce identical terminal strings. This identity constraint cannot be expressed by a context-free grammar.

Other Algol programs illustrate further types of constraint. To derive a valid program from:

begin ^ **real** ^ identifier ^ , ^ identifier ^ ; ^ **end**

we need to specify that the two occurrences of identifier should produce non-identical terminal strings. We cannot express this non-identify constraint with a context-free grammar.

Again, to derive a valid program from:

begin **array** X [bound-pair-list] ; X [subscript-list] := X [subscript-list] **end**

we must specify that bound-pair-list and subscript-list produce lists of the same length. This is essentially an identity constraint and we are unable to express it with a context-free grammar.

Abstract forms of the above three kinds of constraint are respectively exhibited by the following sets of strings over { a, b, c, • } :

$\{a^n \bullet a^n \bullet a^n \mid n \geq 0\}$
$\{a^m \bullet a^n \bullet a^p \mid m,n,p \geq 0 \text{ and pairwise unequal}\}$
$\{a^n \bullet b^n \bullet c^n \mid n \geq 0\}$

All three sets are non-context-free languages.

4. THE CONCEPT OF DYNAMIC SYNTAX

This section puts forward a certain view of the syntactic constraints imposed by declarations.

With a slight re-arrangement to help bring out the point we wish to make, the Algol grammar for expressions contains a rule of the form:

primary → simple-variable | ....

and the grammar for declarations contains the rules:

type-declaration → type ^ type-list
type-list → simple-variable | simple-variable ^ , ^ type-list

The grammar defines simple variables by the rules:

simple-variable → variable-identifier
variable-identifier → identifier

Now the last two rules merely say that a simple variable is an identifier; the grammar could therefore be simplified by throwing away these rules and replacing simple-variable by identifier throughout the grammar. Why were the above five rules formulated in this redundant way? (In a document as carefully composed as the Algol report we can be confident that this was by design.) The reason has been expressed† as follows: "they (the rules) are trying to say something which the notation cannot convey, namely, that a simple variable is an identifier which has occurred in a type declaration". Out of the infinite set of simple variables defined by the context-free grammar, only those which have been declared may be used as primaries.

Consider the following alternative scheme for saying that a simple variable is an identifier which has occurred in a type declaration:

1. Leave unchanged the grammar for expressions, including the rule:

   primary → simple-variable | ....

2. In the grammar for declarations, replace simple-variable by identifier to obtain:

   type-declaration → type ^ type-list
   type-list → identifier | identifier ^ , ^ type-list

† Higman[4], page 48.

3a. Erase the rules:

simple-variable → variable-identifier
variable-identifier → identifier

b. *Specify (in some convenient notation) that the production of an identifier, say a, in a type declaration causes the rule:*

simple-variable → *a*

to be added to the grammar.

3a opens up a gap in the grammar, since **simple-variable** is used in the definition of expressions but is nowhere defined. 3b shows how this gap is bridged *during the process of producing a program.*

Initially, there is no production rule for **simple-variable.** Each time an identifier is declared in a type declaration, a rule for **simple-variable** is added to the grammar. Thus, the specification of the identifiers V1, V2, ... , Vm in type declarations leads to the rule:

simple-variable → V1 | V2 | . . . . . | Vm†

Now, declarations do not always simply introduce the identifiers which can occur in statements; the declaration of a complex object may determine the form of certain grammatical units which can be associated with the name of the object. Thus, the declared dimension of an array determines the number of subscripts in a reference to an element of the array; and the declaration of a procedure specifies the number and form of the actual parameters which can occur in a call of the procedure.

These complications do not, however, prevent us from treating such declarations by the method outlined above. For example, the specification of the identifiers A1, A2, ... as arrays of dimension 2, 1, ... respectively leads to new rules:

array-identifier → A1 | A2 | . . . .
subscripted-variable → A1 [subscript-expression , subscript-expression] |
A2 [subscript-expression] |
. . . . . .

† To simplify the above discussion we found it convenient to ignore the distinction made in Algol between arithmetic simple variables and boolean simple variables.

Similarly, the declaration of procedures P1, P2, ... adds rules of the form:

```
procedure-identifier → P1 | P2 | ....
function-designator → P1 ^(^ array-identifier ^,^ string ^)|
                      P2 ^(^ boolean-expression ^,^ procedure-identifier ^)|
                      ......
```

where the number and form of the parenthesised actual parameter specifications depend on the procedure declaration.

We are now in a position to make the following observation: at any point in an Algol program, the arithmetic expressions which may occur form a context-free language. That is, if we take any arithmetic expression of any Algol program and ask what is the nature of the set of all expressions which could validly be written at this point, we find that this set constitutes a context-free language. For, if we ignore declarations, then the valid expressions are all those expressions which can be constructed solely from constants, and this set constitutes a context-free language. Now take into account the declarations. We saw above that these can be considered to add certain context-free production rules to the grammar. The result follows.

The above discussion relates only to arithmetic expressions and to a program consisting of a single block. Also, our observation (on the context-free nature of the set of valid expressions at any given program point) concerns the *static* nature of an Algol program in which the declarations are taken to be *fixed*. We shall show, however, that the approach we have taken here provides a basis for describing the full syntax of Algol. That is, we shall show that it is possible to devise a program-writing scheme such that, at any point in the process where a terminal string is written down, the choice of available strings forms a context-free language. In this scheme the writing down of a string may be associated with the appearance of new context-free production rules.

The principal formulation problems to be solved are:

1. How do production rules appear as a result of declarations and how are these new production rules made available?
2. How do production rules disappear as a result of the re-declaration of an identifier (the problem of *scope*)?
3. How can a reference to a declared object appear in a statement which precedes the declaration of that object (the problem of *use before declaration*)?
4. How can the solutions to the above problems be expressed in a formal metalanguage?

We have given the name *dynamic syntax* to the concept that a declaration has a metasyntactic effect which can be represented by the dynamic creation of context-free production rules. The remainder of the report is mainly devoted to providing a workable notation for dynamic syntax. The next section deals with a number of preliminaries.

5. SOME PRELIMINARIES

This section reviews some mathematical techniques used in the metalanguage developed in the next section.

5.1 LAMBDA-NOTATION

We shall use the lambda-notation of Church[5], as modified by Landin[6]. The notation is summarised here. For a fuller account see Landin's paper.

The lambda-notation pins down the notion of a function as an object in a universe of discourse. It allows functions to be manipulated with the same freedom as more conventional objects and in particular to be used as arguments and values of other functions.

The function ~~(for example)~~ which squares its argument is denoted by $\lambda\ x\ .\ x^2$, so that the definition:

$$\text{square}\ (x) = x^2$$

can also be written:

$$\text{square} = \lambda\ x\ .\ x^2$$

The lambda-expression $\lambda\ x\ .\ x^2$ is the name of a function. In normal usage, square (x) is used indiscriminately to name both a function and the result of applying that function to an argument x. In the lambda-notation, $\lambda\ x\ .\ x^2$ stands unambiguously for the function in itself, i.e., for an object which is a mapping.

The lambda-notation makes a clear distinction between *functional abstraction*, the creation of a function object $\lambda\ x\ .\ E$ and *functional application*, the process of evaluating a function by applying it to an argument. Functional application is denoted by a *combination* M N. The value of the *operator* M must be a function and the value of the *operand* N must be a valid argument of that function. The argument may itself be a function, as may the value of M N.

The expressions considered here have been of three types: identifiers, lambda-expressions and combinations. These are collectively termed *applicative expressions*. A detailed discussion of the evaluation process for combinations can be found in Landin[6].

A description of a function must be structured to give an expression for the value of the function and to indicate the use of the argument in this expression. Let E be an expression involving x†. Then the lambda-expression:

$\lambda\, x\, .\, E$

denotes the function which has the value E when the argument has the value x. The lambda-expression has *bound variable* x and *body* E.

We are now able to paraphrase an expression of the form:

. . . . square (a) . . . .

by:

. . . . $(\lambda\, x\, .\, x^2)$ (a) . . . .

where the name of a function has been replaced by an expression designating the function.

The idea of a function as an object leads to the possibility of using a variable to stand for a function and of passing a function as an argument. In the expression:

$(\lambda\, n\, .$ square (n)) (10)

with value 100, a function is applied to a number. Now consider:

$(\lambda\, f\, .\, f\, (10))$ (square)

which again has the value 100. Here the bound variable occurs as a function in the body of the lambda-expression, so that a function is required as *argument*. The latter example is equivalent to:

$(\lambda\, f\, .\, f\, (10))\ (\lambda\, x\, .\, x^2)$

A function may not only be passed as a function argument but may also be returned as a function *value* as with:

$\lambda\, m\, .\, \lambda\, x\, .\, m\, x + c$

When applied to a number, this function returns a function which multiplies its argument by this number and adds c.

† Strictly speaking, E is to be a form in which x is the only free variable.

A function which returns a function is called *function-producing*. There follow two examples of function-producing functions, both of which require a function as argument. The function double, defined by:

double = λ f . λ x . f(x) + f(x)

or alternatively:

double(f) = λ x . f(x) + f(x)

accepts a function f and returns a function whose value is double the value of f. The function twice, defined by:

twice = λ f . λ x . f(f(x))

or:

twice(f) = λ x . f (f(x))

accepts a function f and returns a function which applies f twice over to the argument.

5.2 A REVISED NOTATION

This section introduces two devices to simplify the writing of applicative expressions.

5.2.1 Omitting Parentheses and Commas

The use of function-producing functions involves operators which are themselves combinations, as in the formulae:

(double (square)) (4) = $4^2 + 4^2$
(twice (square)) (4) = $(4^2)^2$

We can avoid parentheses by adopting the rule that juxtaposition stands for functional application and associates to the left. This allows us to write, for example:

square 4	for	square (4)
double square 4	for	(double (square)) (4)
twice square 4	for	(twice (square)) (4)

This technique can be extended to allow the dropping of parentheses and commas in the argument lists of multi-argument functions. If x,y,. . . . .,z are variables and E is an expression involving x,y,. . . . .,z, then by a straightforward extension of functional abstraction to allow a *list* of bound variables, we can write:

λ x y . . . . . z. E

to denote the function of x,y,. . . . . ,z that E is. We prefer, however, to represent a function of several arguments by repeated use of simple functional-abstraction involving a single argument.

To illustrate the idea, the combination:

$(\lambda\ x\ y\ .\ x^2 + y^2)\ (3,4)$

can be represented as:

$(\lambda\ x\ .\ \lambda\ y\ .\ x^2 + y^2)\ (3)\ (4)$

or:

$(\lambda\ x\ .\ \lambda\ y\ .\ x^2 + y^2)\ 3\ 4$

In the definition of a function we may choose to write the bound variables on the left-hand side following the function name. Again we can dispense with parentheses and commas. Thus, the following definitions of the function sumsquare are all equivalent:

sumsquare x y = $x^2 + y^2$
sumsquare = $\lambda\ x\ y\ .\ x^2 + y^2$
sumsquare = $\lambda\ x\ .\ \lambda\ y\ .\ x^2 + y^2$
sumsquare x = $\lambda\ y\ .\ x^2 + y^2$

An application of the sumsquare function, ~~conventionally written, for instance~~:

sumsquare (3, 4)

is now written:

sumsquare 3 4

Parentheses now have no meaning other than as grouping signs which are used to vary the standard evaluation sequence. Examples of this use are the formulae:

double (twice square) 4 = $2\ (4^2)^2$ $+ (4^2)^2$ $(4^2+4^2)^2 + (4^2+4^2)^2$
twice (double square) 4 = $2\ (2\ .\ 4^2)^2$

5.2.2 The "where" Notation

The expression:

$(\lambda\ x\ .\ E)\ a$

will be denoted by:

E
where x = a

When the argument is a function, bound variables will be written on the left-hand side. For example:

$(\lambda\ f\ .\ E)\ (\lambda\ x\ .\ G)$

will be written:

E
where f x = G

The **where** part is an *auxiliary definition.*

The **where** notation may be extended to handle a function of more than one argument (i.e. iterated application). For example, the expression:

$(\lambda\ x\ .\ \lambda\ y\ .\ x/y)\ a^2\ -\ b^2\ a^2\ +\ b^2$

with value $a^2\ -\ b^2\ /a^2\ +\ b^2$ is paraphrased by:

x/y
where $x = a^2\ -\ b^2$
and $y = a^2\ +\ b^2$

An indented layout is used to indicate the scope of auxiliary definitions. For example, the above applicative expression is equivalent to:

$(\lambda\ x\ .\ \lambda\ y\ .\ x/y)\ ((\lambda\ f\ .\ f\ a\ b)\ (\lambda\ x\ .\ \lambda\ y\ .\ x^2\ -\ y^2\ ))$
$((\lambda\ g\ .\ g\ a\ b)\ (\lambda\ x\ .\ \lambda\ y\ .\ x^2\ +\ y^2\ ))$

or:

x/y
where x = f a b
where f x y = $x^2\ -\ y^2$
and y = g a b
where g x y = $x^2\ +\ y^2$

5.3 RECURSIVE DEFINITIONS

A *recursive definition* of an object x is a definition of the form:

x = ... x . . . . .

where the x in the definiens is a reference to the object being defined rather than a reference to some other object existing in the environment of the definition. We shall indicate the recursive nature of a definition by writing:

rec x = ... x . . . . .

An example of an expression involving the auxiliary definition of a recursive function is:

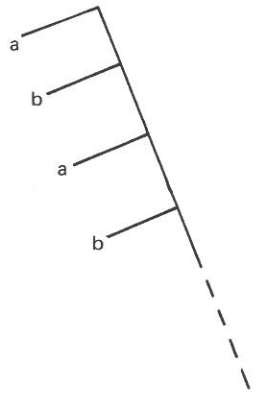
fact 5 + fact 7
where rec fact n = (**if** n = 0 **then** 1 **else** n x fact (n – 1))†

In the expression:

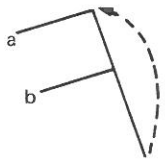
head L
where rec L = (a, (b, L))

L is the infinite list structure (a, (b, (a, (b, ...))))

or



or

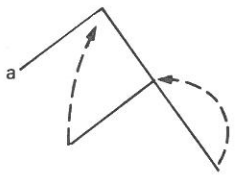


The use of **rec** can be extended to a group of *simultaneous recursive definitions.* For example:

(L, L)
where rec L = (a, M)
and M = (L, M)

† The conditional expression can be formalised in terms of lambda-notation.

Here, L is the infinite list-structure:



The formulation of **rec** in terms of a *fixed-point operator* is described in Landin[6].

5.4 STRUCTURE DEFINITIONS

We describe here the use of *structure definitions* or *abstract syntax* for specifying how a new class of objects is constructed from existing classes of objects. The specification ignores written representations and talks only in terms of a set of functions which operate on the objects being defined.

The following structure definition defines the class of complex numbers:

a complex-number has a real-part which is a real-number
and an imaginary-part which is a real-number

A complex number is thus an object which can deliver a real part and an imaginary part both of which are real numbers (assumed already defined). The new class is described by specifying two new functions real-part and imaginary-part whose domains are the new class and whose ranges are existing classes. These functions are termed *selectors*. Corresponding to the new class is a predicate function which tests for membership of this class. Thus the structure definition for the class of complex numbers introduces two selector functions and a predicate function.

A structure definition is accompanied by a *constructor* function which produces members of the new class. A constructor which creates a complex number from two real numbers is defined by the axioms:

real-part (construct-complex x y) = x
imaginary-part (construct-complex x y) = y
construct-complex (real-part w) (imaginary-part w) = w

The structure description for complex numbers is an example of a *format* specification. A class may however contain objects of a number of different formats. To describe such a class we specify predicate functions which test the format of a member of the class. As an example, we give below a structure description for the class of applicative-expressions, which has three formats. This also illustrates the fact that a structure definition may be recursive. The definition is prefixed by **rec** to indicate that where the name applicative-expression occurs in the body of the definition this refers to the class being defined.

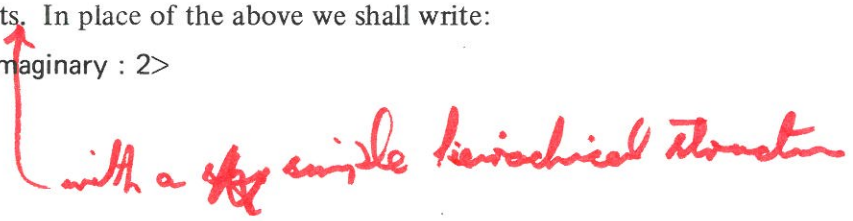
```
rec an applicative-expression is
    either an identifier
        or a lambda-expression and has a bound-variable which is an identifier
                                  and a body which is an applicative-expression
        or a combination and has an operator which is an applicative-expression
                          and an operand which is an applicative-expression
```

A similar technique to structure definitions is available for the description of objects rather than classes of objects. The complex number 1 + 2i may be defined by:

```
has a real which is 1
and an imaginary which is 2
```

This defines a single object and constitutes an *ad hoc* definition of the functions real and imaginary when applied to this particular object. This gives us a standard form of syntax for describing objects. In place of the above we shall write:

```
<real : 1, imaginary : 2>
```



6. DYNAMIC PRODUCTION SYSTEMS

Section 2 introduced context-free production systems for the description of syntax and Section 3 showed that these are inadequate for declarative programming languages. Section 4 put forward a concept of dynamic syntax as a basis for the complete description of the syntax of such languages. This section shows how this concept can be implemented in terms of dynamic production systems which are a generalisation of context-free production systems. In our description of dynamic production systems we shall make use of the techniques reviewed in Section 5 for specifying functions and constructed objects.

Section 6.1 will show how a context-free grammar may be written in lambda-notation. Section 6.2 will describe a metalanguage based on dynamic syntax and Section 6.3 will apply this metalanguage to a subset of Algol 60. Section 6.4 will discuss a slightly different interpretation of BNF from the normal one.

6.1 A VARIANT OF CONTEXT-FREE GRAMMARS

The definition of a context-free grammar may be sharpened by being written as an applicative expression. In particular, the interdependencies between the definitions of certain phrase classes may be clearly displayed. Figure 1 is a definition of expressions in Algol. This clearly indicates that the definitions of arithmetic-expression, simple-arithmetic-expression, boolean-expression, relation and variable form a group of simultaneous recursive definitions. It also shows, for example, that the definition of primary is auxiliary to the definition of simple-arithmetic-expression and is needed nowhere else in the grammar.

6.2 A REALISATION OF DYNAMIC SYNTAX

This section presents the basic ideas for a metalanguage which implements the concept of dynamic syntax. Our realisation of dynamic syntax makes use of the notion of syntactic functions to solve the problems posed at the end of Section 4. In the following discussion, each topic is introduced through an example drawn from Algol.

```
rec arithmetic-expression → simple-arithmetic-expression |
                            if ^boolean-expression^ then ^simple-arithmetic-expression^ else ^arithmetic-expression

and simple-arithmetic-expression → primary |
                                   adding-operator ^primary |
                                   simple-arithmetic-expression ^ arithmetic-operator ^primary

                                   where primary → unsigned-number |
                                                   variable |
                                                   (^arithmetic-expression^)

and boolean-expression → simple-boolean-expression |
                         if ^boolean-expression^ then simple-boolean-expression ^ else ^ boolean-expression

                         where simple-boolean-expression → boolean-secondary |
                                                           simple-boolean-expression ^ boolean-binary-operator ^ boolean-secondary

                                                           where boolean-secondary → boolean-primary |
                                                                                     ¬ | ^boolean-primary

                                                                                     where boolean-primary → true |
                                                                                                             false |
                                                                                                             variable |
                                                                                                             relation |
                                                                                                             (^boolean-expression^)

and relation → simple-arithmetic-expression ^ relational-operator ^ simple-arithmetic-expression

and variable → simple-variable |
               subscripted-variable

               where simple-variable → identifier
                 and subscripted-variable → identifier ^ [ ^ subscript-list ^ ]

                                            where rec subscript-list → arithmetic-expression |
                                                                       subscript-list ^ , ^ arithmetic-expression
```

Figure 1. A Definition of the Context-Free Syntax of Algol 60 Expressions

6.2.1 Rule Creation : A Rule as Part of a Produced Object

A simple form of a **real** type declaration in Algol can be defined by the following production rule:

real-type-declaration → <text : **real** ^ x,
rule : real-simple-variable → x>
where x → identifier

This says that a real type declaration is a constructed object with two components. One component (with selector **text**) is a string of the form:

real ^ x

and the other component (with selector **rule**) is a production rule of the form:

real-simple-variable → x

x stands for an identifier and both occurrences of x are to be replaced by the *same* identifier. The constructed object has the following interpretation: the first component is a program phrase, the second component is a production rule representing the metasyntactic effect which that phrase has on any program in which it appears.

Declarations are therefore obtained through functions which return constructed objects as their values. These constructed objects have two components, one a piece of program text and the other a set of new rules. In this way, the production of a declaration is associated with the creation of new production rules.

There is a serious objection to the above formulation. In the evaluation of the combination:

(λ x . < text : **real** ^ x, rule : real-simple-variable → x>) identifier

we insist that the operand **identifier** be fully evaluated to a terminal string before the operator is applied to it. An evaluation process in which the operand of a combination is always evaluated before application of the operator has been termed *normal evaluation* by Landin[6]. The lambda-calculus however lays down no particular evaluation process since all processes give the same result. The difficulty arises because in our application of the lambda-calculus to generative grammars we use as a primitive the operator | ('or') which is non-deterministic. We shall return to this point in Section 6.4, where we shall see that the difficulty disappears if we interpret a grammar as a definition of a set of strings rather than as a recipe for getting some string. Meanwhile, normal evaluation of operator/operand combinations is assumed.

6.2.2 Use Before Declaration: Representing Partially Produced Text by a Function

Consider the formulation of a labelled statement in Algol (for simplicity assume that it can declare only a single label). A context-free representation is:

labelled-statement → label^ : ^unlabelled-statement

Now, the label implicitly declared here may be referenced in any statement of the same block, including those statements which precede this particular labelled statement (it may even be referenced in the labelled statement itself).

Thus, the set of grammatically correct unlabelled statements of the block depends on all those identifiers chosen as labels in the block. We deal with this problem by formulating the definition of a labelled statement as follows:

labelled-statement → <text-function : λ y . x^ : ^unlabelled-statement y,
rule : label → x>

This says that a labelled statement is a constructed object with two components. One component is a function:

λ y . x^ : ^unlabelled-statement y

whose body contains an identifier x.

The other component is a rule of the form:

label → x

The two occurrences of x stand for the same identifier.

The effect of this scheme is that the (initial) production of a labelled statement is incomplete: merely the label is produced. This enables the metasyntactic effect of this labelled statement to be derived, in the form of a new rule. The incomplete production becomes the body of a function. A subsequent application of this function to some argument will produce a program phrase consisting of a statement with the identifier as its label.

Thus, use before declaration is dealt with by means of function-producing functions (more precisely, functions which return constructed objects with function components). This effectively allows the partial production of a program phrase through a function whose second component is, as before, a set of new production rules but whose first component is not a phrase, but a function which can be subsequently applied to obtain a phrase. The body of this function component contains the declarative sub-phrases which have created the production rules of the second component.

To summarise, so far we have seen how new production rules appear as a result of declarations and how functions may be created containing partially produced phrases. We look next at how new production rules subsequently become available for use in the production process and how produced functions subsequently get applied. These topics are discussed in the context of the Algol structure.

6.2.3 Using Created Rules: Rules as Arguments of Other Rules

An Algol block B has the form:

begin D ; D ; . . . . ; D ; S ; S ; . . . . . ; S **end**

where D stands for an explicit declaration (i.e. any declaration except an implicit label declaration) and S stands for a statement.

Declarations made outside B have scope which includes B, except where their identifiers are redeclared. Declarations made inside B have scope B. This leads us to give the following interpretation to a block:

Let an *environment* be some complete representation of the metasyntactic effect of a set of declarations. Then a block is a function of an environment. The argument passed to a block is a *global* environment representing the declarations made outside the block.
The function block is defined in terms of an object block* which has two components. The first component is a function (of an environment argument) whose body contains all of the declarative phrases of the value of block. The second component is a *local* environment representing the metasyntactic effect of these declarations. block now forms a *total* updated environment from the global environment passed to it as argument and the local environment created by block*. block then applies the function component of block* to the total environment. The result is a phrase which is an Algol block.

Thus, effectively, block* is the first pass in a two-pass production process.

This has assumed the existence of a class of objects, called environments, which completely characterise the metasyntactic effect of the declarations of a block-structured program. We shall implement an environment as a set of generated production rules together with a mechanism for delimiting the scope of these rules. This scope mechanism is based on the concept of a dynamic set of symbols. When a declaration uses an identifier to name a new object, a new symbol is created. We call a symbol created by a declaration a *generated symbol* and say that the identifier *belongs* to the generated symbol. Then a generated symbol designates

a particular declaration of the identifier which belongs to it. At any point in the program an identifier may belong to at most one generated symbol: if the identifier is re-declared, the generated symbol to which it belongs in the enclosing block does not exist in the enclosed block. The scope of the declaration (and therefore of its metasyntactic effect) is just that part of the program in which the corresponding generated symbol exists.

Let us represent a generated symbol by an (identifier, token) pair, where a *token* is some mark unique to a particular generated symbol. For each block B there is a local set of identifiers and a local set of generated symbols. Both are initially empty. When a new local identifier is declared, any identifier not already in the local set of identifiers may be chosen. An (identifier, token) pair is created and added to the set of generated symbols. An inexhaustible source of different tokens is assumed.

The block B also has a global set of generated symbols, with a member for each declared identifier whose scope includes the block immediately surrounding B. The local and global sets of generated symbols for B are summed to form a set whose members designate those declared identifiers whose scope includes B. Any global generated symbol whose identifier occurs also in a local generated symbol is excluded from the sum set. The members of this sum set are the generated symbols *for* B.

Generated symbols control the scope of generated production rules in the following way. In a generated rule a declared identifier i is represented, not by itself, but by a function whose argument is a set s of generated symbols. The function tests if i is a member of s and, if it is, returns i; in the case that i is not a member of s, the function returns the nullstring, thus creating a blind alley in the production process. In this way the use of a generated production rule is blocked off in any block outside its scope. The method requires that all generated production rules be functions of a set of generated symbols.

As an example, the Algol declaration:

real ABC

creates:

1. A generated symbol (ABC, 23), say, (assuming tokens are obtained by enumerating the integers).
2. The generated production rule:

real-simple-variable s → **if** (ABC, 23) ϵ s **then** ABC **else nullstring.**

We can now restate our view of a block as follows. To improve readability, the application of selector functions is distinguished from the application of other functions by using the application operator **of** in the case of selectors.

```
block global-rules global-symbols →
        partial-text all-rules active-symbols
        where partial-text → text-functionofblock*
        and all-rules → union global-rules local-rules
                        where local-rules → rulesofblock*
        and active-symbols → update-symbols global-symbols local-symbols
                             where local-symbols → symbolsofblock*
        where block* → etc.
```

In words, the function block of two arguments global-rules and global-symbols is defined as follows:

1. An object block* with three components is produced.
2. The set local-symbols is obtained by applying the selector symbols to block*.
3. The set active-symbols is obtained by applying the function update-symbols to the set global-symbols and the set local-symbols.
4. The set local-rules is obtained by applying the selector rules to block*.
5. The set all-rules is obtained by applying the function union to the set global-rules and the set local-rules.
6. The function partial-text is obtained by applying the selector text-function to block*.
7. The block is obtained by applying the function partial-text to the set all-rules and the set active-symbols.

To summarise:

a. Part of the value of a declaration is a set of new production rules which represent its metasyntactic effect.
b. The ability to use an identifier before declaring it can be modelled by a multi-pass production scheme in which one pass produces a function and an argument which are combined in the next pass.
c. Scope in a block-structure language can be formulated in terms of generated symbols and production rules with arguments: a block is a function of a set of generated production rules and a set of generated symbols; a generated production rule is a function of a set of generated symbols.

6.3 THE DEFINITION OF A SUBSET OF ALGOL 60

A definition of the complete syntax of a subset of ECMA Algol[7] is given in the Appendix. To shorten the definition, the following cuts have been made in the language: procedure declarations and the built-in procedures are omitted; a bound pair expression is restricted to be a number; the operator ÷ may take operands of type **real** as well as type **integer**; a program must be a block.

We first review the auxiliary definitions which support the syntax description.

The concatenation of two functions is defined by:

function-concatenation f g = λ x . f x ^ g x

function-concatenation operates on two string functions f and g to produce a function whose value for a given argument is the concatenation of the values of f and g for the same argument.† Because of the importance of this function, we introduce an infix operator to denote it:

f ^^ g = function-concatenation f g

We often wish to use a function corresponding to some set of delimiters as an operand of the operator ^^. A n-ary function which always returns the delimiter **then** (for example) is given by k_n **then**, where k_n is the function-producing function defined by:

$k_n\ x = \lambda\ y_1\ .\ \lambda\ y_2\ .\ .\ .\ .\ \lambda\ y_n\ .\ x$

We shall omit the k_n. Thus (for example) the expression:

if ^^ boolean-expression ^^ **then**

where boolean-expression is a binary function, is to be interpreted as:

k_2 **if** ^^ boolean-expression ^^ k_2 **then**

i.e. as the function:

λ x . λ y . **if** ^ boolean-expression x y ^ **then**

Throughout the definition, a set is represented by a list formed from its elements. A structure definition schema is given for lists. By substituting rule and symbol for the parameter, definitions are obtained for rule-list and symbol-list.

head selects the first item of a list. tail selects all but the first item of a list. prefix creates a list from a head item and a tail list. unitlist creates a list of one item. append joins two lists. concatenate joins the members of a list of lists. union is a synonym for append and unitset is a synonym for unitlist.

---

†Here, and throughout the report, functional application takes precedence over the concatenation operator.

Two functions are defined which involve sets (lists) of generated symbols. update-set-with-symbol adds a generated symbol to a set of generated symbols, knocking out of the set any existing element with the same identifier as the new element. update-set-with-set sums two sets of generated symbols in a similar manner, with elements of the second set knocking out elements of the first set.

There follow two functions that simplify the writing of syntax definitions. list1 accepts two string functions and returns a function which is the concatenation of some arbitrary non-zero number of occurrences of the second argument separated by occurrences of the first argument. In all cases, the first argument corresponds to some set of delimiters.

list2 is a generalised function-producing function for the definition of a class of phrase class functions with a certain structure, essentially that of a list of names in which each name can occur at most once. This structure is exhibited in declarative programming languages by lists of phrases in which each phrase depends on the set of declared identifiers and each phrase adds a new identifier to this set. Examples in Algol are a declaration list (block head) and a label declaration list.

Consider a label declaration list, for example. This is derived from:

label-declaration ^ : ^ label-declaration ^ : ^ ..... ^ : ^ label-declaration ^ :

In the left-to-right production of such a list, the first label must be different from the set l of local identifiers so far declared and each successive label must be different both from the members of l and from the labels which precede it in the list. This is expressed by making label-declaration a function of the set of local identifiers and by adding each new label to the set of local identifiers immediately it is declared. The first label of a label declaration list is produced by a call of label-declaration with argument l; if this first label is p, then the second label is produced by a call of label-declaration with argument $l \cup \{ p \}$; similarly for the production of each label. list2 has been designed to handle this kind of non-context-free syntactic pattern.

list2 creates a function from three functions f, g and h. In our applications of list2, f is a phrase class function (typically, label-declaration) and list2 f g h is also a phrase class function (label-declaration-list). f is called an arbitrary non-zero number of times. g is used to create the argument for each successive call from the result of the preceding call. h is used to combine the values resulting from successive calls of f. For example, in the case of a label declaration list, g creates the argument for a call of label-declaration by adding the label produced by the preceding call into the argument set of local identifiers; and h (in part) inserts the commas between the labels of the produced list. h is a function of two arguments. In our use of list2, g is always a selector function operating on a constructed object.

The function list2 is defined as follows: list2 f g h is a function. A value of this function for an argument x is given either by applying f to x, or by first applying f to x to obtain a, then recursively applying list2 f g h to g a to obtain b, and finally combining a and b by means of h. Thus:

```
rec list2 f g h x = a | h a (list2 f g h (g a))
                    where a = f x
```

The definition of label-declaration-list in terms of list2 is:

```
label-declaration-list l →
list2 label-declaration locals λ x . λ y . < text :     textofx ^ : ^textofy
                                            rules :     union rulesofx symbolsofy
                                            symbols :   union symbolsofx symbolsofy
                                            locals :    localsofy>
```

The h function here combines two 4-component objects into a single 4-component object. The g function, locals, is a selector for one of the components of these objects.

Given two sets of string functions A, B, we define their concatenated-cross-product to be the set $\{ f \hat{} g \mid f \in A_{\wedge}\ g \in B \}$. The function function-cross-product is the extension which accepts an arbitrary number of sets of string functions.

The function new-value accepts a list, a function and an argument for that function. It returns a value of the given function for the given argument such that the value does not occur in the list. For this to make sense, the argument function must be non-deterministic and have an infinite number of possible values for a given argument. These conditions are satisfied by our use of the new-value function to produce a new local identifier: we apply new-value to a list of local identifiers, the 0-ary function identifier and the empty argument.

rule is the constructor function for creating a rule from a nonterminal and a set of productions.

Finally, get-prod is a function which accepts a nonterminal symbol x and a set r of rules and returns a replacement for x out of r. This replacement is a function of a set of symbols. If r contains no rule for x, get-prod returns the constant function k nullstring. get-prod uses an unspecified function random-element which makes a random choice of an element from its list argument.

We are now in a position to describe the definition of the Algol subset. This is done by indicating how an informal explanation of a line of the definition can be derived more or less

mechanically from the line itself. Some of the locutions used are as follows. Functional application in a combination M N is rendered by 'an M is produced from an N'. Arguments r, s, l are referred to as 'a (the) set r of rules', 'a (the) set s of symbols' and 'a (the) set l of locals'. The part of a constructed object obj with selector sel is referred to as 'the sel part of obj'.

Note that the selector operator **of** is assumed to have higher precedence than normal functional application, which has higher precedence than the concatenation operators $\hat{}$ and $\hat{\hat{}}$. Also, recall that a symbol typed in bold, e.g. **arithmetic-operator**, denotes a string of some set of strings that is considered to be a primitive and is left undefined.

In the following excerpt from the Appendix, the indented structure is not shown:

program → block () ()

A program is a block produced from an empty set of rules and an empty set of symbols

where rec block r s → **begin** $\hat{}$ block-body r s $\hat{}$ **end**

where a block is produced from a set r of rules and a set s of symbols, and is a **begin** followed by a block-body produced from r and s followed by an **end**

where block-body r s → functionofx (union r rulesofx) (update-set-with-set s symbolsofx)

where a block-body is produced from a set r of rules and a set s symbols, and is produced by the function part of x from the set of rules given by adding to r the rules part of x, and the set of symbols given by updating s with the symbols part of x.

where x → block-body* ( )

where x is a block-body* produced from an empty set of locals

where block-body* l → < function : functionofx $\hat{\hat{}}$; $\hat{\hat{}}$ functionofy
rules : union rulesofx rulesofy
symbols : union symbolsofx symbolsofy
locals : localsofy >

where a block-body* is produced from a set l of locals and consists of:

i. the concatenation of the function part of x, the constant function k_1 ; and the function part of y
ii. the union of the rules part of x and the rules part of y
iii. the union of the symbols part of x and the symbols part of y
iv. the locals part of y

where y → statement-list* localsofx

where y is a statement-list* produced from the locals part of x (*Note*: x is a declaration-list*)

where rec statement-list* l →
list2 statement* locals λ x . λ y . < function : functionofx ^^ ; ^^ functionofy
rules : union rulesofx rulesofy
symbols : union symbolsofx symbolsofy
locals : localsofy >

where a statement-list* is produced from a set l of locals and is constructed from a sequence of statement*s (where the first statement* of the sequence is produced from l and each subsequent statement* is produced from the locals part of its predecessor) and consists of:

i. the concatenation of the function parts of the statement*s separated by the constant function k_1 ;
ii. the union of the rules parts of the statement*s
iii. the union of the symbols parts of the statement*s
iv. the locals part of the final statement*.

where rec statement* l →
unlabelled-statement* l | <function : functionofx ^^ functionofy
rules : union rulesofx rulesofy
symbols : union symbolsofx symbolsofy
locals : localsofy>

where a statement* is produced from a set l of locals and is either an unlabelled-statement* produced from l or consists of:

i. the concatenation of the function part of x and the function part of y
ii. the union of the rules part of x and the rules part of y
iii. the union of the symbols part of x and the symbols part of y
iv. the locals part of y

where y → unlabelled-statement* localsofx

where y is an unlabelled-statement* produced from the locals part of x

---

where x → label-declaration-list l

where x is a label-declaration-list produced from a set l of locals

where label-declaration-list l →
list2 label-declaration locals λ x . λ y . < text : textofx ^ : ^ textofy
rules : union rulesofx rulesofy
symbols : union symbolsofx symbolsofy
locals : localsofy >

where a label-declaration-list is produced from a set l of locals and is constructed from a sequence of label-declarations (where the first label-declaration of the sequence is produced from l and each subsequent label-declaration is produced from the locals part of its predecessor) and consists of:

i. the concatenation of the text parts of the label-declarations separated by colons
ii. the union of the rules parts of the label-declarations
iii. the union of the symbols parts of the label-declarations
iv. the locals part of the last label-declaration

where label-declaration l → < text: x
rules : rule 'label' definiensofx
symbols : symbolsofx
locals : localsofx>

where a label-declaration is produced from a set l of locals and consists of:

i. x
ii. a rule for label whose right-hand side is the definiens part of x
iii. the symbols part of x
iv. the locals part of x

where x → declaration-identifier l

where x is a declaration-identifier produced from a set l of locals

---

where declaration-identifier l →
< text : x
definiens : unitset λ r . λ s . **if** y ϵ s **then** x **else nullstring**
symbols : unitset y
locals : prefix x l>

where a declaration-identifier is produced from a set l of locals and consists of:

i. x

ii. a function of a set r of rules and a set s of symbols which tests if y is a member of s and if it is returns x and otherwise returns the nullstring
iii. a set whose sole element is y
iv. a set obtained by adding x to the set l of locals.

where y →construct-symbol x t

where y is a generated symbol created from x and t

where x → new-identifier l

where x is a new-identifier produced from a set l of locals

where new-identifier l → new-value l identifier ( )

where a new-identifier is produced from a set l of locals and is a new-value produced from the set l of locals, the 0-ary function identifier and an empty argument

and t → token ( )

and t is a new token

6.4 A DIFFERENT VIEW-POINT

In this section we discuss an alternative interpretation of syntax definitions which has a technical advantage over the conventional interpretation.

We interpreted a context-free grammar as the specification of an automaton (production system) whose input is a unique starting symbol (sentence symbol) and whose final output is some string of the language defined. In general, the automaton is nondeterministic since its auxiliary symbols (nonterminal symbols) may have more than one rule.

To represent noncontext-free languages we generalised a rule so that a nonterminal symbol may be replaced by a complex object with, in general, an applicative structure. This generalisation ran us into a difficulty however, (see Section 6.2), due to the conjunction of functional application and nondeterminism. This forced us to take a non-standard and restrictive interpretation of applicative structure to ensure that any nondeterminism is removed from an operand before an operator is applied to it. This difficulty and the restriction disappear if we take the following view of syntax definitions.

A syntax definition may be interpreted as the explicit recursive definition of a set of terminal strings, involving the definition of auxiliary sets of strings. Under this interpretation, a

nonterminal symbol stands for a set of strings and the operators | and ^ denote *union* and *concatenated cross-product* respectively:

$$A \mid B = \{ x \mid x \in A \vee x \in B \}$$
$$A \hat{} B = \{ \text{append } x\ y \mid x \in A \wedge y \in B \}$$

where A and B are sets of strings. If **emptyset** is the empty set of strings and **emptystringset** is the set whose only member is the emptystring then, for any set A:

$$A \mid \textbf{emptyset} = \textbf{emptyset} \mid A = A$$
$$A \hat{} \textbf{emptyset} = \textbf{emptyset} \hat{} A = \textbf{emptyset}$$
$$A \hat{} \textbf{emptystringset} = \textbf{emptystringset} \hat{} A = A$$

Thus, **emptyset** and **emptystringset** act as zero and identity elements respectively.

Syntax definitions can be generalised under the string set interpretation analogously to their generalisation under the production interpretation. But now there is no nondeterminism and it is no longer necessary to ~~predicate~~ assume a restricted rule of functional application.

Corresponding to the production-oriented language definition of the Appendix, a definition can be made in terms of sets of strings. This makes use of the function map defined by:

$$\text{map } f\ s = \{ f\ x \mid x \in s \}$$

map applies its functional argument f to all the items of its set argument s and returns the set of results. To improve readability we can introduce infix operators **for** and ϵ. Then, in place of map f s, the following can be written:

f x **for** x ϵ s

The following excerpt from the Appendix serves as an example:

```
compound-statement* l → < function :  begin ^^ functionofx ^^ end
                          rules :     rulesofx
                          symbols :   symbolsofx
                          locals :    localsofx >
                          where x → statement-list* l
```

becomes:

```
compound-statement* l → < function :  begin ^^ functionofx ^^ end
                          rules :     rulesofx
                          symbols :   symbolsofx
                          locals :    localsofx >
                        for x ∈ statement-list* l
```

For the analogue of the Appendix definition it is helpful to extend the **for** notation to allow expressions of the form:

f x y **for** x ϵ s **and** y ϵ t

with meaning map f (s $\otimes$ t), where $\otimes$ is an operator for set cross-product, and also to allow expressions of the form:

f x y **for** y ϵ g x **for** x ϵ s

with meaning sumset (map (λ x . map (f x) (g x)) s). sumset forms the union of a set of sets. g is a set-valued function.

7. RELATED WORK

After a characterisation of work on the syntax of programming languages, this section reviews very briefly the work of some other authors on the formal description of non-context-free syntax.

Three main areas of work in the syntax of programming languages can be distinguished (ignoring relevant work in the theory of automata). The first concerns syntax-directed compilers, in which a BNF grammar is used to structure the compiler. This is realised either by an interpretive parsing routine which is driven by a stored form of the grammar and uses a push-down organisation[8], or by a set of interdependent parsing routines which are mechanically produced from the grammar[9]. In both cases the BNF grammar is augmented by compiling directives.

In syntax-directed methods, a grammar is used in analysing a given string as a prelude to determining its meaning. The power and efficiency (and in the case of ambiguity, the meaning) of a grammar used in this way depend not only on the grammar but on the algorithm which uses it. The second area of development in syntax has been concerned with searching for special types of context-free grammar which lend themselves to economical parsing. This work seeks to establish new nodes below the context-free node in the hierarchy of grammars.

The third area of development, on the other hand, is concerned with establishing new nodes *above* context-free grammars in the hierarchy and is motivated by the inability of context-free grammars to fully describe the syntax of present-day high-level programming languages. Although presently of interest mainly to people working in formal language definition, solutions to this problem could lead to new techniques in compiler production.

It is to the third line of development that the present report aims to contribute.

Some of the work done to date on the formal specification of the complete syntax of programming languages is now mentioned.

Context-free languages are Chomsky Type 2 languages. In Chomsky Type 1 or *context-sensitive languages*[10], production rules have the form:

$$x\ A\ y \rightarrow x\ a\ y$$

x, y, a are strings and A is a nonterminal symbol. A can produce a if A occurs in the context characterised as being bordered on the left by x and on the right by y. Although Type 1 grammars are powerful enough to describe non-context-free features of programming languages, their use is not a practical proposition owing to the extreme complexity of such descriptions.

Whitney [11] defines *table grammars* which are an extension of BNF grammars. The right-hand side of a production contains table functions which carry out table operations similar to the dictionary table operations in compilers. The approach can thus be considered as an idealisation

of the notion of syntax-driven compilers. Whereas the generation of a string from a BNF grammar involves a single object – a string which is successively modified by application of the productions of the grammar – generation from a table grammar involves three objects: a string which is a head-string of the eventual language string, a pushdown string store for the as yet unwritten parts of the productions selected in the generation process and a table for recording all declarative information met so far. For block-structure languages, the table is replaced by a pushdown table store. A table grammar is defined only for a left-to-right generation sequence, that is, one in which at each replacement step the leftmost nonterminal symbol is the one chosen for rewriting. The functions incorporated in productions are designed as follows:

1. They return a terminal string value.
2. They may act on the table by side effect.
3. They may apply a predicate to the table, which if not satisfied will cause the generation to be abandoned.

A use before declaration is considered as a request for a subsequent declaration. The request is recorded in the table and predicates are used to check that all such requests have been satisfied. Table grammars are able to handle the declaration of scalar variables but fail to handle more complicated situations.

Ghandour[12] and Donovan and Ledgard[13] have taken an approach using *canonic systems*, which are variants of Post's[14] canonical systems and Smullyan's[15] elementary formal systems. Canonic systems are capable of specifying any recursively enumerable set and are used here to recursively define sets of strings. The syntax of a computer language is specified by defining a set which is just the set of all syntactically valid programs. Although canonic systems are capable of fully describing the syntax of programming languages, the specification of even severe subsets of actual high-level languages tend to lose clarity and intuitive appeal – see the specification of 'Little PL/I' given by Donovan and Ledgard[13].

The Algol 68 report[16] uses a method for syntax definition due to van Wijngaarden. Chastellier and Colmerauer[17] call this the method of *W-grammars*. Sintzoff[18] has shown that W-grammars are powerful enough to define every recursively enumerable set. However, in the Algol 68 report, which defines a proper program to be a program satisfying certain *context conditions*, these context conditions are described in natural language.

Di Forino[19] was the first to propose the line of attack taken in this report. Our discovery of the concept of a dynamic set of context-free rules was made independently of the work of di Forino.

8. SUMMARY

In summarising the paper it is possible, and hopefully interesting, to trace the development of the ideas. The authors were faced with the problem of writing a "test-case" generator: a program which created syntactically correct but meaningless programs for any arbitrary language. As has been indicated above, context-free languages were not adequate for our purposes. The idea that the complete syntax of a language could be defined by a context-free grammar which has the power to modify itself occurred to us in late 1966.

In order to utilise this simple idea one has to provide a notation with which the dynamic changes can be described. A notation was developed[20] with which several languages were defined and this was used as the basis for our "text-case generator"[21]. However, this notation had several algorithmic facets which the authors considered were unacceptable in a language-definition language.

In a rather intermittent way, the authors discussed the notation during 1968/9 and eventually developed the current notation in which one language (that of functions written in the lambda-calculus) is used to describe both the syntax rules and their modification.

The authors hope that this uniformity has provided a language in which the syntax of programming languages can be clearly presented. We are, however, aware that our work is extremely informal by the standard of most work on grammars, but are unable to fill this gap. If others find the notation as useable as do the authors, more skilled hands may be applied to the problem.

an effort is currently being made to fill this gap.

REFERENCES

1. Backus, J. W., "The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference", *Proceedings of the International Conference on Information Processing,* pp. 125-132, June 1959.
2. Naur, Peter (Ed.) et al, "Revised report on the algorithmic language Algol 60", *CACM,* Vol. 6, No. 1, pp. 1-17.
3. Floyd, R. W., "On the nonexistence of a phrase structure grammar for Algol 60", *CACM,* Vol. 5, No. 9, pp. 483-484.
4. Higman, B., *A comparative study of programming languages,* MacDonald Computer Monographs, London, 1967.
5. Church, A., *Introduction to mathematical logic,* Vol. 1, Princeton University Press, Princeton.
6. Landin, P. J., "The mechanical evaluation of expressions", *Computer Journal,* Vol. 6, No. 4, pp. 308-320.
7. Gorn, S. (Ed.), "ECMA subset of Algol 60", *CACM,* Vol. 6, No. 10, pp. 595-599, October 1963.
8. Floyd, R. W., "The syntax of programming languages – a survey", *IEEE Trans,* EC 13, Vol. 4, p. 346, August 1964.
9. Foster, J. M., "A syntax improving program", *Computer Journal,* Vol. II, p. 31, May 1968.
10. Chomsky, N., "Formal properties of grammars", in D. Luce, R. Bush and E. Galanter (Eds.) *Handbook of Mathematical Psychology,* John Wiley and Sons, New York.
11. Whitney, G. E., "The generation and recognition properties of table languages", *IFIP Congress 68,* Software 1, pp. B18-22, Edinburgh, 1968.
12. Ghandour, Z.J., "Formal systems and syntactical analysis", Ph.D. Dissertation, Yale University, 1968.
13. Donovan, J.J. and Ledgard, H.F., "A formal system for the specification of the syntax and translation of computer languages", *AFIPS Conference Proceedings, Fall Joint Computer Conference 31,* pp. 553-580, 1967.
14. Post, E.L., "Formal reductions of the general combinational decision problem", *American Journal of Mathematics,* Vol. 65, pp. 197-215.
15. Smullyan, R. M., *Theory of formal systems,* Princeton University Press, Princeton, 1961.
16. Van Wijngaarden, A. (Ed.) et al, *Report on the algorithmic language Algol 68,* Mathematisch Centrum, Amsterdam, MR 101, October 1969.

17. de Chastellier, G. and Colmerauer, A., "W-grammar", *Proceedings of 24th National ACM Conference,* pp. 511-518, 1969.

18. Sintzoff, M., "Existence of a Van Wijngaarden syntax for every recursively enumerable set" *Annales de la Société Scientifique de Bruxelles,* Vol. 81, No. II, pp. 115-118, Brussels, 1967.

19. di Forino, A. C., "Some remarks on the syntax of symbolic programming languages", *CACM,* Vol. 6, No. 8, pp. 456-460.

20. Hanford, K. V. and Jones, C. B., "An approach to context dependency", IBM Programming Conference, June 1967.

21. Hanford, K.V., "Automatic generation of test cases", *IBM Systems Journal,* Vol. 9, No. 4, 1970.

APPENDIX

STRUCTURE DEFINITIONS

An x-list is either null
or nonnull and has a head which is an x
and a tail which is an x-list

A symbol has an id which is an identifier
and a name which is a token

A rule has a l.h.s. which is a nonterminal symbol
and a r.h.s. which is a production-list

PRIMITIVE FUNCTIONS

List Functions

prefix is the constructor function for creating a list from a given head and tail. The functions head, tail and prefix satisfy the following axioms:

```
prefix  (head l) (tail l) = l
head  (prefix x l) = x
tail  (prefix x l) = l
```

```
unitlist x = prefix x ( )
```

```
rec append l m = if null l then m
                 else prefix (head l) (append (tail l) m)
```

```
rec concatenate L = if null L then ( )
                    else append (head L) (concatenate (tail L))
```

Symbol-table Functions

```
rec update-set-with-symbol t s =
        if null t then unitlist s
        else if id (head t) = id s then prefix s (tail t)
        else prefix (head t) (update-set-with-symbol (tail t) s)
```

update-set-with-symbol accepts a symbol-set and a symbol. A symbol is an (identifier, token) pair. If some symbol in the set has the same identifier as the argument symbol, the set symbol is replaced by the argument symbol. Otherwise, the argument symbol is added to the set.

```
rec update-set-with-set t u =
        if null u then t
        else update-set-with-set (update-set-with-symbol t (head u)) (tail u)
```

update-set-with-set creates a new set of symbols from an old set and an updating set.

String Functions

rec list1 f g = g | g $\hat{\hat{}}$ f $\hat{\hat{}}$ list1 f g
rec list2 f g h x = a | h a (list2 f g h (g a))
where a = f x

function-cross-product $(A_1, A_2, \ldots, A_n) = \{ f_1 \hat{\hat{}} f_2 \hat{\hat{}} \ldots \hat{\hat{}} f_n \mid f_i \in A_i, i = 1, 2, \ldots; n \}$

New-Value Function

rec new-value l f a = (λ x . **if** x ∈ l **then** new-value l f a **else** x)(f a)

Rule Functions

rule x y creates a production rule from a nonterminal x and a list y of productions.

rec get-prod x r = **if** null r **then** k **nullstring**
else if x = l.h.s. (head r) **then** random-element (r.h.s.(head r))
else get-prod x (tail r)

```
program → block () ()
where rec block r s → begin^block-body r s^end
        where block-body r s → functionofx (union r rulesofx) (update-set-with-set s symbolsofx)
              where x → block-body* ()
                    where block-body* l → <function: functionofx^^;^^functionofy
                                           rules:    union rulesofx rulesofy
                                           symbols:  union symbolsofx symbolsofy
                                           locals:   localsofy>
                          where y → statement-list* localsofx
                                where rec statement-list* l → list2 statement* locals λx.λy.<function: functionofx^^,^^functionofy    l
                                                                                             rules:    union rulesofx rulesofy
                                                                                             symbols:  union symbolsofx symbolsofy
                                                                                             locals:   localsofy>
                                      where rec statement*-l → unlabelled-statement* l | <function: functionofx^^functionofy
                                                                                           rules:    union rulesofx rulesofy
                                                                                           symbols:  union symbolsofx symbolsofy
                                                                                           locals:   localsofy>
                                            where y → unlabelled-statement* localsofx
                                                  where unlabelled-statement* l → unconditional-statement* l | conditional-statement* l | for-statement* l
                                                        where rec unconditional-statement* l → <function: assignment-statement> | <function: goto-statement>
                                                                                               <function: block> | compound-statement* l
                                                              where assignment-statement r s → real-left-part-list r s^:=^arithmetic-expression r s |
                                                                                               integer-left-part-list r s^:=^arithmetic-expression r s |
                                                                                               boolean-left-part-list r s^:=^boolean-expression r s
                                                                    where real-left-part-list → list1 := real-variable
                                                                      and integer-left-part-list → list1 := integer-variable
                                                                      and boolean-left-part-list → list1 := boolean-variable
                                                                and goto-statement r s → goto^designational-expression r s
                                                                and compound-statement* l → <function: begin^^functionofx^^end
                                                                                            rules:    rulesofx
                                                                                            symbols:  symbolsofx
                                                                                            locals:   localsofx>
                                                                    where x → statement* l
                                                                and conditional-statement* l → <function: if^^boolean-expression^^then^^functionofx
                                                                                                  rules:    rulesofx
                                                                                                  symbols:  symbolsofx
                                                                                                  locals:   localsofx>
                                                                                                <function: if^^boolean-expression^^then^^functionofx^^else^^functionofy
                                                                                                  rules:    union rulesofx rulesofy
                                                                                                  symbols:  union symbolsofx symbolsofy
                                                                                                  locals:   localsofx>
                                                                    where y → statement* localsofx
                                                                    where x → unconditional-statement* l
                                                                and for-statement* l → <function: for^^arithmetic-variable^:=^^list1, for-list-element^^do^^functionofx
                                                                                          rules:    rulesofx
                                                                                          symbols:  symbolsofx
                                                                                          locals:   localsofx>
                                                                    where x → statement* l
                                                                    where for-list-element r s → arithmetic-expression r s |
                                                                                                 arithmetic-expression r s^step^arithmetic-expression r s^until^arithmetic-expression r s |
                                                                                                 arithmetic-expression r s^while^boolean-expression r s
                                            where x → label-declaration-list l
                                                  where label-declaration-list l → list2 label-declaration locals λx.λy.<text:    textofx^textofy    l
                                                                                                                         rules:    union rulesofx rulesofy
                                                                                                                         symbols:  union symbolsofx symbolsofy
                                                                                                                         locals:   localsofy>
                                                        where label-declaration l → <text:    x^:
                                                                                     rules:    rule 'label' definiensofx
                                                                                     symbols:  symbolsofx
                                                                                     locals:   localsofx>
                                                              where x → declaration-identifier l
where x → declaration-list* ()
      where declaration-list* l → list2 declaration* locals λx.λy.<function: functionofx^^;^^functionofy    l
                                                                   rules:    union rulesofx rulesofy
                                                                   symbols:  union symbolsofx symbolsofy
                                                                   locals:   localsofy>
            where declaration* l → type-declaration* l | array-declaration* l | switch-declaration* l
                  where type-declaration* l → <function: λr.λs.textofx^textofy
                                               rules:    rule simple-variable-definiendumofx definiensofy
                                               symbols:  symbolsofy
                                               locals:   localsofy>
                        where x → type
                              where type → <text:    real
                                            simple-variable-definiendum: 'real-simple-variable'> | <text:    integer
                                                                                                    simple-variable-definiendum: 'integer-simple-variable'> | <text:    boolean
                                                                                                                                                               simple-variable-definiendum: 'boolean-simple-variable'>
                                and y → declaration-identifier-list l
                            and array-declaration* l → <function: λr.λs.textofx^textofy
                                                        rules:    union (rule subscripted-variable-definiendumofx subscripted-variable-definiensofy)
                                                                        (rule array-definiendumofx array-definiensofy)
                                                        symbols:  symbolsofy
                                                        locals:   localsofy>
                              where x → type
                                    where type → <text:    array | real^array
                                                  subscripted-variable-definiendum: 'real-subscripted-variable'
                                                  array-definiendum:                'real-array'> | <text:    integer^array
                                                                                                     subscripted-variable-definiendum: 'integer-subscripted-variable'
                                                                                                     array-definiendum:                'integer-array'> | <text:    boolean^array
                                                                                                                                                           simple-variable-definiendum: 'boolean-subscripted-variable'
                                                                                                                                                           array-definiendum:           'boolean-array'>
                                      and y → array-list l
                                          where array-list l → list2 array-segment locals λx.λy.<text:    textofx^;^textofy    l
                                                                                                  subscripted-variable-definiens: union subscripted-variable-definiensofx subscripted-variable-definiensofy
                                                                                                  array-definiens:                union array-definiensofx array-definiensofy
                                                                                                  symbols:                        union symbolsofx symbolsofy
                                                                                                  locals:                         localsofy>
                                                where array-segment l → <text:    textofx^;^textofy    l
                                                                         subscripted-variable-definiens: function-cross-product (definiensofx, unitset [,subscript-list-definiensofy,unitset]
                                                                         array-definiens:                definiensofx
                                                                         symbols:                        symbolsofx
                                                                         locals:                         localsofx>
                                                      where x → declaration-identifier-list l
                                                        and y → bound-pair-list ()
                                                            where bound-pair-list w → list2 bound-pair empty λx.λy.<text:    textofx^,^textofy    w
                                                                                                                    subscript-list-definiens: function-cross-product (unitset arithmetic-expression,unitset ,,subscript-list-definiensofy)
                                                                  where bound-pair w → <text:    number^:^number
                                                                                        subscript-list-definiens: unitset arithmetic-expression>
                                      and switch-declaration* l → <function: switch^textofx^:=^switch-body
                                                                   rule 'switch' definiensofx
                                                                   symbolsofx
                                                                   localsofx>
                                            where x → declaration-identifier l
                                              and switch-body → list1 ; designational-expression
                                        where declaration-identifier-list l → list2 declaration-identifier locals λx.λy.<text:    textofx^,^textofy    l
                                                                                                                 definiens: union definiensofx definiensofy
                                                                                                                 symbols:   union symbolsofx symbolsofy
                                                                                                                 locals:    localsofy>
where declaration-identifier l → <text:    x
                                  definiens:  unitset λr.λs.if y ∈ s then x else nullstring
                                  symbols:    unitset y
                                  locals:     prefix x l>
      where y → construct-symbol x t
      where x → new-identifier l
            where new-identifier l → new-value l identifier ()
        and t → token ()
where rec arithmetic-expression r s → simple-arithmetic-expression r s |
                                      if^boolean-expression r s^then^simple-arithmetic-expression r s^else^arithmetic-expression r s
    and simple-arithmetic-expression → list1 arithmetic-operator arithmetic-primary |
                                       add-operator^list1 arithmetic-operator arithmetic-primary
        where arithmetic-primary r s → unsigned-number |
                                       arithmetic-variable r s |
                                       (^arithmetic-expression r s^)
    and boolean-expression r s → simple-boolean-expression r s |
                                 if^boolean-expression r s^then^simple-boolean-expression r s^else^boolean-expression r s
        where simple-boolean-expression → list1 boolean-binary-operator boolean-secondary
              where boolean-secondary r s → boolean-primary r s |
                                            ¬^boolean-primary r s
                    where boolean-primary r s → boolean-constant |
                                                boolean-variable r s |
                                                relation r s |
                                                (^boolean-expression r s^)
                          where relation r s → simple-arithmetic-expression r s^relational-operator^simple-arithmetic-expression r s
    and designational-expression r s → simple-designational-expression r s |
                                       if^boolean-expression r s^then^simple-designational-expression r s^else^designational-expression r s
        where simple-designational-expression r s → label r s |
                                                    switch r s^[^arithmetic-expression r s^] |
                                                    (^designational-expression r s^)
              where label r s → get-prod 'label' r s
                and switch r s → get-prod 'switch' r s
where arithmetic-variable r s → get-prod 'real-simple-variable' r s |
                                get-prod 'integer-simple-variable' r s |
                                get-prod 'real-subscripted-variable' r s |
                                get-prod 'integer-subscripted-variable' r s
  and boolean-variable r s → get-prod 'boolean-simple-variable' r s |
                             get-prod 'boolean-subscripted-variable' r s
```

Figure 2. A Definition of the Complete Syntax of a Subset of ECMA Algol