The School of Squiggol
A History of the Bird–Meertens Formalism

Jeremy Gibbons

University of Oxford

Abstract. The Bird–Meertens Formalism, colloquially known as "Squiggol", is a calculus for program transformation by equational reasoning in a function style, developed by Richard Bird and Lambert Meertens and other members of IFIP Working Group 2.1 for about two decades from the mid 1970s. One particular characteristic of the development of the Formalism is fluctuating emphasis on novel 'squiggly' notation: sometimes favouring notational exploration in the quest for conciseness and precision, and sometimes reverting to simpler and more rigid notational conventions in the interests of accessibility. This paper explores that historical ebb and flow.

1 Introduction

In 1962, IFIP formed Working Group 2.1 to design a successor to the seminal algorithmic language Algol 60 [4]. WG2.1 eventually produced the specification for Algol 68 [63, 64]—a sophisticated language, presented using an elaborate two-level description notation, which received a mixed reception. WG2.1 continues to this day; technically, it retains responsibility for the Algol languages, but practically it takes on a broader remit under the current name *Algorithmic Languages and Calculi*. Over the years, the Group has been through periods of focus and periods of diversity. But after the Algol 68 project, the period of sharpest focus covered the two decades from the mid 1970s to the early 1990s, when what later became known as the Bird–Meertens Formalism (BMF) drew the whole group together again. It is the story of those years that is the subject of this paper.

BMF arose from the marriage of the work of Richard Bird (then at the University of Reading) in recursive programming [15, 14] and of Lambert Meertens (then at the Mathematisch Centrum in Amsterdam) in programming language design, notably ABC [48, 34].[1] The motivation for the BMF is *transformational programming*: developing an efficient program by starting with an obviously correct but possibly hopelessly inefficient—maybe even unexecutable—initial specification, then applying a series of meaning-preserving transformations to yield an extensionally equivalent but acceptably efficient final program. In other words, the approach follows Christopher Strachey's First Law of Programming: "Decide what you want to say before you worry about how you are going to say it" [5].

[1] Guido van Rossum, who worked on the ABC project in Amsterdam, was mentored by Meertens and went on to design Python [55] based on some of the ideas in ABC.

The essence of the formalism is a concise functional notation. The functional approach ensures referential transparency, and admits the straightforward manipulation technique of substitution of equals by equals, as in high school algebra. Concision is necessary in order to make such manipulations feasible with just pen and paper. In particular, like APL [36], BMF embraced funny symbols such as a slash for reduction ("$+/$" sums a sequence of numbers), arrows for directed folds and scans ("$\not\rightarrow$" and "$/\!\!/\!\!\rightarrow$", now called "`foldl`" and "`scanl`" in Haskell), and banana brackets ("$(\!|\ldots|\!)$") for homomorphisms; this tendency led to the notation being nicknamed *Squiggol*. Little emphasis was placed on executability: the notation was 'wide-spectrum' [7], accommodating convenient specification notations such as inverses and intersection as well as a sublanguage with an obvious correspondence to executable code.

The BMF research paradigm consisted of establishing a body of theorems about recurring problem structures and corresponding solution techniques. Typical examples are *fusion* properties (combining two traversals over a data structure into one), *scan lemmas* (replacing the independent reductions of overlapping parts of a data structure with a single accumulation across the whole), and *Horner's Rule* (exploiting distributivity, as for products over sums in polynomial evaluation). These three formed the core of a beautiful derivation of a linear-time solution to the Maximum Segment Sum problem [23], a central example in the BMF canon. The effort culminated in Bird and de Moor's book *The Algebra of Programming* [10], with a collection of theorems expressed in a relational notation providing greedy and dynamic-programming solutions to optimization problems.

WG2.1's passion for the approach started to fade after Bird and de Moor's book appeared, and the group's focus diversified again. Partly this was due to falling out of love with the Squiggolly notation, which may be convenient for aficionados but excludes the unfamiliar reader; later work favours more conventional syntax. It was also partly due to dissatisfaction with the relational approach, which seems necessary for many optimization problems but is too complicated for most readers (and even for writers!); in fact, Bird is returning in a forthcoming book [24] to tackling many of the same 'Algebra of Programming' optimization problems but using a nearly completely functional approach. The purpose of this paper is to pick out some of the lessons from this ebb and flow of enthusiasm.[2]

[2] From its start, my own research career has been intimately entwined with WG2.1 and the BMF, although I came somewhat late to the party. Bird supervised my DPhil dissertation (1987–1991) [35], and Meertens was my external examiner. I have worked on and off with Bird ever since, and most of my research has been inspired by the BMF; I am Bird's co-author on his forthcoming book [24]. I served as secretary of WG2.1 for thirteen years (1996–2009), during the Chairmanships of Doug Smith and Lambert Meertens, and then succeeded Meertens as Chair myself for the following six years (2009–2015).

2 Abstracto

The history of the development of Algol 68 has been well reported [38, 53, 37], and we will not dwell on it here. After 1968, WG2.1 spent a few years making small improvements to the language and clarifying its description, leading to the publication of the Revised Report [64] in 1974.[3] The Group then entered a brief 'what next?' phase, setting up a *Future Work* subcommittee chaired by Robert Dewar. This subcommittee in turn organized two conferences on *New Directions in Algorithmic Languages* in 1975 and 1976, with proceedings [56, 57] edited by Stephen Schuman. These conferences were public, intended to collect input from the broader community about research topics in algorithmic languages.

After that short period of scanning the horizon, the Group decided to focus again on specific topics. Robert Dewar, as Chair of the Future Work Subcommittee, wrote a letter to members in July 1977, in advance of Meeting #23 of the Group in Oxford in December of that year, explaining:

> *We have decided to break with our two year old 'tradition' of holding conferences with invited outside participants. These conference have been helpful in exploring ideas, but now it is time to get back to the work of our working group and concentrate on the resources of our membership.* [30]

The decision was for the Group to focus for the time being on two topics: programming languages for beginners, and "Abstracto". The former direction led to Meertens's development of ABC [42, 43, 34, 54] and hence eventually to Python [55]; but it is the latter that is of interest to us here.

The name *Abstracto* arose through a misunderstanding:

> *The first author [Geurts], teaching a course in programming, remarked that he would first present an algorithm "in abstracto" (Dutch* [sic] *for "in the abstract") before developing it in Algol 60. At the end of the class, a student expressed his desire to learn more about this Abstracto programming language.* [33]

Abstracto itself is defined in Dewar's letter as follows:

> *We have taken the name to describe a programming language some of whose features we know:*
> 1. *It is very high level, whatever that means.*
> 2. *It is suitable for expressing initial thoughts on construction of a program.*

[3] It is fair to say that the Reports did not meet with universal acclaim. One reason for the mixed reception was the use of van Wijngaarden's two-level grammar notation for describing the language, whereby a possibly infinite language grammar is generated by a finite meta-grammar. The Algol 68 experience has engendered a keen interest in notational issues within the Group ever since.

```
As to (S2), this fits (f) with the assertion
z•x^y=X^Y for p and y≠0 for b. For the mapping 0 we
can simply take the identity, since the "goal" is
to get y to 0. We thus refine (S2) to

        *(y≠0 → z,x,y := z′,x′,y′ | z′,x′,y′:
            z•x^y = X^Y & y≠0 ⊃ z′•x′^y′=X^Y & y′<y).

Using (g), this may again be refined to

        *(y≠0 → z,x,y := z′,x′,y′ | z′,x′,y′,r:
            z′= z•x^r & x′= x•x & y=2y′+r &
                (r=0 ∨ r=1)).

If operations / and % are available, satisfying y =
2(y/2)+(y%2) and (y%2=0 ∨ y%2=1), the use of the
unit list u = ZZ,x•x,y/2,y%2 in (d) of Lemma 2,
where ZZ is shorthand for (y%2=0→z ▯ y%2=1→z•x),
allows to simplify this to

        *(y≠0 → z,x,y := ZZ,x•x,y/2).
```

Fig. 1. Abstracto 84 [41]

3. *It need not be (and probably is not) executable. This arises either from efficiency considerations, or even non-effective dictions, say those involving infinite sets.*

Abstracto is not a specification language as such since it is still concerned with how to do things and not just what is to be done, but it allows the expression of the 'how' in the simplest and most abstract possible way. [30]

So Abstracto was envisioned as an *algorithmic language*: for describing the algorithmic steps in a computation, not just the input–output relation or similar behavioural specification. But it was still envisaged as an exploratory medium, a pen-and-paper notation, a 'tool of thought', rather than primarily an implementation language.

A representative example of Abstracto is shown in Figure 1. This is part of the development of a 'fast exponentiation' algorithm: given natural numbers X and Y, compute $z = X^Y$ using only $O(\log_2 Y)$ iterations. The first program shows a 'while' loop, with invariant $z \times x^y = X^Y$, variant y, and guard $y \neq 0$. The second program factors out $r = y \bmod 2$, refining the nondeterminism in the first program to a deterministic loop. Thus, Meertens' vision for Abstracto is a kind of refinement calculus for imperative programs, as later developed in much greater depth by Ralph Back [1, 2] and Carroll Morgan [49, 50].[4]

[4] Indeed, the loop body in the first exponentiation program is a 'specification statement' in Morgan's sense [49], albeit one without a frame.

Although it was intended as a focus for the whole Group, the work on a notation named Abstracto was mainly undertaken by Meertens and his group at the Mathematisch Centrum (later CWI) in Amsterdam, and the few published papers [33, 41] are written by them. (In 1987, Meertens very helpfully collected these papers—and other papers of his on BMF—into a reader [46] for WG2.1, interspersed with a retrospective commentary on the background to the original publications.)

However, other members of the Group were conducting parallel projects with similar goals. Fritz Bauer's group at the Technical University in Munich, including Helmuth Partsch, Bernhard Möller, and Peter Pepper, were working on the Computer-Aided, Intuition-Guided Programming (CIP) project [6–8], developing a wide-spectrum language to encompass both abstract specifications and efficient implementations of programs. Jack Schwartz, Robert Dewar, and Bob Paige at New York University designed SETL [58, 51, 59] as a language that accommodated the gradual transformation of specifications using high-level set-oriented dictions such as comprehensions into lower-level programs by instantiating abstract datatypes with concrete implementations and by applying 'strength reduction' [52] to loops. These are just two of the larger projects; there were many smaller ones as well.

3 Disillusionment and enlightenment

For some of the subsequent meetings of the Group, members were set specific problems to work on in advance [31], so that approaches and solutions could be presented at the meeting—applications such as a text editor and a patient monitoring system, and more technical problems such as string matching and longest upsequence. Meertens observed in the introduction to the Algorithmics paper [45] included in the Abstracto Reader [46]:

> *Using the framework sketched in* [41]*, I did most of the examples from the problem sets prepared for the Brussels meeting of WG2.1 in December 1979* [Meeting #26] *and the meeting in Wheeling WV in August 1980* [Meeting #27]*. On the whole, I was reasonably successful, but I nevertheless abandoned the approach.* [46]

To illustrate Meertens' disillusionment, consider the two programs shown in Figure 2. The problem is to find the (assumed unique) oldest inhabitant of the Netherlands, where the data is given by a collection dm of Dutch municipalities, and an array $mr[-]$ of municipal registers of individuals, one register per municipality. The program on the left combines all the municipal registers into one national register; the program on the right finds the oldest inhabitant of each municipality, and then findest the oldest among these "local Methuselahs". Provided that no municipality is empty of inhabitants, these programs have equivalent behaviour. However, one cannot reasonably expect precisely the transformation from one to the other to be present in any catalogue of transformations; the development should proceed by a series of simpler steps that

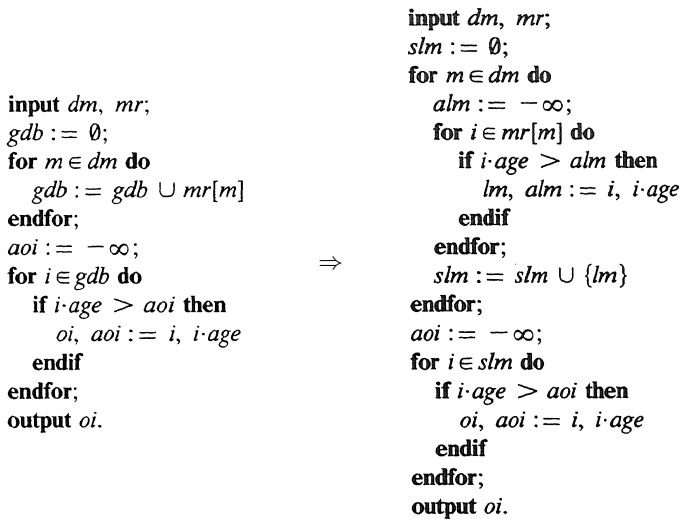

Fig. 2. The oldest inhabitant, in Abstracto [45]

themselves are present in a smaller and more manageable catalogue of more general-purpose transformations. But what would those atomic general-purpose transformations be?

Meertens continued:

> *The framework is, in fact, largely irrelevant: finding the theorems to be applied is the key to the development* [...] *If the Abstracto dream is to come true* [...] *the key 'transformations' are the mathematical theorems and not the boring blind-pattern-match manipulations that I looked upon until now as being 'the' transformations.* [46]

The breakthrough was to abandon the imperative Algol-like language and the corresponding refinement-oriented approach of Abstracto, and to switch to a more algebraic, functional presentation. Meertens continued:

> *Then came the Nijmegen meeting* [Meeting #28] *in May 1981, at which Richard Bird entered the stage*[5] *and presented a paper entitled "Some Notational Suggestions for Transformational Programming".*[6] *It used an applicative (functional) style* [...] *There were notations for high-level concepts, and just the kind of manipulations, at the right level, that you would want to see* [...] *Investigating this led to a whole lot of other*

[5] In fact, Bird and Meertens had both been present at Meeting #27 in Wheeling in August 1980, although the meeting of minds evidently had to wait a bit longer.

[6] Meertens' preface in the Abstracto Reader, Meertens' contemporary papers, and the WG2.1 minutes all record Bird's paper under the title "Some Notational Suggestions..." [17]; but the technical report [16] is entitled "Notational Suggestions...".

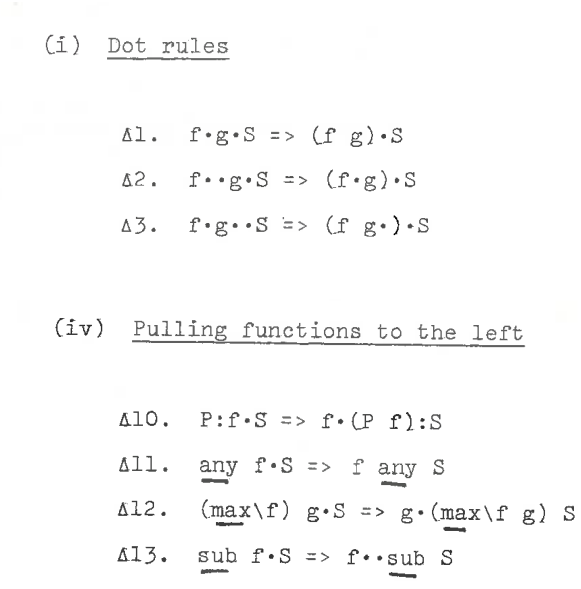

Fig. 3. Notational Suggestions for Functional Programming [16]

> *discoveries (the applicability to 'generic' structures* [...]*), and I was very excited about this.* [46]

Some of Bird's suggested notations are shown in Figure 3: "$f \cdot S$" for mapping function f over collection S, "$P : S$" for filtering collection S to retain only elements satisfying predicate P, "$\underline{\text{any}}\, S$" for choosing an arbitrary element of (nonempty) collection S; "$(\underline{\text{max}}\backslash f)\, S$" for the element of collection S that maximizes function f; "$\underline{\text{sub}}\, S$" for the powerset of collection S; juxtaposition for function composition; and so on. Thus rule $\Delta 1$ is what became known as "map fusion"[7] and $\Delta 10$ as "filter promotion"[8].

The equivalent transformation for the problem of the oldest inhabitant using Bird's suggestions [45] is:

$$\uparrow_{age}/ +/mr \cdot dm = \uparrow_{age}/(\uparrow_{age}/mr) \cdot dm$$

The left-hand side takes the oldest in the union of the registers of each of the municipalities, and the right-hand side takes the oldest among the oldest inhabitants of each of the municipalities. (Here, "$\oplus/$" reduces a collection using binary

[7] Applying g to every element of S and then f to every element of the result is the same as applying g then f to every element of S in a single pass.

[8] Applying f to every element and then filtering to keep the results that satisfy P is the same as filtering first, using the predicate "P after f", then applying f to every element that will subsequently satisfy P.

here are three rules that correspond to the above in the case of generalized union $\bigcup \{s1, s2, \ldots\} = s1 \cup s2 \cup \ldots$:

$$p{:}\bigcup X = \bigcup ((p{:}) * X)$$
$$f* \bigcup X = \bigcup ((f*) * X)$$
$$\max(\bigcup X) = \max(\max * X).$$

The second rule, for example, says that the application of f to each member of the union of a collection of sets gives the same result as taking the union of the collection of sets in which f is applied to each member. The rule is easy enough to justify:

$$\begin{aligned} \bigcup ((f*) * X) &= \bigcup \{f * x \mid x \in X\} \\ &= \bigcup \{\{fa \mid a \in x\} \mid x \in X\} \\ &= \{fa \mid a \in \bigcup X\} \\ &= f* \bigcup X. \end{aligned}$$

Fig. 4. The Promotion and Accumulation Strategies [18]

operator $\oplus$, absent from Bird's suggestions; "+" is binary union; "$\uparrow_f$" chooses which of two arguments has the greater f-value; "$g*$" maps function g over a collection; and function composition is indicated by juxtaposition.)

Clearly the BMF presentation is an order of magnitude shorter than the Abstracto one. It is also easier to see what form the small general-purpose transformation steps should take—just the kinds of equation shown in Figure 3.

4 Evolution

The BMF notations evolved through use, and through interactions at subsequent WG2.1 meetings. The Algorithmics paper [45] was presented at the Symposium on Mathematics and Computer Science in November 1983, when the Mathematisch Centrum in Amsterdam changed its name to the Centrum voor Wiskunde en Informatica (CWI).[9]

Bird and Meertens produced another working paper [12] for IFIP WG2.1, trying to converge on notational conventions such as operator precedence and semantic considerations such as indeterminacy for an "as yet unborn Science of Algorithmics". This was done together with Dave Wile, whose PhD thesis [65] had been on "a closely related approach to language design" [46]; although Wile was also a member of WG2.1, he could only contribute by post whereas Bird and

[9] Publication of the proceedings of this conference seems to have taken frustratingly long: in a 1984 working paper [44] using the same notation, Meertens cites the Algorithmics paper [45] as appearing in the year "$[]/(1984{\leq}) \triangleleft \mathbb{U}$", that is, the arbitrary choice of any number at least 1984. The same joke appears in a 1985 working paper [12] but with a 1985 lower bound.

There are a number of algebraic laws relating the operators introduced so far. The seven which follow are all easily proved from the definitions above:

(L1) $f * g * S = (f \circ g) * S,$

(L2) $P : g * S = g * (P \circ g) : S,$

(L3) $f\!\downarrow g * S = g * (f \circ g)\!\downarrow S,$

(L4) $f * A \cup B = (f * A) \cup (f * B),$

(L5) $P : A \cup B = (P : A) \cup (P : B),$

(L6) $f\!\downarrow A \cup B = f\!\downarrow (f\!\downarrow A) \cup (f\!\downarrow B),$

(L7) $f\!\downarrow f\!\downarrow A = f\!\downarrow A.$

Fig. 5. Transformational Programming and the Paragraph Problem [20]

Meertens met twice in person, so "his influence [...] has probably been much less than it otherwise would have been" [46].[10]

Bird used his version of the notation in journal papers published in 1984 [18] and 1986 [20], extracts from which are shown in Figure 4 and Figure 5 respectively. Note that Bird has switched to Meertens' convention of using an asterisk rather than a centred dot for 'map',[11] but still has no general 'reduce' operator. The latter only came with a series of tutorial papers [19, 21, 22], produced in quick succession and with very similar notation, two being lecture notes from Marktoberdorf and one from the University of Texas at Austin Year of Programming; an example, the calculation for the Maximum Segment Sum problem, is shown in Figure 6. Now the centred dot is used for function composition, and juxtaposition (not shown) is used only for function application; moreover, filter (also not shown) is written with a triangle "◁" rather than a colon.

Around the time of Bird's three sets of lecture notes, presumably during one of Bird's presentations at WG2.1, Robert Dewar passed a note to Meertens which has one word on it, "Squigol", making a pun with language names such as Algol, Cobol, and Snobol [47]. The name first appears in the minutes of Meeting #35 in Sausalito in December 1985. However, it has come to be written "Squiggol", perhaps to emphasise that the pronunciation should be ˈskwɪgɒl ("qui") rather than ˈskwaɪgɒl ("quae").

[10] Nevertheless, Wile's 'sectioning' notation (giving a binary operator one of its two arguments, as in the positivity predicate "(> 0)" and the reciprocal function "$(1/)$") was discussed, and it persists today in Haskell.

[11] In fact, Meertens says that he deliberately used a very small asterisk for 'map', looking from a distance or on a poor photocopy like a ragged dot, so as not to have to choose between the two notations.

$$
\begin{array}{lll}
mss & = & \text{definition} \\
 & & \uparrow/ \cdot +/* \cdot segs \\
 & = & \text{definition of } segs \\
 & & \uparrow/ \cdot +/* \cdot +\!\!+/ \cdot tails* \cdot inits \\
 & = & \text{map and reduce promotion} \\
 & & \uparrow/ \cdot (\uparrow/ \cdot +/* \cdot tails)* \cdot inits \\
 & = & \text{Horner's rule with } a \circledast b = (a + b) \uparrow 0 \\
 & & \uparrow/ \cdot \circledast /\!\!/_0 * \cdot inits \\
 & = & \text{accumulation lemma} \\
 & & \uparrow/ \cdot \circledast /\!\!/_0
\end{array}
$$

Fig. 6. Constructive Functional Programming, showing Maximum Segment Sum [22]

5 Generic structures

An important practical concern for a calculus of program transformations is that the body of transformations is not only large enough and sufficiently general to cover lots of applications, but also small enough and sufficiently structured to be easy to navigate. In his preface to the Algorithmics paper [45], Meertens writes:

> *My main worry was the scope of applicability. Would I find that I needed more and more primitive functions and corresponding rules as I did more examples? So I started doing some problems this way. First I found that I indeed had to invent new functions and laws all the time, which was disappointing. I put it down for some time, but took it up again while I was visiting NYU in '82/'83, since it still looked like the most promising line of research. Then I suddenly realized that there was a pattern in the new functions and laws.* [46]

The pattern Meertens noticed is that several of the core datatypes (namely lists, bags, and sets) form a hierarchy of algebraic structures, and many of the core operations (such as maps, filters, and reductions) are homomorphisms from these algebras. Specifically, each of these three datatypes is generated from an empty structure, singleton structures, and a binary combination operation—for example, the empty list, singleton lists, and list concatenation—and differ only in terms of the algebraic laws (associativity, commutativity, idempotence) imposed on the binary operation. Meertens called these 'generic structures' in the Algorithmics paper, as shown in Figure 7.

Meertens used the same names for all three datatypes ("0" for the empty structure, "$\hat{}x$" for a singleton structure containing element x, "+" for the binary operation), disambiguating by context. In contrast, Bird introduced different names for the different datatypes, as shown in Figure 8. One might also impose no laws on the binary operation, yielding a kind of binary tree as a fourth member of the hierarchy, as in the following table:

Let us start with algebraic structures that are about as simple as possible. Using the notation of McCarthy [17], we have

$$S_D = D \oplus S_D \times S_D .$$

This defines a domain of "D-structures", each of which is either an element of the (given) domain D (e.g., numbers, or sequences of characters), or is composed of two other D-structures.

The diligent reader will have noticed an important difference between the structures defined now, and the S-expressions as used for Lisp. The value **nil** is missing. We can introduce it by writing (using "0" instead of "**nil**"):

$$S_D = D \oplus \{0\} \oplus S_D \times S_D .$$

It becomes more interesting if we impose an algebraic law: $s + 0 = 0 + s = s$. This gives about the poorest-but-one possible algebra. Now we have a more dramatic deviation from the S-expressions, for it is certainly not the case that, e.g., $\mathsf{cons}(s, \mathbf{nil}) = s$.

Fig. 7. Generic structures, from the Algorithmics paper [45] (reference [17] in the figure is to McCarthy's 1963 paper "A Basis for a Mathematical Theory of Computation")

type	*empty*	*singleton*	*binary*	*laws*
tree	$\langle\,\rangle$	$\langle\cdot\rangle$	$\perp\!\!\!\perp$	identity
list	$[\,]$	$[\cdot]$	$+\!\!+$	... and associativity
bag	$)\,($	$)\cdot($	$\uplus$	... and commutativity
set	$\{\,\}$	$\{\cdot\}$	$\cup$	... and idempotency

(although there is no consensus on the naming conventions for trees).

Crucially, each datatype is the *free* algebra on the common signature with a given set of equations, generated by a domain of individual elements; that is, there exists a unique homomorphism from the datatype to any other algebra of the same kind. Therefore to define a homomorphic function over one of the datatypes in the hierarchy, it suffices to identify the target algebra. This leads to the canonical definition scheme (see Figure 8)[12], as used for example for defining

[12] Essentially the same canonical scheme is commonly used today in modern functional programming languages like Haskell:

$$\begin{aligned} map\ f\ [\,] &= [\,] \\ map\ f\ (x : xs) &= f\ x : map\ f\ xs \end{aligned}$$

but for the signature of asymmetric 'cons' lists, rather than symmetric 'cat' lists. This again depends on lists being a free algebra, so the equations have a unique solution, namely the function being defined.

In order to specify functions over lists we need one more assumption, namely that $([\alpha], +\!\!+, [\,])$ is the free monoid generated by α under the assignment $[\cdot] : \alpha \to [\alpha]$. This algebraic statement is equivalent to the assertion that for each function $f : \alpha \to \beta$ and associative operator $\oplus : \beta \times \beta \to \beta$, the three equations

$$\begin{array}{lcl} h[\,] & = & id_\oplus \\ h[a] & = & f\ a \\ h(x +\!\!+ y) & = & h\ x \oplus h\ y \end{array}$$

specify a unique function $h : [\alpha] \to \beta$. In the case that $id_\oplus$ is not defined, the last two equations by themselves determine a uuique fnnction $h : [\alpha]^+ \to \beta$.

A similar algebraic statement about freeness holds for bags and sets as well as lists. We assume that $(\rbag\alpha\lbag, \uplus, \rbag\ \lbag)$ is the free commutative monoid generated by α under the assignment $\rbag\cdot\lbag : \alpha \to \rbag\alpha\lbag$. Similarly, $(\{\alpha\}, \cup, \{\,\})$ is the free commutative and idempotent monoid generated by α under the assignment $\{\cdot\} : \alpha \to \{\alpha\}$. In the case of bags this means that for each $f : \alpha \to \beta$ and associative and commutative operator $\oplus : \beta \times \beta \to \beta$, the equations

$$\begin{array}{lcl} h\rbag\ \lbag & = & id_\oplus \\ h\rbag a\lbag & = & f\ a \\ h(x \uplus y) & = & h\ x \oplus h\ y \end{array}$$

define a unique function $h : \rbag\alpha\lbag \to \beta$. Similar remarks apply to sets, except that we also require $\oplus$ to be idempotent.

Fig. 8. Generic structures, from "Constructive Functional Programming" [22]

maps:

$$\begin{array}{lcl} f*[\,] & = & [\,] \\ f*[a] & = & [f\ a] \\ f*(x +\!\!+ y) & = & f*x\ +\!\!+\ f*y \end{array}$$

filters:

$$\begin{array}{lcll} p \lhd [\,] & = & [\,] & \\ p \lhd [a] & = & [a], & \text{if } p\ a \\ & = & [\,], & \text{otherwise} \\ p \lhd (x +\!\!+ y) & = & p \lhd x\ +\!\!+\ p \lhd y & \end{array}$$

and reductions:

$$\begin{array}{lcl} \oplus/[\,] & = & 1_\oplus \\ \oplus/[a] & = & a \\ \oplus/(x +\!\!+ y) & = & \oplus/x\ \oplus\ \oplus/y \end{array}$$

This hierarchy of datatypes has become known as the 'Boom Hierarchy'—a neat pun. The hierarchy was introduced by and named after Hendrik Boom [26]; but Hendrik Boom is Dutch, and 'boom' is also Dutch for 'tree'. Stephen Spackman was a local observer at Meeting #37 hosted by Boom in Montreal in May

1987, and gave a presentation [61] involving the Boom Hierarchy. Spackman was studying for a Master's degree at Concordia University at the time, supervised by Boom and by Peter Grogono. Spackman recalls:

> *My recollection of how the name came about is that it was Peter Grogono's coinage, that Hendrik instantly said, "what, because it's about trees?", that I laughed, and the name stuck from that moment. My contribution was the appreciation of the joke, not the naming!* [60]

Backhouse [3] presents a detailed study of the Boom Hierarchy, and a comparison to the quantifier notation introduced by Edsger Dijkstra and colleagues at Eindhoven. Like Meertens and unlike Bird, Backhouse uses a common naming scheme for all members of the Hierarchy, albeit a different one from Meertens': "$1_{\#}$", "τ", and "$\#$".

6 Retrenchment

The concern about whether or not to use a single notation for all the members of the Boom Hierarchy gets to a key issue: a novel, rationalized notation can help to reduce the number of definitions and laws and organize the theory, but by disregarding mathematical convention it can make the presentation less accessible to outsiders. In his preface to the 1984 working paper [44], Meertens recalls:

> *You can perhaps imagine my disappointment when I heard from Richard that he had dropped this whole approach because he found it was generally ununderstandable to audiences. Subsequent presentations of the Algorithmics paper at WG2.1 meetings strongly suggested the same to me.* [46]

But the convenience of a rational notation is seductive. In the preface to the 1985 working paper [12] (which was written jointly with Bird and Wile), Meertens continues the story:

> *Somehow or other Richard picked up interest in my 'squiggles' again (really his, if he had not disowned them). It cannot have been the general acclaim they met with at my presentations that made him do so. Maybe it was the ease with which I kept pulling functions and operators to the left or pushing them to the right (while writing the formulas upside-down) over a beer, even after many beers, that convinced him of the continued value of this approach.* [46]

Similar concerns apply more widely to the choice of notation. The 1985 working paper itself reports a difference of opinion with Wile:

> *Whereas RB and LM feel that the predefined infix operators should preferably be single symbols, DW prefers longer operator names. Moreover, LM does not like predefined names that are English words.* [12]

$$
\begin{array}{ll}
\mathsf{mss} = & \text{definition} \\
& \quad \mathsf{max} \cdot \mathsf{map\,sum} \cdot \mathsf{segs} \\
= & \text{definition of segs} \\
& \quad \mathsf{max} \cdot \mathsf{map\,sum} \cdot \mathsf{concat} \cdot \mathsf{map\,tails} \cdot \mathsf{inits} \\
= & \text{map promotion (7)} \\
& \quad \mathsf{max} \cdot \mathsf{concat} \cdot \mathsf{map}(\mathsf{map\,sum}) \cdot \mathsf{map\,tails} \cdot \mathsf{inits} \\
= & \text{definition of max and fold promotion (8)} \\
& \quad \mathsf{max} \cdot \mathsf{map\,max} \cdot \mathsf{map}\,(\mathsf{map\,sum}) \cdot \mathsf{map\,tails} \cdot \mathsf{inits} \\
= & \text{map distributivity (3)} \\
& \quad \mathsf{max} \cdot \mathsf{map}\,(\mathsf{max} \cdot \mathsf{map\,sum} \cdot \mathsf{tails}) \cdot \mathsf{inits} \\
= & \text{Horner's rule, with } x \otimes y = (x+y)\uparrow 0 \\
& \quad \mathsf{max} \cdot \mathsf{map}\,(\mathsf{foldl}\,(\otimes)\,0) \cdot \mathsf{inits} \\
= & \text{scan lemma (12)} \\
& \quad \mathsf{max} \cdot \mathsf{scanl}\,(\otimes)\,0 \\
= & \text{fold-scan fusion (13)} \\
& \quad \mathsf{fst} \cdot \mathsf{foldl}\,(\odot)\,(0,0) \\
\text{where } \odot & \text{ is defined by} \\
& \quad (u,v) \odot x = (u\uparrow w, w) \quad \text{where} \quad w = (v+x)\uparrow 0
\end{array}
$$

Fig. 9. Algebraic Identities for Program Calculation, and Maximum Segment Sum [23]

In a journal paper published in 1989 [23], Bird revisited the Maximum Segment Sum problem he had tackled in earlier Marktoberdorf lectures [22]. But he abandoned the squiggles and reverted to mostly alphabetic identifiers, perhaps under pressure from the journal editor; compare the development in Figure 9 with the earlier one shown in Figure 6. Bird wrote in the paper:

> *In order to make the material as accessible as possible, we shall use the notation for functional programming described by Bird and Wadler. This is very similar to that used in Miranda. (Our preferred notation* [19] *is rather different. For a start, it is more concise and mathematical* [...]*)* [23]

7 The Book on Algorithmics

A recurring theme in the Abstracto papers is the idea of an imagined textbook on algorithmics:

> *Suppose a textbook has to be written for an advanced course in algorithmics. Which vehicle should be chosen to express the algorithms? Clearly, one has the freedom to construct a new language, not only without the restraint of efficiency considerations, but without any considerations of implementability whatsoever.* [33]

The textbook theme is frequently mentioned in the minutes of discussions at WG2.1 meetings around this time, and becomes a central desideratum for Squiggol. It is alluded to in the title "Two Exercises Found in a Book on Algorithmics"

Theorem 7.2 If S is monotonic on a preorder R°, then

$$(\!|min\, R \cdot \Lambda S|\!) \quad \subseteq \quad min\, R \cdot \Lambda((\!|S|\!)).$$

Proof. We reason:

$$\begin{array}{ll}
 & (\!|min\, R \cdot \Lambda S|\!) \subseteq min\, R \cdot \Lambda((\!|S|\!)) \\
\equiv & \quad \{\text{universal property of } min\} \\
 & (\!|min\, R \cdot \Lambda S|\!) \subseteq (\!|S|\!) \quad \text{and} \quad (\!|min\, R \cdot \Lambda S|\!) \cdot (\!|S|\!)^\circ \subseteq R \\
\equiv & \quad \{\text{since } min\, R \cdot \Lambda S \subseteq S\} \\
 & (\!|min\, R \cdot \Lambda S|\!) \cdot (\!|S|\!)^\circ \subseteq R \\
\Leftarrow & \quad \{\text{hylomorphism theorem (see below)}\} \\
 & min\, R \cdot \Lambda S \cdot \mathsf{F}R \cdot S^\circ \subseteq R \\
\Leftarrow & \quad \{\text{monotonicity: } \mathsf{F}R \cdot S^\circ \subseteq S^\circ \cdot R\} \\
 & min\, R \cdot \Lambda S \cdot S^\circ \cdot R \subseteq R \\
\Leftarrow & \quad \{\text{since } min\, R \cdot \Lambda S \subseteq R/S^\circ\text{; division}\} \\
 & R \cdot R \subseteq R \\
\equiv & \quad \{\text{transitivity of } R\} \\
 & true.
\end{array}$$

Fig. 10. The Algebra of Programming [10]

of a short paper by Bird and Meertens [25], another paper with a long gestation period (discussed at Meeting #34 in Utrecht in April 1985, presented at the TC2 Working Conference on Program Specification and Transformation in Bad Tölz in April 1986, and eventually published in 1987).

About a decade later, Bird published the book "The Algebra of Programming" [10] together with Oege de Moor. This book develops general theorems and specific constructions for solutions to optimization problems: greedy algorithms, dynamic programming, and so on. Bird and de Moor discovered that this class of problem really calls for a calculus of *relations* rather than one of functions, because many problems are most naturally expressed in terms of converses, intersections, orderings, and other notions that are awkward to handle using pure functions alone. Moreover, in the quest for crisp statements of general results, the book followed Grant Malcolm's lead [39, 40] in bringing in ideas from category theory such as functors, natural transformations, and initial algebras. Thus, it has more squiggles, and different ones, such as superscript circles for converses, inclusions, relational divisions (a kind of weakest prespecification), as shown in Figure 10.

The Algebra of Programming book is many things: a work of art, and a tour de force, and perhaps even a coup de grâce for Squiggol. But one thing it is not: an easy read. The relational algebra is very elegant, and unquestionably the idealist's tool for this class of problems; but it is inherently complicated, because there are simply a lot of laws to remember.

Bird envisioned this work as fulfilling the promise of the legendary textbook on algorithmics [9], although the book does not actually present itself that way. In fact, it follows closely the approach taken by de Moor in his doctoral thesis [29], which itself drew on Freyd and Scedrov's work on allegories as a categorical axiomatization of relations [32], and Bird now says that "it turned out very different to the book I had envisaged" [9].

8 Conclusions

The story of Squiggol is one of an ebb and flow of enthusiasm for the squiggly notation. The notation is intended as a *tool of thought* more than a programming language; so there is the freedom to experiment, to invent new operators, to capture newly-observed recurring patterns, unfettered by the need to keep all the paraphernalia of an automated tool chain up to date. But that freedom is a mixed blessing, and it is all too easy to disappear down a rabbit-hole of private scribbling; the notation should also be a *tool of communication*—with other people, and even with one's future self—and undisciplined invention blocks that communication channel.

The supplementary website [11] for the Algebra of Programming book describes it as an "introductory textbook", which is rather optimistic: few people have read the book all the way through, and fewer still have assimilated and can remember all the laws it presents. With a few honourable exceptions, almost everybody who was involved has abandoned the relational squiggles. De Moor soon left this field and moved into work on programming tools, eventually leaving academia to found the company Semmle. Bird also quickly gave up on the squiggly notation, succumbing to his 1989 critics [23] and doing almost everything since the book in a purely functional (Haskell) notation.

Bird and the present author are putting the finishing touches to a new book "Algorithm Design with Haskell" [24], addressing essentially the same material as the Algebra of Programming book with no squiggles at all. This latest approach definitely represents a compromise: a small excursion out of the world of pure functions is required in order to accommodate nondeterministic choice [13]. Only time will tell whether the balance is better this time, with greater accessibility compensating for the loss of expressive power.

Acknowledgements

I am especially indebted to Richard Bird and Lambert Meertens: for much discussion about this work, and correcting some of my misunderstandings, but more importantly for being the inspiration for essentially my entire research career. I would also like to thank Doaitse Swierstra, Hendrik Boom, and Stephen Spackman for answering my many questions, and Cezar Ionescu, Dan Shiebler, the members of IFIP WG2.1, the participants at the workshop on the History of Formal Methods in Porto in October 2019, and the anonymous reviewers for their helpful comments and enthusiastic feedback.

References

1. Ralph-Johan Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.
2. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.
3. Roland Backhouse. An exploration of the Bird-Meertens formalism. In *International Summer School on Constructive Algorithmics, Hollum, Ameland*. STOP project, 1989. Also available as Technical Report CS 8810, Department of Computer Science, Groningen University, 1988.
4. John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy, Peter Naur, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Report on the algorithmic language ALGOL 60. *Numerische Mathematik*, 2(1):106–136, 1960.
5. David W. Barron. Christopher Strachey: A personal reminiscence. *Computer Bulletin*, 2(5):8–9, 1975.
6. Friedrich L. Bauer. Programming as an evolutionary process. In *International Conference on Software Engineering*, pages 223–234. IEEE, 1976.
7. Friedrich L. Bauer, Rudolf Berghammer, Manfred Broy, Walter Dosch, Franz Geiselbrechtinger, Rupert Gnatz, Erich Hangel, Wolfgang Hesse, Bernd Krieg-Brückner, Alfred Laut, Thomas Matzner, Bernhard Möller, Friederike Nickl, Helmuth Partsch, Peter Pepper, Klaus Samelson, Martin Wirsing, and Hans Wössner. *The Munich Project CIP, Volume 1: The Wide-Spectrum Language CIP-L*, volume 183 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1985.
8. Friedrich L. Bauer, Herbert Ehler, Alexander Horsch, Bernhard Möller, Helmuth Partsch, Otto Paukner, and Peter Pepper. *The Munich Project CIP, Volume 2: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.
9. Richard Bird. "Algebra of Programming" as the textbook on algorithmics. Private email to JG, February 2020.
10. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, 1997.
11. Richard Bird and Oege de Moor. Website for The Algebra of Programming. `http://www.cs.ox.ac.uk/publications/books/algebra/`, 1997.
12. Richard Bird, Lambert Meertens, and David Wile. A common basis for algorithmic specification and development. IFIP WG2.1 Working Paper ARK-3, 1985.
13. Richard Bird and Florian Rabe. How to calculate with nondeterministic functions. In Graham Hutton, editor, *Mathematics of Program Construction*, volume 11825 of *Lecture Notes in Computer Science*, pages 138–154. Springer-Verlag, 2019.
14. Richard S. Bird. Improving programs by the introduction of recursion. *Communications of the ACM*, 20(11):856–863, November 1977.
15. Richard S. Bird. Notes on recursion elimination. *Communications of the ACM*, 20(6):434–439, 1977.
16. Richard S. Bird. Notational suggestions for transformational programming. Technical Report RCS 144, University of Reading, April 1981.
17. Richard S. Bird. Some notational suggestions for transformational programming. Working Paper NIJ-3, IFIP WG2.1, 1981.
18. Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984.

19. Richard S. Bird. An introduction to the theory of lists. Monograph PRG-56, Programming Research Group, University of Oxford, October 1986. Published in [27].
20. Richard S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6:159–189, 1986.
21. Richard S. Bird. A calculus of functions for program derivation. Monograph PRG-64, Programming Research Group, University of Oxford, December 1987. Published in [62].
22. Richard S. Bird. Lectures on constructive functional programming. Monograph PRG-69, Programming Research Group, University of Oxford, September 1988. Published in [28].
23. Richard S. Bird. Algebraic identities for program calculation. *Computer Journal*, 32(2):122–126, April 1989.
24. Richard S. Bird and Jeremy Gibbons. *Algorithm Design with Haskell*. Cambridge University Press, 2020. To appear.
25. Richard S. Bird and Lambert Meertens. Two exercises found in a book on algorithmics. In Lambert Meertens, editor, *Program Specification and Transformation*, pages 451–457. North-Holland, 1987.
26. Hendrik Boom. Further thoughts on Abstracto. Working Paper ELC-9, IFIP WG2.1, 1981.
27. Manfred Broy, editor. *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987. NATO ASI Series F Volume 36.
28. Manfred Broy, editor. *Constructive Methods in Computer Science*. Springer-Verlag, 1988. NATO ASI Series F Volume 55.
29. Oege de Moor. *Categories, Relations and Dynamic Programming*. PhD thesis, Programming Research Group, Oxford, April 1992. Available as Technical Monograph PRG-98.
30. Robert Dewar. Letter to members of IFIP WG2.1. `http://ershov-arc.iis.nsk.su/archive/eaindex.asp?did=29067`, 26 July 1977.
31. Robert Dewar. Letter to members of IFIP WG2.1. `http://ershov-arc.iis.nsk.su/archive/eaindex.asp?did=29096`, 19 September 1979.
32. Peter Freyd and Andre Scedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.
33. Leo Geurts and Lambert Meertens. Remarks on Abstracto. *Algol Bulletin*, 42:56–63, 1978. Also in [46].
34. Leo Geurts, Lambert Meertens, and Steven Pemberton. *The ABC Programmer's Handbook*. Prentice-Hall, 1990. ISBN 0-13-000027-2.
35. Jeremy Gibbons. *Algebras for Tree Algorithms*. D. Phil. thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94. ISBN 0-902928-72-4.
36. Kenneth E. Iverson. *A Programming Language*. John Wiley, 1962.
37. Cornelius H. A. Koster. The making of Algol 68. In Dines Bjørner, Manfred Broy, and Igor Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1996.
38. Charles H. Lindsey. A history of Algol 68. In *HOPL-II: The Second ACM SIGPLAN Conference on History of Programming Languages*, pages 97–132, April 1993.
39. Grant Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.
40. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

41. Lambert Meertens. Abstracto 84: The next generation. In *Proceedings of the 1979 Annual Conference*, pages 33–39. ACM, 1979.
42. Lambert Meertens. Draft proposal for the B programming language. Technical report, Mathematisch Centrum, Amsterdam, 1981.
43. Lambert Meertens. Issues in the design of a beginners' programming language. In J. W. de Bakker and H. van Vliet, editors, *Algorithmic Languages*, pages 167–184, New York, July 1981. Elsevier North-Holland.
44. Lambert Meertens. Some more examples of algorithmic developments. IFIP WG2.1 Working Paper ADP-7, 1984.
45. Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In J. W. de Bakker, M. Hazewinkel, and J. K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986. Available at `https://ir.cwi.nl/pub/20634`.
46. Lambert Meertens. An Abstracto reader prepared for IFIP WG 2.1. Technical Report CS–N8702, CWI, Amsterdam, April 1987.
47. Lambert Meertens. Squiggol versus Squigol. Private email to JG, September 2019.
48. Lambert G. L. T. Meertens and Steven Pemberton. Description of B. *SIGPLAN Notices*, 20(2):58–76, 1985.
49. Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–419, 1988.
50. Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.
51. Robert Paige. Transformational programming: Applications to algorithms and systems. In John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum, editors, *Principles of Programming Languages*, pages 73–87. ACM, 1983.
52. Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.
53. John E. L. Peck. Aad van Wijngaarden and the Mathematisch Centrum: A personal recollection. In Gerard Alberts, editor, *Conference on the History of ALGOL 68*, volume AM-HN9301. CWI, Amsterdam, January 1993.
54. Steven Pemberton. A short introduction to the ABC language. *SIGPLAN Notices*, 26(2):11–16, 1991.
55. Python Software Foundation. Python website. `https://www.python.org/`, 1997.
56. Stephen A. Schuman, editor. *New Directions in Algorithmic Languages*. Prepared for IFIP Working Group 2.1 on Algol, Institut de Recherche d'Informatique et d'Automatique, 1975.
57. Stephen A. Schuman, editor. *New Directions in Algorithmic Languages*. Prepared for IFIP Working Group 2.1 on Algol, Institut de Recherche d'Informatique et d'Automatique, 1976.
58. Jacob T. Schwartz. On programming: An interim report on the SETL project. Technical report, New York University, 1974.
59. Jacob T. Schwartz, Robert B. K. Dewar, Ed Dubinsky, and Edmond Schoenberg. *Programming with Sets: An Introduction to SETL*. Texts and Monographs in Computer Science. Springer, 1986.
60. Stephen Spackman. Boom and Abstracto. Private email to JG, October 2019.
61. Stephen Spackman and Hendrik Boom. Foop, Poof, and parsing. Working Paper 560 COR-15, IFIP WG2.1, 1987.
62. David A. Turner, editor. *Research Topics in Functional Programming*. University of Texas at Austin, Addison-Wesley, 1990.

63. Adriaan van Wijngaarden, Barry J. Mailloux, John E. L. Peck, and Cornelius H. A. Koster. Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, 14(2):79–218, 1969.
64. Adriaan van Wijngaarden, Barry J. Mailloux, John E. L. Peck, Cornelius H. A. Koster, Michel Sintzoff, Charles H. Lindsey, Lambert G. L. T. Meertens, and Richard G. Fisker. Revised report on the algorithmic language Algol 68. *Acta Informatica*, 5(1–3):1–236, 1975. Also appeared as Mathematical Centre Tract 50, CWI, Amsterdam, and published by Springer Verlag in 1976.
65. David S. Wile. *A Generative, Nested-Sequential Basis for General Purpose Programming Languages*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1973.