stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 181/81 NOVEMBER

A. VAN WIJNGAARDEN

LANGUAGELESS PROGRAMMING

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

ACM-Computing Reviews-category: 5.23, 5.26, 4.20

LANGUAGELESS PROGRAMMING*

by

A. van Wijngaarden

ABSTRACT

This paper demonstrates that it is readily possible to express an algorithm through a two-level grammar, with an ease comparable to but under avoidance of an intervening programming language.

KEY WORDS & PHRASES: Two-level grammars, programming languages

* This paper is not for review; it will be published elsewhere.

1. INTRODUCTION

If a programmer wants to specify a process to be performed on data then (s)he writes a particular program in some programming language, e.g., ALGOL 68. The particular program might be:

(INT n; read(n); print(("p(", whole(n,0), ")_:_-_:_25_:_=_:_", whole(prime(n)-25,0))))

where the marks are representations of the ALGOL 68 terminal symbols.

This particular program does not completely specify the process to be performed for two reasons. First, RR68, i.e., the definition of ALGOL 68 [2] does not provide a declaration for *prime*. This declaration might however be given in a particular prelude. Let us assume that this is the case and that the call *prime(n)* delivers what the wording suggests, viz., the *n*-th prime number. The second reason is that the call *read(n)* might assign a nonpositive value to *n*. Thus we must assume that *prime* can cope with this situation in some way or another, or, otherwise, the programmer should correct the particular program, e.g., by writing *read(n); n:=ABSn+1* instead of *read(n)*. We shall assume the first possibility.

This particular program may be elaborated by a computer. Let us assume that *read(n)* has the same effect as *n:=125* would have had. Since the 125-rd prime number is, as well known, 691 the result would be

p (1 2 5) - 2 5 = 6 6 6

where the marks are from a totally different vocabulary, only partially defined by RR68.

The definition of the programming language, here ALGOL 68, obviously also belongs to the specification of the process because, otherwise, the meaning of the program would be undefined. This definition is given in RR68 partly in the English language and partly by means of a specific two-level grammar with production rules like

program : strong void new closed clause.

where the marks are again from a totally different vocabulary.

The definition of the concept "two-level grammar" obviously also belongs to the specification of the process because, otherwise, the meaning of the specific two-level grammar of the language would be undefined. This definition was first given in [1] in a loose manner and later more elaborately in [2]. A formal operational definition was given in [3] and is included here in section 2.

There are therefore, apart from the definition in the English language, at least four levels of definition involved in the specification, viz., the definition of the two-level-grammar concept by the tool maker; the specific two-level grammar of a language by the language designer; the particular program in that language by the programmer; the output of that particular program by the computer.

This is a considerable hierarchy and, moreover, a voluminous one: RR68 contains more than two hundred pages. The reward is only that ALGOL 68 programs can be elaborated by actual present-day computers but still specification is poor; in effect the (truth) value of $0<0$ is not better defined by RR68 than by the loose remark 'usual mathematical meaning' and the (truth) value of $0-0=0$ is not defined at all by RR68.

The reason for this deplorable situation is that we have on one hand the extreme definitional power of two-level grammars and on the other hand a modest process that has to be defined. The insertion between these two of a large programming language not specifically adapted to that process is — apart from that elaboration by an actual present-day computer — like the use of an elephant to carry groceries home from the supermarket; the elephant may eat part of them.

In this paper it is shown how the language can be eliminated on the hand of a specific example. In fact, a set of metarules and a set of hyperrules is given whose power exceeds by far this ex-

ample and only the choice of a specific hyperrule for the startword restricts it to this example. In some sense these sets of metarules and hyperrules may be regarded as a language but there is an essential difference: the "grammarian", i.e., the person who specifies a process by means of a two-level grammar, never writes down a representation of a terminal letter; on the contrary, these representations are equivalent with the marks that are written down by the computer introduced above. The language is thereby completely eliminated.

The critical remarks on ALGOL 68 should not be misinterpreted; remarks on other languages might very well have been much more critical indeed.

2. TWO-LEVEL GRAMMARS

A "vocabulary" is a set; its elements are termed "letters". A "word" over a vocabulary V is a mapping $[1:n]\rightarrow V$, for any $n\in\mathbb{N}_0$, and is thus a set of n ordered pairs (i,v_i), for $i=1,\cdots,n,v_i\in V$. Therefore, v_i is termed the "i-th letter" and n the "length" of the word. If $n=0$, then the word is the empty set, also termed the "empty word". For each vocabulary V, V^* denotes the set of words over V, and V^+ the set of nonempty words over V. A "sentence" over a vocabulary V is a word over the vocabulary whose letters are the words over V; hence V^{**} is the set of sentences over V.

A "rule" is an ordered pair (v,w) where v and w are words over certain vocabularies.

A "two-level grammar" VWG is an ordered sextuple $(V_m,V_o,V_t,R_m,R_h,w_s)$, where V_m, V_o, V_t are finite vocabularies, whose letters are termed "metaletters", "ortholetters" and "terminal letters" respectively, R_m and R_h are finite sets of rules, termed "metarules" and "hyperrules" respectively, and w_s is some word over V_o, termed the "startword". Let $V_h:=V_m\cup V_o$. It is required that $V_m\cap V_o=\{\}$, $V_t\subset V_o^+$, $R_m\subset V_m\times V_h^*$, $R_h\subset V_h^+\times V_h^{**}$, $w_s\in V_o^+$.

The grammar VWG "produces" a sentence set L defined as follows:

Let $R_{mo}:=V_m\times V_o^*$, $R_{oo}:=V_o^+\times V_o^{**}$ and $R_{st}:=\{w_s\}\times V_t^*$. A set R_m' identical with R_m and a set R_h' identical with R_h are introduced and then extended by applying, arbitrarily often, if possible, the following extension, where at each application, out of the three alternatives separated by ',' and enclosed by '( )', consistently either the first, or the second, or the third must be chosen:

Extension: To (R_m',R_h',R_h') a rule is added, obtained by replacing, in a copy of some rule $(v,w)\in(R_m',R_h',R_h')$ and for some rule $(v',w')\in(R_m'\cap R_{mo},R_m'\cap R_{mo},R_h'\cap R_{oo})$, (some, each, some) occurrence of v' in $(w,v$ and $w,w)$ by w'.

Then, $L:=\{w\mid(w_s,w)\in R_h'\cap R_{st}\}$.

The set of "terminal metaproductions" of a metaletter $\boldsymbol{M}$ is the set $\{w\mid(\boldsymbol{M},w)\in R_m'\cap R_{mo}\}$.

In this paper a metaletter is a conventional 'capital letter' possibly followed by one or more times '" (apostrophe); an ortholetter is a conventional 'lower case letter', 'decimal digit', 'opening parenthesis' or 'closing parenthesis'. Four other letters play a role, viz., ':' (colon), '.' (point), ',' (comma) and ';' (semicolon).

Letters are "written" one after the other in such a way that the order in which they have been written is clear, in this paper conventionally to the 'right of' the letter lastly written before, or 'on the next line' or 'on the next page', whatever this may mean.

A word is written when its writing starts by writing its first letter, if it exists, and, when its i-th letter has been written by writing its $(i+1)$-th letter, if it exists.

A metarule (v,w) is written by writing v, then writing twice a colon, then writing w and then writing a point.

A hyperrule (v,w) is written by writing v, then writing a colon, then writing w and then writing a point. Writing w, however, poses the problem that a sentence over V_o might later on be

misread. In natural languages this problem is overcome by separating the words of the sentence by blanks. Here, traditionally, one separates the words by writing a comma after a word has been written and before the next word of the sentence is going to be written, which leaves open the use of blanks for display purposes inside the words.

The grammar mechanism as defined so far is, however, not yet complete. The terminal letters are words over V_o but this internal structure is of no relevance to the user. Therefore, V_t is mapped onto another vocabulary W_t, the set of "representations" of the terminal letters, which may be marks chosen by the user at his convenience as long as they differ from all letters mentioned above.

A terminal production of a *VWG*, i.e., an element of L, is represented by replacing each terminal letter in a copy of that element by its representation and taking out the comma that follows it, if any.

A useful shorthand notation for rules is the following one: If two rules have the same left-hand side up to and including the colon or double colon, then they may be combined into one rule consisting of the first rule in which the point has been replaced by a semicolon, followed by the right-hand side of the second rule.

Thus **H :: 9 ; 8 ; 7.** stands for **H :: 9. H :: 8. H :: 7.**.

Another useful convention stems from the fact that one frequently needs metarules differing only in the left-hand side, because one wants to circumvent the effect of the, utterly necessary, word 'each' in the Extension. Therefore, by convention, it holds:

Let ***M*** stand for any element of V_m. Then any occurrence of ***M'*** in a *VWG* tacitly implies that ***M'* :: *M*.** is an element of R_m.

Thus, the occurrence of V″ implies the metarule **V″ :: V′.**, which implies the metarule **V′ :: V.**, so that V, V′ and V″ have the same terminal productions over V_o.

3. AN EXAMPLE OF A TWO-LEVEL GRAMMAR FOR LANGUAGELESS PROGRAMMING

The feasibility of languageless programming is shown by the following two-level grammar.

The metaletters are in first instance **H**, **J**, **K**, **L**, **M**, **N**, **P**, **Q**, **R**, **V**, **W** and, moreover, if ***M*** stands for any metaletter a metaletter ***M'*** may be added.

The ortholetters are **n**, **p**, **s**, **9**, **8**, **7**, **6**, **5**, **4**, **3**, **2**, **1**, **0**, **(**, **)**.

The metarules are in first instance

M1 **H :: 9 ; 8 ; 7 ; 6 ; 5 ; 4 ; 3 ; 2 ; 1 ; 0.**
M2 **J :: HJ ; .**
M3 **K :: n ; p.**
M4 **N :: nN ; .**
M5 **P :: pP ; .**
M6 **M :: nN ; pP ; .**
M7 **Q :: HQ ; KQ ; .**
M8 **V :: sM.**
M9 **W :: sQ ; (R).**
M10 **L :: VL ; .**
M11 **R :: WR ; .**

The terminal letters are terminal metaproductions of **V**.

The startword is **s0**.

The hyperrules are

H1 **L(R′)R : LR′R.**
H2 **s0 : sp25sn8spPs1ns10sn9sn2sn41sn2sp25s1ns10sn2sn36sn2spPs50sp25s41s1ns10sps64ps2.**

H3 **Ls00R : .**
H4 **LsKHJR : Lss0KHJR.**
H5 **LVs0KHJR : LVsppppppppppps42s00KH0123456789s40s0KJR.**
H6 **LVs0KR : LVR.**
H7 **LsM00KHH'JR : LsKM00KHJR.**
H8 **LsM00KHHJR : LsMR.**
H9 **LVs1MR : LVss24s4sn41sMVs03ps9s01s01R.**
H10 **Ls01PsP'L's01NR : LsP'sp10s34s4(sP'sp10s41s9(s01pP)L's01N)**
(sPss22s4(s01sPsP'L's01nN)(sP'L'snN))R.
H11 **LVs2R : V , LR.**
H12 **LsKMs3M'R : LsMsKs4s3nM's3pM'R.**
H13 **Lss3M'R : LsM'R.**
H14 **LVs03KR : LVVs4( )s3sKs4( )s3R.**
H15 **LsPs4WW'R : LWR.**
H16 **LsNs4WW'R : LW'R.**
H17 **LVs5(WW'R')R : LVss32s4W(Vsps41s5(W'R'))R.**
H18 **LVs5(W)R : LWR.**
H19 **Ls6pPVR : LVs6PR.**
H20 **Ls6pPs6HMWR : LR.**
H21 **LVV's6nNR : LVs6NV'R.**
H22 **Vs6nNLs6HMWR : VLR.**
H23 **Ls7WW'R : LWs4(W's7WW')( )R.**
H24 **LVV'V''s8WW'R : LV'ss36s4sp(VV''V's4$32s34)VV'V''s8pWW'R.**
H25 **LsMVV'V''s8KWW'R : LsMs4(sKs4(WVV'V''s8n)(W'VV's40V'V''s8)WW')sKR.**
H26 **LVs9WR : LWVR.**
H27 **LVs10R : LR.**
H28 **LVs11R : LVVR.**
H29 **LsMs12WR : LWMR.**
H30 **LVs13R : LVs4spsnR.**
H31 **LVV's20R : LVV's21s11s3s21R.**
H32 **LVV's21R : LVV's24s4VV'R.**
H33 **LVV's22R : LVV's41s022R.**
H34 **LspPs022R : LspPR.**
H35 **LsNs022R : LsnNR.**
H36 **LVV's23R : LVR.**
H37 **LVV's24R : LV'Vs22R.**
H38 **LVV's25R : LV'R.**
H39 **LVV's26R : LVV's22VV's24s27R.**
H40 **LVV's27R : LVV's22s4VV'R.**
H41 **LVV's3HR : LVV's2Hs3R.**
H42 **LsMsM's40R : LsMM'R.**
H43 **LsnNpPR : LsNPR.**
H44 **LspPnNR : LsPNR.**
H45 **LVV's41R : LVV's3s40R.**
H46 **LVV's42R : LV'ss36s4s(V's4(VV'sps41s42Vs40)(VV's3s42s3))R.**
H47 **LVV's43R : LsVs03pV's03ps043Vs4(V's4( )(sns64s3))(V's4(sns64ps3)s3)R.**
H48 **LsP''sPspP's043R : LsPspP's34s4(spP''sPspP's41spP's043)(sP''sP)R.**
H49 **LVsPs44R : LsPss36s4sp(VsPsps41s44Vs42)R.**
H50 **LVV'spPs45R : LVV'V's40s03pspPs22s4(V's13s40)( )R.**
H51 **LVV's40pPR : LVV's40R.**

H52 **LVV's41pPR : LVV's41R.**
H53 **LVV's42pPR : LVV's42spPs60s43spPs60s45R.**
H54 **LVV's43pPR : LVspPs60s42V's43spPs60s45R.**
H55 **LVV's44pPR : LV'ss36s4(spPs60)(V's4(VV'sps41s44pPVs42pP) (spPs60VV's3s44pPs43pP))R.**
H56 **LspPs50R : LspPsppspps050R.**
H57 **LspPsppP'sppP''s050R : LsppP'sppP''s43sns64s10ss22 s4(spPsppP'spppP''s050) (sppP'sppP''s22s4(spPspppP'spps050)(spPsps22s4(sPspppP'spps050)sppP'))R.**
H58 **LsM's6HMWR : sM'ss22s4(s6M'L)(Ls6M')s6HMWR.**
H59 **LVs6L's60R : LVL'VR.**
H60 **LVs6L'V's61R : LV'L'R.**
H61 **Ls6L'V's62R : LV'L'R.**
H62 **Ls6L's63KR : LsKs4L'LL'R.**
H63 **LVs6L's64MWR : LVWsMss36s4(L')(s6ML's64MW)R.**

4. INTERPRETATIONS

A two-level grammar defines a formal play with letters without the need of an interpretation. We might, therefore, leave the reader with the lists of metarules and hyperrules without comment. However, he may wonder what they are good for and he may find it hard to understand their effect. Therefore, we shall help him with some "interpretations", in this section, and "explanations", in section 5, but he should keep in mind that these interpretations and explanations have no defining power, they are just comment.

The terminal productions of **W** are termed "syllables". A syllable is either a "value" or an "operator". Values are the terminal productions of **V**. They are therefore

... , **snnn** , **snn** , **sn** , **s** , **sp** , **spp** , **sppp** , ...

and operators are therefore, e.g.,

s23 , **s3n** , **sp25** , **spppnn** , **(sns64s10)** .

Each, possibly empty, sequence of syllables is a terminal production of **R** (for right) and, more specifically, each, possibly empty, sequence of values is a terminal production of **L** (for left). The idea is that a "word" typically consists of a, possibly empty, sequence of values, followed by an operator, the "first operator", followed by a, possibly empty, sequence of syllables. Hyperrules the define how the word produces another word due to the first operator in its context. The location of the first operator is specified by the "name" *F*; the location of any value to the left of the first operator is specified by a name which "refers to" the value at that location. These names are, counting from the left to the right, *Sp* , *Spp* , *Sppp* , ... , *F* and also, counting from the right to the left, ... , *Snnn* , *Snn* , *Sn* , *S* , *F* . Each value to the left of the first operator is therefore referred to by two names, a "positive" one and a "nonpositive" one. A pair formed by a name and the value to which that name refers is a "variable". A variable with a positive name is a "declared" variable, i.e., a variable whose name does not depend on the location specified by *F*; a variable with a nonpositive name is a "stack" variable, i.e., a variable whose name does depend on that location.

Values may be "interpreted", e.g., in the sense of ALGOL 68, i.e., integers, real numbers, truth values, names, etc. The interpretation of the above sequence of values as integers is

... , -3, -2, -1, 0, 1, 2, 3, ...

The interpretation as real numbers is

... , $-3/B$, $-2/B$, $-1/B$, $0/B$, $1/B$, $2/B$, $3/B$, ...

where *B*, the "base", stands for some integer greater than one.

The interpretation as truth values in a many-valued logic is

... , *fff*, *ff*, *f*, *u*, *t*, *tt*, *ttt*, ...

where *f* suggests FALSE, *t* TRUE and *u* UNDEFINED.

For a given word the interpretation as names is

... , *Snnn* , *Snn* , *Sn* , *S* , *Sp* , *Spp* , *Sppp* , ...

names which have been defined above.

Which of these interpretations is in a given case the most appropriate one depends on the first operator and its context.

The terminal letters of the grammar are certain values if they stand on their own, i.e., not as syllables of a larger word. Strictly speaking, an interpretation of the terminal letters is unnecessary since the grammarian never writes down the representation of the terminal letters, in contrast to the situation that a programmer writes a program in some programming language where he writes only representations of the terminal letters of the grammar defining that language. It is, however, more helpful to provide a list of representations in order to show an interpretation of the terminal letters. In the list we write for the terminal letter consisting of **s** followed by, e.g., 25 times **p** simply **sp25** and for **s** followed by, e.g., 8 times **n** simply **sn8**. This is not so sloppy as it might seem at first sight since, according to the hyperrules H4-H8, the operator **sp25** indeed produces **s** followed by 25 times **p** and the operator **sn8** indeed produces **s** followed by 8 times **n**. The list might then run as follows:

sn42	×	**sn41**	−	**sn40**	+	**sn36**	=	**sn34**	⩾
sn32	⩽	**sn29**	}	**sn28**	{	**sn26**	≠	**sn24**	<
sn22	>	**sn19**	]	**sn18**	[	**sn17**	\	**sn16**	|
sn15	/	**sn13**	←	**sn12**	→	**sn11**	↑	**sn10**	↓
sn9	)	**sn8**	(	**sn7**	:	**sn6**	,	**sn5**	.
sn4	(newpage)			**sn3**	(newline)				
snn	(blank space)			**sn**	(invisible)				
s	*0*	**sp**	*1*	**spp**	*2*	**sp3**	*3*	**sp4**	*4*
sp5	*5*	**sp6**	*6*	**sp7**	*7*	**sp8**	*8*	**sp9**	*9*
sp10	*a*	**sp11**	*b*	**sp12**	*c*	**sp13**	*d*	**sp14**	*e*
sp15	*f*	**sp16**	*g*	**sp17**	*h*	**sp18**	*i*	**sp19**	*j*
sp20	*k*	**sp21**	*l*	**sp22**	*m*	**sp23**	*n*	**sp24**	*o*
sp25	*p*	**sp26**	*q*	**sp27**	*r*	**sp28**	*s*	**sp29**	*t*
sp30	*u*	**sp31**	*v*	**sp32**	*w*	**sp33**	*x*	**sp34**	*y*
sp35	*z*	**sp36**	*A*	**sp37**	*B*	**sp38**	*C*	**sp39**	*D*
sp40	*E*	**sp41**	*F*	**sp42**	*G*	**sp43**	*H*	**sp44**	*I*
sp45	*J*	**sp46**	*K*	**sp47**	*L*	**sp48**	*M*	**sp49**	*N*
sp50	*O*	**sp51**	*P*	**sp52**	*Q*	**sp53**	*R*	**sp54**	*S*
sp55	*T*	**sp56**	*U*	**sp57**	*V*	**sp58**	*W*	**sp59**	*X*
sp60	*Y*	**sp61**	*Z*						

The choice of the representations of **s** up to **sp61** is obvious. The representations of the remaining terminal letters are partially chosen for mnemonic reasons. Since, e.g., **s40** is the operator that performs the addition of two integers, + is chosen as representation of **sn40**.

5. EXPLANATIONS

We now start to explain the effect of the hyperrules for the operators in some detail. We shall not follow the order in which they are given in Section 3 but shall deal with them in such an order that on the right side only operators occur that have been dealt with already or that are under discussion. That this is possible proves at the same time that the grammar does not contain viciously circular definitions.

It often occurs that in a hyperrule **W** occurs whereas in a certain application of that hyperrule we actually like to replace that **W** not by one syllable but by a sequence of syllables, e.g., **sns64s10**. We then replace it instead by a "compound operator", in this case **(sns64s10)**, which is a terminal production of **W**, i.e., a syllable, but not of **V**, i.e., not a value and hence an operator. If this operator becomes the first operator then the general rule

H1 **L(R')R : LR'R.**

decomposes it into a sequence of its constituent syllables, after which the hyperrule for the new first operator takes over control.

In the context **LVs4WW'R** the operator **s4** "chooses" between **W** and **W'** roughly as *IF* **V** *THEN* **W** *ELSE* **W'** *FI* does:

H15 **LsPs4WW'R : LWR.**

H16 **LsNs4WW'R : LW'R.**

Here the obvious interpretation of **V** is a truth value. Since our logic is many-valued, **V** may be **s**, interpreted as UNDEFINED, and then both hyperrules are applicable, leaving it undefined whether **W** or **W'** is chosen.

In the context **LVs3R** the operator **s3** "negates" **V**, i.e., it transforms **s** followed by a number of times **p** resp. **n** into **s** followed by the same number of times **n** resp. **p**. If **V** is interpreted as an integer or a real number then this negation means a change of the sign of **V**, whereas if **V** is interpreted as a truth value then it means logical negation. Its definition is an application of the operator **s4**:

H12 **LsKMs3M'R : LsMsKs4s3nM's3pM'R.**

H13 **Lss3M'R : LsM'R.**

The operator **s3** might also have been defined without using the operator **s4**. This would cost four hyperrules instead of two, and our strategy is always to minimize the number of hyperrules.

In the context **LVs03KR** the operator **s03p** delivers *ABS* **V** and the operator **s03n** delivers *−ABS* **V**:

H14 **LVs03KR : LVVs4( )s3sKs4( )s3R.**

In the context **LVs9WR** the operator **s9** "swaps" **V** and **W**:

H26 **LVs9WR : LWVR.**

In the context **LVs10R** the operator **s10** "voids" **V**:

H27 **LVs10R : LR.**

In the context **LVs11R** the operator **s11** "duplicates" **V**:

H28 **LVs11R : LVVR.**

In the context **LsMs12WR** the operator **s12** "adorns" **W**:

H29 **LsMs12WR : LWMR.**

In the context **LVs13R** the operator **s13** "projects" **V** onto **sK**:

H30 **LVs13R : LVs4spsnR.**

An application of this projection is the transformation of many-valued logic into Boolean logic, but see also H50.

We now turn to the simplest arithmetic operations on integers.

In the context **LVV's40R** the operator **s40** delivers **V+V'**:

H42 **LsMsM's40R : LsMM'R.**

H43 **LsnNpPR : LsNPR.**

H44 **LspPnNR : LsPNR.**

In the context **LVV's41R** the operator **s41** delivers **V−V'**:

H45 **LVV's41R : LVV's3s40R.**

Now that we have subtraction of integers at our disposal we turn to operators that deal with relations between values that can be interpreted as truth values, integers or real numbers at will. For some of them the interpretation as truth values and for other ones the interpretation as integer or real numbers is more obvious. Anyhow, we start from the sixteen Boolean operators and drop then the restriction that the two arguments and the result be two-valued. We follow their natural order and number them from **s20** up to **s27** and from **s30** up to **s37**. We shall treat them in a different order so that the reader does not need to look ahead. The context is always **LVV's2HR** resp. **LVV's3HR**.

The operator **s22** delivers **V>V'**:

H33 **LVV's22R : LVV's41s022R.**

H34 **LspPs022R : LspPR.**

H35 **LsNs022R : LsnNR.**

The operator **s24** delivers **V<V'**:

H37 **LVV's24R : LV'Vs22R.**

The operator **s21** delivers **V** *AND* **V'** or min(V,V'):

H32 **LVV's21R : LVV's24s4VV'R.**

The operator **s27** delivers **V** *OR* **V'** or max(V,V'):

H40 **LVV's27R : LVV's22s4VV'R.**

The operator **s20** delivers (**V** *AND* **V'**) *AND NOT* (**V** *AND* **V'**); i.e., FALSE of some size:

H31 **LVV's20R : LVV's21s11s3s21R.**

The operator **s23** delivers **V**, its first operand:

H36 **LVV's23R : LVR.**

The operator **s25** delivers **V'**, its second operand:

H38 **LVV's25R : LV'R.**

The operator **s26** delivers **V≠V'**:

H39 **LVV's26R : LVV's22VV's24s27R.**

The operator **s3H** delivers *NOT* (**VV's2H**):

H41 **LVV's3HR : LVV's2Hs3R.**

The operators **s20**, **s21**, **s23**, **s25**, **s27**, **s30**, **s31**, **s35** and **s37** can deliver **s**, i.e., UNDEFINED, whereas **s22**, **s24**, **s26**, **s32**, **s34** and **s36** cannot.

We now consider operators which have to do with names. In the context **LsM's6HMWR**, where the syllable **sM'** can best be interpreted as a name which refers to *SM'*, the operator **s6HM** generates an operator **s6M'** and takes out **sM'** but not itself; an operator **s6pP** is inserted to the left of *Sp*, an operator **s6N** is inserted to the right of *S*:

H58 **LsM's6HMWR : sM'ss22s4(s6M'L)(Ls6M')s6HMWR.**

An inserted operator **s6pP** steps over the value on its right losing one **p** at that step and an inserted operator **s6nN** steps over the value on its left losing one **n** at that step. In doing so, an inserted operator **s6M'** either comes to rest as **s6** at the right of *SM'* or, if it threatens to step out of **L**, it disappears together with **s6HMW**:

H19 **Ls6pPVR : LVs6PR.**
H20 **Ls6pPs6HMWR : LR.**
H21 **LVV's6nNR : LVs6NV'R.**
H22 **Vs6nNLs6HMWR : VLR.**

What happens after the operator **s6** has come to rest, to the right of *SM'*, depends on **H** and **M**.

In the context **LsM's60R** the operator **s60** "dereferences" **sM'**:
H59 **LVs6L's60R : LVL'VR.**

In the context **LV'sM's61R** the operator **s61** "assigns" **V'** to **sM'**:
H60 **LVs6L'V's61R : LV'L'R.**

In the context **LV'sM's62R** the operator **s62** "inserts" **V'** to the right of *SM'*:
H61 **Ls6L'V's62R : LV'L'R.**

In the context **LsM's63KR** the operator **s63p** duplicates the part of **L** to the right of *SM'* and the operator **s63n** duplicates the part of **L** that ends with *SM'*:
H62 **Ls6L's63KR : LsKs4L'LL'R.**

In the context **LsM's64MWR** the operator **s64** makes the operator **W** operate on *SM'* whereas the operators **s64p** resp. **s64n** make **W** operate on *SM'* and all syllables of **L** to the right resp. to the left of it:
H63 **LVs6L's64MWR : LVWsMss36s4(L')(s6ML's64MW)R.**

Now that we have relations and operations concerning names at our disposal we can continue defining operations on integers.

In the context **LVV's42R** the operator **s42** delivers **V**×**V'**:
H46 **LVV's42R : LV'ss36s4s(V's4(VV'sps41s42Vs40)(VV's3s42s3))R.**

In the context **LVV's43R** the operator **s43** delivers the quotient **V**÷**V'** followed by the remainder **V**÷×**V'**:
H47 **LVV's43R : LsVs03pV's03ps043Vs4(V's4( )(sns64s3))(V's4(sns64ps3)s3)R.**
H48 **LsP''sPspP's043R : LsPspP's34s4(spP''sPspP's41spP's043)(sP''sP)R.**

In the context **LVsPs44R** the operator **s44** delivers **V**↑**sP**:
H49 **LVsPs44R : LsPss36s4sp(VsPsps41s44Vs42)R.**

In the context **LVV'spPs45R** the operator **s45** "rounds" **V** if $2 \times ABS$ **V'**>**spP**:
H50 **LVV'spPs45R : LVV'V's40s03pspPs22s4(V's13s40)( )R.**

Next we consider operations on real numbers. The mathematical interpretation of a value as a real number is its interpretation as an integer divided by the "base", i.e., some positive integer greater than one, e.g., 10↑18. The operator **s4HpP** interprets syllable *SpP* as that base.

In the context **LVV's40pPR** the operator **s40pP** delivers **V**+**V'**:
H51 **LVV's40pPR : LVV's40R.**

In the context **LVV's41pPR** the operator **s41pP** delivers **V**−**V'**:
H52 **LVV's41pPR : LVV's41R.**

In the context **LVV's42pPR** the operator **s42pP** delivers **V**×**V'**:
H53 **LVV's42pPR : LVV's42spPs60s43spPs60s45R.**

In the context **LVV's43pPR** the operator **s43pP** delivers **V**/**V'**:
H54 **LVV's43pPR : LVspPs60s42V's43spPs60s45R.**

In the context **LVV's44pPR** the operator **s44pP** delivers **V**↑**V'**, where **V'** is interpreted as an integer:
H55 **LVV's44pPR : LV'ss36s4(spPs60)(V's4(VV'sps41s44pPVs42pP) (spPs60VV's3s44pPs43pP))R.**

So far, we have discussed only operators of general and widely applicable character like those in the standard-prelude of ALGOL 68. We shall now give one example of a very specific operator like one that one might expect in a particular-prelude of ALGOL 68.

In the context **LspPs50R** the operator **s50** delivers the **spP**-th prime number:

H56 **LspPs50R : LspPsppspps050R.**

H57 **LspPsppP'sppP''s050R : LsppP'sppP''s43sns64s10ss22 s4(spPsppP'spppP''s050) (sppP'sppP''s22s4(spPspppP'spps050)(spPsps22s4(sPspppP'spps050)sppP'))R.**

Rule H56 introduces two new values both set to two, the first of which is a candidate for being a prime number and the second of which is a trial divisor to investigate whether the candidate is actually a prime number, and a new operator **s050**. Rule H57 determines the remainder of the candidate upon division by the trial divisor; if this remainder is positive then the trial divisor is not a divisor of the candidate and the next trial divisor is tried; if the remainder is zero then the rule determines whether the candidate is greater than the trial divisor; if this is so then the candidate is a composite number and the next candidate is investigated starting again with a trial divisor two; otherwise, the candidate is a prime number and the rule investigates the number of prime numbers that were still to be found; if this number is greater than one then that number is diminished by one and the next candidate is investigated; otherwise, the candidate is the sought prime number.

Next we define some ALGOL 68-like constructs.

In the context **LVs5(WR')R** the operator **s5** "selects" a syllable from the sequence **WR'** roughly as *CASE* **V** *IN* **W, R'** *ESAC* does:

H17 **LVs5(WW'R')R : LVss32s4W(Vsps41s5(W'R'))R.**

H18 **LVs5(W)R : LWR.**

Here the obvious interpretation of **V** is an integer. The syllables of **WR'** are counted from 0 up to k, say; if $V \leqslant 0$ then **W** is selected, if $V \geqslant k$ then syllable k and, otherwise, syllable **V** is selected.

In the context **Ls7WW'R** the operator **s7** "repeats" **W'** subject to the condition **W** like *WHILE* **W** *DO* **W'** *OD* does:

H23 **Ls7WW'R : LWs4(W's7WW')( )R.**

In the context **LVV'V''s8WW'R** the operator **s8** repeats **W'** subject to a bound test and the condition **W** like *FROM* **V** *BY* **V'** *TO* **V''** *WHILE* **W** *DO* **W'** *OD* does:

H24 **LVV'V''s8WW'R : LV'ss36s4sp(VV''V's4s32s34)VV'V''s8pWW'R.**

H25 **LsMVV'V''s8KWW'R : LsMs4(sKs4(WVV'V''s8n)(W'VV's40V'V''s8)WW')sKR.**

In contrast to the ALGOL 68 clause a non-void result is delivered, viz., **sp** when the test on **V** and **sn** when the test on **W** failed.

Now we consider operators that perform the transition from conventional decimal notation to our unary notation and vice versa.

In the context **LsKHJR** the operators **spHJ** resp. **snHJ** deliver **s** followed by **HJ** times **p** resp. **n**:

H4 **LsKHJR : Lss0KHJR.**

H5 **LVs0KHJR : LVsppppppppppps42s00KH0123456789s40s0KJR.**

H6 **LVs0KR : LVR.**

H7 **LsM00KHH'JR : LsKM00KHJR.**

H8 **LsM00KHHJR : LsMR.**

Rule H4 introduces **s**, i.e., zero as preliminary result, and the new operator **s0KHJ**. Rule H5 multiplies the preliminary result by ten, applies the operator **s00KH0123456789** to add the number "suggested" by the leading decimal digit **H** with the sign as given by **K**, and then deals with the next digit of **J**, if any, by a repeated application of rule H5. The rules H7 and H8 deal with this new operator. Notice that this operator in H5 has zero letters **p** or **n** between **s** and **00** and that **H** is fac-

ing the digit **0**. Application of rule H7 inserts a letter **p** or **n** after **s**, so that there is now one letter **p** or **n** between **s** and **00** and the digit **0** that **H** was facing is taken out so that **H** is now facing the digit **1**. In general, after a number of applications of rule H7 the number of letters **p** or **n** between **s** and **00** equals the number suggested by the digit **H′** that **H** is facing. Repeated application of the rule H7 eventually leads to a blind alley, the only one in our grammar, because **H** is no longer facing a digit **H′**. However, in this process **H** will necessarily face itself once and then rule H8 delivers the desired result, i.e., the value suggested by **H**. When all digits of **HJ** have been dealt with in this way rule H6 delivers the final result.

The operator **sKHJ** enables the grammarian to enter constants in his grammar like, in decimal notation, *125* resp. *−256* by writing **sp125** resp. **sn256**.

In the context **LVs1MR** the operator **s1M** translates **V**, in our unary notation, into a sequence of values, viz., the sign of **V**, the successive decimal digits of **V** and an indication of the number of digits obtained:

H9 **LVs1MR : LVss24s4sn41sMVs03ps9s01s01R.**

H10 **Ls01PsP′L′s01NR : LsP′sp10s34s4(sP′sp10s41s9(s01pP)L′s01N)**
(sPss22s4(s01sPsP′L′s01nN)(sP′L′snN))R.

If **V** is negative then the sign is **sn41**; if **V** is nonnegative then the sign is **sM**. This enables the grammarian to choose, e.g., between the plus mark **sn40**, the blank-space mark **snn** or the invisible mark **sn** by writing **sn40s12s1**, **s1nn** or **s1n** respectively. Rule H9 first determines the sign and then swaps the absolute value of **V** between the operators **s01** and **s01**. Rule H10 then decomposes **V** gradually into a sequence of decimal digits, in unary notation, of course, thereby using the first **s01** as portmanteau for intermediate values of the quotient upon division by ten and the second **s01** as portmanteau for intermediate values of the indication of the number of decimal digits obtained. The final indication is then the negation of that number. The reason for not building up that number itself will soon become clear. Anyhow, if the grammarian does not want the indication he can void it immediately by means of **s10**.

In the context **LVs2R** the operator **s2** "outputs" **V**:

H11 **LVs2R : V , LR.**

It is our only hyperrule in which the comma occurs. Since **V** is a member of the terminal vocabulary V_t it plays no further role in the production process and it can be replaced by its representation. This is our equivalent of output in a programming language.

Since **LVs1MR** produces a sign followed by a number of decimal digits followed by the negation of that number, **LVs1Ms64ps2R** outputs **V** in decimal notation; hence, **Lsp25s2R** outputs *p* and **Lsp25s1ns64ps2R** outputs *25*.

In the context **Ls00R** the operator **s00** "finishes" the production process:

H3 **Ls00R : .**

In the context **s0**, at last, the operator **s0**, i.e., the startword, produces the specific problem that the grammarian has in mind. A very specific hyperrule is, by way of example:

H2 **s0 : sp25sn8spPs1ns10sn9sn2sn41sn2sp25s1ns10sn2sn36sn2spPs50sp25s41s1ns10sps64ps2.**

First we observe that on the right-hand side the metaletter **P** occurs which does not occur on the left; hence its terminal metaproduction can be freely chosen. This is our equivalent of 'input' in a programming language. Suppose that we choose a sequence of one hundred and twenty four times the letter **p** for that terminal production. Then the operators on the right-hand side are harmless ones like **sp25** that delivers **s** followed by twenty five times **p**, and **sn8** that delivers **s** followed by eight times **n**. The first interesting combination is **spPs1ns10**, which produces **snspsppspppppp**. The second interesting combination is **spPs50** which produces **s** followed by six hundred and ninety one times **p** because the 125-rd prime number is 691, which follows from rule H56. Hence the combina-

tion **spPs50sp25s41s1ns10** produces **sppppppspppppppspppppp** because 691 − 25 = 666. At last, the combination **sps64ps2** selects the left-most syllable and outputs it together with all its right-hand side syllables. This then produces

p (1 2 5) − 2 5 = 6 6 6

a result which is the same as that of the ALGOL 68 particular program in the introduction but, of course, now rigorously defined.

6. REFERENCES

[1] WIJNGAARDEN, A. VAN, *Orthogonal design and description of a formal language*, Mathematical Centre, Amsterdam, MR76 (1965).

[2] WIJNGAARDEN, A. VAN et al., eds., *Revised Report on the algorithmic language ALGOL 68*, Acta Informatica **5** (1975) 1-234.

[3] WIJNGAARDEN, A. VAN, *Thinking on two levels*, **in** Proc. Bicentennial Congress of the Wiskundig Genootschap, part 2, P.C. Baayen, D. van Dulst & J. Oosterhoff, eds., Mathematical Centre, Amsterdam, Mathematical Centre Tracts, Vol. 101 (1979) 417-428.