Acceptors from van Wijngaarden grammars: an application of 'Programming languages for automata'

John L. Baker
**

70-02-10

University of Washington
Computer Science Group

Technical Report
70-02-10

* Research supported in part by the National Science Foundation, Grant number GJ - 66

** Supported by IBM fellowship, 1969-70

list of figures, numbered formulas, and symbols

ACCEPTORS FROM VAN WIJNGAARDEN GRAMMARS: AN APPLICATION OF 'PROGRAMMING LANGUAGES FOR AUTOMATA'

introduction

Baker (1970) has characterized the type 0 and type 1 languages in terms of van Wijngaarden grammars (vWg). It is the purpose of the present paper to make a formal presentation of a construction which is only outlined there. The construction is of a nondeterministic linear bounded automaton (lba)- or turing machine (tm)- acceptor for the language specified by a vWg.

The construction is presented here as a schema for a program in a language which is a modification of that defined by Knuth and Bigelow (1967). The programming language (for the classes of lba and tm) defined here may itself be of interest, partly in confirmation of the usefulness of the programming language technique in theory of computing, partly as an extension of the range of applicability of that technique, and partly for the additional language features it includes.

The section of the present paper titled 'background' is extracted from Baker (1970). It is reproduced here to provide a setting for the rest of the present paper, the principal purpose of which is to supply details of the construction in the proof of theorem 2.

Tm and lba are here specified as sets of quadruples (p, a, b, q), where p and q are machine states, a is a tape symbol, and b is either a tape symbol or one of the symbols {L, R}, which indicate one-square shifts of the tape scan point, left and right respectively. The quadruple (p, a, b, q) occurs in the specification of a machine if and only if the machine, in state p and scanning a on its tape, may enter state q and (if b is a tape symbol) print b or (if b ε {L, R}) shift its tape scan according to b.

These machines are nondeterministic (I.e. (p,a) ↦ (q, b) is not required to be univocal.); halt only for want of an instruction (i.e. only by reaching a state p, scanning a symbol a for which no quadruple (p, a, b, q) occurs in the specification), and accept an input by halting in one of a designated set of accepting states. Inputs to these machines are assumed to be left- and right- end-marked with '¢' and '$' respectively. Lba differ from tm only in that, if (p, '¢' [resp. '$'], b, q) occurs in the specification of an lba, then b ε {R, '¢'} [resp. {L, '$'}]. It is convenient here to suppose, in considering tm computations, that each tape square not considered as input contains '$', i.e. to suppose that '$' is the natural tape blank [1].

---

1. In the main construction of the present paper, this supposition is only significant to the details of the operation of the procedure 'endtape'. The reader will have no difficulty in bringing the construction given here into conformity with any other reasonable convention concerning blank tape.

notation

$\setminus$ denotes set difference: $A \setminus B = \{ x \in A \mid x \notin B \}$.

The notion string is not formally defined here, but is supposed to have the following properties: A string x has a unique length $|x| \geq 0$. A string of length one is distinct from its unique component. $\diamond$ is the empty string, the unique string of length zero.

If S is a set, then S^* is the set of strings whose components are elements of S , and $S^+ = S^* \setminus \{\diamond\}$.

If S is a set and n a natural number, then S^n is the n^{th} cartesian power of S , the set of ordered n- tuples of elements of S . It is here explicitly not assumed that $S^n = \{ x \in S^* \mid n = |x| \}$.

The vinculum ($\overline{}$) is used here to indicate a string of length one. Thus: if x and y are strings, $|\overline{xy}| = 1$, $|\bar{x}\,\bar{y}| = 2$, $|xy| = |x| + |y|$; also, $\bar{\bar{s}}$ is the string of length one whose component is the string of length one whose component is s ; also, $\overline{\diamond} \neq \diamond$. This vinculum notation is used only when strings of strings are under consideration. Otherwise, a string of length one is notationally identified with its component.

Notations and definitions not otherwise specified follow Hopcroft and Ullman (1969). Invocation of 'familiar' results and techniques without specific reference may also be taken as referring to the same, in particular to chapters 6-9.

background

Definition 1: A van Wijngaarden grammar (vWg) is an ordered sextuple (M, N, R_M, B, R, s), where:

M is a finite set, the metavariables.

N is a finite set, the protovariables, $M \cap N = \emptyset$.

R_M is a finite set of metarules, $X \to Y$, where $\{ X, Y \} \subset (M \cup N)^*$, with $\forall_{W \in M} (M, N, R_M, W)$ a csg .

B is a finite set, the basic strings, $B \subset N^*$.

R is a finite set of rules $(f_0, f_1, \ldots, f_k, h)$ $(k \geq 0)$, where $\forall_{i=0}^{k} \exists_{n_i \geq 0} f_i \in (N^*)^{n_i+1}$ and h is a function, $\underline{dom}(h) = \{ (i, j) \mid i \in \{0, 1, \ldots, k\}$ and $f_i \in (N^*)^{n_i+1}$ and $j \in \{1, \ldots, n_i\} \}$, $\underline{ran}(h) \subset M$.

$s \in N^*$, s is the starting variable.

In this paper, it is supposed that ',' is not a meta- or protovariable of any vWg.

To permit an explicit presentation of the notions to be defined here, the following example of a vWg with a finite language is given. (Also see figure 2.)

$$
\begin{aligned}
G_0 = (\ & \{ T \}, \\
& \{ a, d, e, f, n, p, s, v, w \}, \\
& \{ T \to f\ ,\ T \to p \}, \\
& \{ \bar{d}, \bar{n}, \bar{v}, \bar{w} \}, \\
& \{ (\bar{s}, \bar{n}, f_2, \bar{v}, f_4, \{ ((2, 1), T), ((4, 1), T) \}), \\
& \qquad (\overline{fa}, \bar{w}, \emptyset), (\overline{pa}, \emptyset), (\overline{fe}, \emptyset), (\overline{pe}, \bar{d}, \emptyset) \}, \\
& \bar{s}\ ) \text{ , where } f_2 = (\diamond, \bar{a}) \text{ and } f_4 = (\diamond, \bar{e}) .
\end{aligned}
$$

(This example is suggested by a familiar analysis of a small piece of the english language, according to the following correspondence:

tense auxiliary -ed ending future noun past sentence verb will

T a d e f n p s v w .)

Notation (for rules): The rule $(f_0 = (x_{00}, x_{01}, \ldots, x_{0n_0}), f_1 = (x_{10}, x_{11}, \ldots, x_{1n_1}), \ldots, f_k = (x_{k0}, x_{k1}, \ldots, x_{kn_k}), h = \{ ((i, j), W_{ij}) \mid i \in \{0, 1, \ldots, k\} \text{ and } j \in \{1, \ldots, n_i\} \})$ may be abbreviated

(2) $$\overline{x_{00}W_{01}x_{01} \cdots W_{0n_0}x_{0n_0}} \to \overline{x_{10}W_{11}x_{11} \cdots W_{1n_1}x_{1n_1}} \cdots \overline{x_{k0}W_{k1}x_{k1} \cdots W_{kn_k}x_{kn_k}}$$ or

$$\overline{f_0(W_{01}, \ldots, W_{0n_0})} \to \overline{f_1(W_{11}, \ldots, W_{1n_1})} \cdots \overline{f_k(W_{k1}, \ldots, W_{kn_k})},$$

omitting parentheses after f_i in case $n_i = 0$.

Thus, the rules of G_0 may also be written:

$$\bar{s} \to \bar{n}\ \overline{Ta}\ \bar{v}\ \overline{Te}\ ,\ \overline{fa} \to \bar{w}\ ,\ \overline{pa} \to \diamond\ ,\ \overline{fe} \to \diamond\ ,\ \overline{pe} \to \bar{d} \quad \text{or}$$

$$\bar{s} \to \bar{n}\ \overline{f_2(T)}\ \bar{v}\ \overline{f_4(T)}\ ,\ \overline{fa} \to \bar{w}\ ,\ \overline{pa} \to \diamond\ ,\ \overline{fe} \to \diamond\ ,\ \overline{pe} \to \bar{d}\ .$$

Definition 2 (canonical representation): If $G = (M, N, R_M, B, R, s)$ is a vWg, then c_G is a function, $\underline{dom}(c_G) = B^*$, $\underline{ran}(c_G) \subset (N \cup \{,\})^*$, defined inductively: $c_G(\diamond) = \diamond$. If $x \in B$ and $X \in B^*$, then $c_G(\bar{x}\ X) = x', 'c_G(X)$.

Thus, $c_{G_0}(\bar{n}\ \bar{w}\ \bar{v}) = n,w,v,$ and $c_{G_0}(\bar{n}\ \bar{v}\ \bar{d}) = n,v,d,$.

Definition 3: If $G = (M, N, R_M, B, R, s)$ is a vWg and E is a set, then a representation of G (in E) is a generalized sequential machine (gsm) D with input alphabet E and output alphabet a subset of $N \cup \{,\}$.

For example, D_0 is a representation of G_0, where

$D_0 = (\{y, z\}, \{c, d, H, w, \_\}, \{d, n, v, w, ',' \}, g, y, \{y\})$ (a gsm), where g is given by:

g	y	z
c	{ (y, v',') }	∅
d	{ (y, d',') }	∅
H	{ (z, h',') }	∅
w	{ (z, w',') }	∅
_	∅	{ (y, ◇) }

Definition 4 (substitution in rules): If $f = (x_0, x_1, \ldots, x_n) \varepsilon (N^*)^{n+1}$ and $(y_1, \ldots, y_n) \varepsilon (N^*)^n$, then $f(y_1, \ldots, y_n) = x_0 y_1 x_1 \cdots y_n x_n \varepsilon N^*$. (In case $n = 0$, understand $f = x_0$ as a nullary function.)

Definition 5 (universal assignment to metavariables): If M, N, R_M are as in definition 1, then

$$\hat{C}(M, N, R_M) = \{ \hat{c} : M \to \cup \{ L((M, N, R_M, W)) \mid W \varepsilon M \} \mid \forall_{W \varepsilon M} \hat{c}(W) \varepsilon L((M, N, R_M, W)) \} .$$

Definition 6: If $G = (M, N, R_M, B, R, s)$ is a vWg, $r = (f_0, f_1, \ldots, f_k, h) \in R$, and $\hat{c} \in \hat{C}(M, N, R_M)$, then the strict rule of G corresponding to $(r, \hat{c})$ is

$$r_S(r, \hat{c}) = \overline{f_0(y_{01}, \ldots, y_{0n_0})} \to \overline{f_1(y_{11}, \ldots, y_{1n_1})} \ldots \overline{f_k(y_{k1}, \ldots, y_{kn_k})}, \text{ where } \forall_{i=0}^{k} \forall_{j=1}^{n_i} y_{ij} = \hat{c}(h(i, j)).$$

The set of strict rules of G is

$$R_S(G) = \{ r_S(r, \hat{c}) \mid r \in R \text{ and } \hat{c} \in \hat{C}(M, N, R_M) \}.$$

For example,

$$R_S(G_0) = \{ \bar{s} \to \bar{n}\, \overline{f_2(y)}\, \bar{v}\, \overline{f_4(y)} \mid y \in \{ \bar{f}, \bar{p} \} \} \cup \{ \overline{fa} \to \bar{w}, \overline{pa} \to \diamond, \overline{fe} \to \diamond, \overline{pe} \to \bar{d} \}$$

$$= \{ \bar{s} \to \bar{n}\, \overline{fa}\, \bar{v}\, \overline{fe}, \bar{s} \to \bar{n}\, \overline{pa}\, \bar{v}\, \overline{pe}, \overline{fa} \to \bar{w}, \overline{pa} \to \diamond, \overline{fe} \to \diamond, \overline{pe} \to \bar{d} \}, \text{ since}$$

$$L((\{T\}, \{a, d, e, f, n, p, s, v, w\}, \{T \to f, T \to p\}, T)) = \{ \bar{f}, \bar{p} \}.$$

Definition 7: If $G = (M, N, R_M, B, R, s)$ is a vWg and D is a representation of G, then the language specified by G is

$$L(G) = \{ X \in B^* \mid s \overset{*}{\underset{G}{\Rightarrow}} X \}, \text{ where } \overset{*}{\underset{G}{\Rightarrow}} \text{ is the reflexive-transitive closure of the binary relation } \underset{G}{\Rightarrow} \text{ in } (N^*)^*, \text{ which is defined by:}$$

$X \underset{G}{\Rightarrow} Y$ if and only if $\exists_{\{U,V\} \subset (N^*)^*}\ X = UX'V$ and $Y = UY'V$ and $(X' \to Y') \in R_S(G)$;

and the representation language specified by (G, D) is

$$L(G, D) = D^{-1}(\{ c_G(X) \mid X \in L(G) \}).$$

For example,

$$L(G_0) = \{ \bar{n}\,\bar{w}\,\bar{v}\ ,\ \bar{n}\,\bar{v}\,\bar{d}\ \} \text{ and } L(G_0, D_0) = \{\ H\_w\_c\ ,\ H\_cd\ \}\ .$$

Definition 8: If $f = (x_0, x_1, \ldots, x_n) \in (N^*)^{n+1}$, then the weight of f is $|f| = \sum_{i=0}^{n} |x_i|$. Note that the weight of f is the length of its abbreviation (2), ignoring metavariables.

Definition 9: A rule $(f_0, f_1, \ldots, f_k, h)$ of a vWg is context-sensitive (cs) if and only if $|f_0| \leq \sum_{i=1}^{k} |f_i|$ and $\forall_{W \in M}\ \underline{card}(\{\ j \mid h(0, j) = W\}) \leq \underline{card}(\{\ (i, j) \mid i > 0 \text{ and } h(i, j) = W\ \})$. Note that a rule abbreviated as (2) is cs if and only if every metavariable occurs at least as often in the rhs as in the lhs and, ignoring metavariables, its rhs looks no shorter than its lhs.

Definition 10: A vWg is cs if and only if each of its rules is cs.

By way of example, note that the following rules fail to be cs:

$$\overline{abcW} \to \overline{aW}\ \bar{U}\ ,\ \overline{abcW} \to \overline{abU}\ \overline{abcV}\ ,\ \overline{abcW} \to \overline{abWUV}\ ,$$

where a, b, c are protovariables, U, V, W are metavariables.

The following is an obvious consequence of the above definitions.

Theorem 0: If $G = (M, N, R_M, B, R, s)$ is a cs vWg and $(\overline{x_0} \to \overline{x_1} \ldots \overline{x_k}) \in R_S(G)$, then $|x_0| \leq \sum_{i=1}^{k} |x_i|$. If $X = \overline{x_1} \ldots \overline{x_n} \in (N^*)^*$, define (temporarily) $[X] = \sum_{i=1}^{n} |x_i|$. If $X_0 \overset{=}{G}> X_1 \overset{=}{G}> \ldots \overset{=}{G}> X_m$, then $[X_0] \leq [X_1] \leq \ldots \leq [X_m]$.

Theorem 1: If G is a [cs] grammar, then there is a [cs] vWg G' and a representation D of G' with $L(G', D) = L(G)$.

Proof: See Baker (1970).

Lemma 1: If $G = (M, N, R_M, B, R, s)$ is a [cs] vWg, then there is a [cs] vWg $G' = (M', N', R_M', B', R', s')$ with $\forall_{W \in M'} \diamond \notin L((M', N', R_M', W))$ and $L(G') = L(G)$.

Proof: By a familiar result, there is, given $W \in M$, a csg (M_W, N, R_W, W) with $L((M_W, N, R_W, W)) = L((M, N, R_M, W)) \setminus \{\diamond\}$. Without loss of generality, suppose the M_W $(W \in M)$ and M, N are pairwise disjoint, and that, in each rule of each R_W, some element of M_W occurs on the lhs. Then the R_W are also pairwise disjoint. Define:

$$M' = M ,$$

$$N' = N ,$$

$$B' = B ,$$

$$s' = s ,$$

$$R_M' = \bigcup \{ R_W \mid W \in M \} , \text{ and}$$

$$R' = \{ \overline{x_{00} W'_{01} x_{01} \cdots W'_{0n_0} x_{0n_0}} \rightarrow \overline{x_{10} W'_{11} x_{11} \cdots W'_{1n_1} x_{1n_1}} \cdots \overline{x_{k0} W'_{k1} x_{k1} \cdots W'_{kn_k} x_{kn_k}} \mid (\overline{x_{00} W_{01} x_{01} \cdots W_{0n_0} x_{0n_0}} \rightarrow \overline{x_{10} W_{11} x_{11} \cdots W_{1n_1} x_{1n_1}} \cdots \overline{x_{k0} W_{k1} x_{k1} \cdots W_{kn_k} x_{kn_k}}) \in R$$
$$\text{and } \forall_{i=0}^{k} \forall_{j=1}^{n_i} (W'_{ij} = W_{ij} \text{ or } (\diamond \in L((M, N, R_M, W)) \text{ and } W'_{ij} = \diamond)) \} .$$

It is easy to see that $R_S(G') = R_S(G)$, whence, since G' and G have the same protovariables, basic strings, and starting string, $L(G') = L(G)$. Evidently, G' is cs if and only if G is.

Theorem 2: If G is a vWg [resp. a cs vWg] and D is a representation of G , then there is a nondeterministic tm [resp. lba] which accepts $L(G, D)$.

Proof: By construction, given $G = (M, N, R_M, B, R, s)$, of a nondeterministic tm [resp. lba] A , acting, by familiar techniques, on a six-track tape. In discussing the construction, it is convenient to make the following definitions:

(3) c is a function, $\underline{dom}(c) = ((N \cup M)^*)^*$, $\underline{ran}(c) \subset (N \cup M \cup \{,\})^*$, $c(\diamond) = \diamond$, (inductively) if $x \in (N \cup M)^*$, $X \in ((N \cup M)^*)^*$, then $c(\bar{x}\, X) = x','c(X)$.

$\underline{aug}$ is a function. If $X \in (N \cup \{,\})^*$, then $\underline{aug}(X)$ is the six-track string with X on track 1, blanks on tracks 2 through 6.

D_a is a gsm, identical to D , except that D_a outputs $\underline{aug}(X)$ whenever D outputs X .

I is a function. If X is a six-track string, then $I(X)$ is the part on track 1.

Immediately from the definitions, c is one-to-one, $I(\underline{aug}(X)) = X$, and $D_a(Y) = \{ \underline{aug}(c(X)) \mid c(X) \varepsilon D(Y) \}$. In the construction to be given,

(4) A accepts $T = \{ X \mid I(X) \varepsilon c(L(G)) \}$.

By definition, $D_a^{-1}(T) = \{ Y \mid D_a(Y) \cap T \neq \emptyset \}$. Therefore, by the present remarks, $D_a^{-1}(T) = \{ Y \mid \exists_Z\ c(Z) \varepsilon c(L(G))$ and $c(Z) \varepsilon D(Y) \}$ $= L(G, D)$, and, by the familiar result that the set of languages accepted by tm [resp. lba] is closed under inverse gsm mappings, the construction of A satisfying (4) is sufficient to prove the existence of the required acceptor of $L(G, D)$.

A produces on its track 1, given input X, a sequence $X_0, X_1, \ldots$ with $X_0 = I(X)$, $\forall_i\ c^{-1}(X_{i+1}) \overset{=}{G}\!\!> c^{-1}(X_i)$ [and $|X_{i+1}| \leq |X_i|$]. A accepts X if and only if $c^{-1}(X_0) \varepsilon B^*$ and $\exists_n\ X_n = c(\overset{=}{s})$. Therefore, $T \subset \{ X \mid I(X) \varepsilon c(L(G)) \}$. A produces X_{i+1} from X_i by selecting (nondeterministically) a rule $(Y \to Z) \varepsilon R$ (consider the rule to be written in the form (2).); placing $c(Y)$ on track 3 and $c(Z)$ on track 2, each left-justified with X_i; generating (nondeterministically) a strict rule $(Y' \to Z') \varepsilon R_S(G)$ from $(Y \to Z)$ with $c(Y')$ on track 3 and $c(Z')$ on track 2, each still left-justified with X_i; shifting tracks 2 and 3 right (nondeterministically) k squares $(k \geq 0)$; and finally, in case $X_i = Uc(Z')V$ and $|U| = k$ and ($U = \diamond$ or U ends with ','), replacing X_i on track 1 by $X_{i+1} = Uc(Y')V$. Tracks 4, 5, and 6 are used in the generation of $(Y' \to Z')$ from $(Y \to Z)$. (See figure 1.) All strict rules of G which could be used in a derivation $\overset{=}{s} \overset{*}{\overset{=}{G}}\!\!> c^{-1}(X_0)$ are available for selection by A at each stage,

since there is no restriction on its choices. [resp. since the only restrictions on its choices are $|c(Z')| + k \leq |X_0|$, $|c(Y')| + k \leq |X_0|$, $|c(Z)| \leq |X_0|$, and $|c(Y)| \leq |X_0|$, all of which follow (via theorem 0) in case G is cs and

(5) $$\forall_{W \in M} \Diamond \notin L((M, N, R_M, W)) ,$$

from the obvious restriction $|c(Z')| + k \leq |X_i|$ on the applicability of $(Y' \rightarrow Z')$; and since lemma 1 shows there to be no loss of generality in supposing G to satisfy (5).] Therefore, if $(\bar{s} = X_0), X_1 , \ldots, X_n$ is a derivation in G , then A can produce the sequence $c(X_n), c(X_{n-1}), \ldots, c(X_0)$ on its track 1 , given input $\underline{aug}(c(X_n))$. I.e. $\{ X \mid I(X) \in c(L(G)) \} \subset T$.

Figure 1 is another outline of the algorithm embodied in A , with some further details. (In figure 1, T.i denotes the contents (non-blank) of track i .)

Figure 2 is an example of the operation of such an algorithm corresponding to the grammar G_0 .

Corollary (main result): A language L is type 0 [resp. type 1] if and only if there is a vWg [resp. a cs vWg] G and a representation D of G with $L = L(G, D)$.

Proof: By familiar results, directly from theorems 1 and 2.

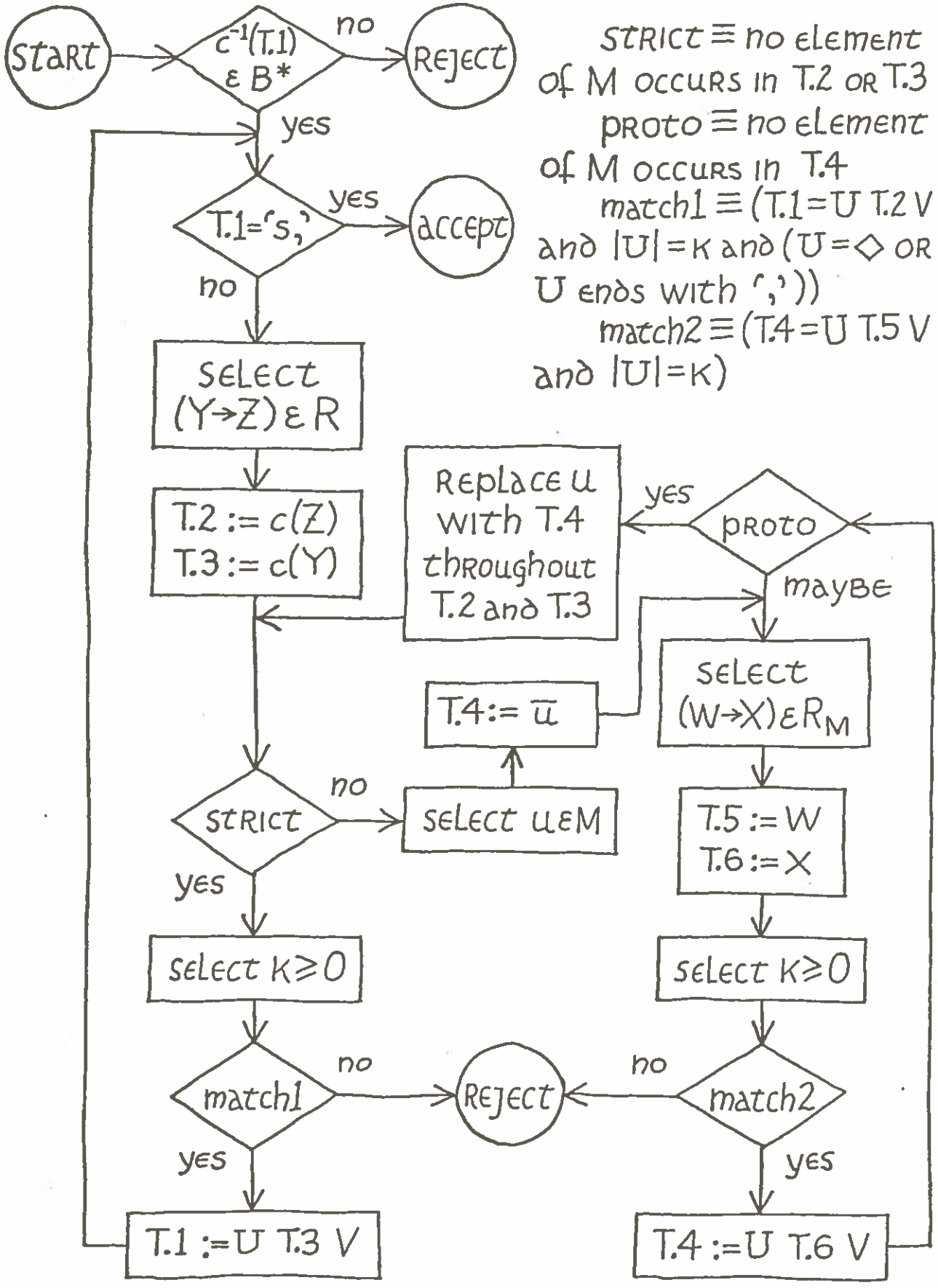
start
$c^{-1}(T.1) \in B^*$
no
Reject
yes
T.1='s,'
yes
accept
no
select $(Y \to Z) \in R$
T.2 := c(Z)
T.3 := c(Y)
strict
no
select $u \in M$
T.4 := u
Replace u with T.4 throughout T.2 and T.3
yes
proto
maybe
select $(W \to X) \in R_M$
T.5 := W
T.6 := X
select $K \geq 0$
select $K \geq 0$
match1
no
Reject
no
match2
yes
yes
T.1 := U T.3 V
T.4 := U T.6 V
strict ≡ no element of M occurs in T.2 or T.3
proto ≡ no element of M occurs in T.4
match1 ≡ (T.1 = U T.2 V and |U| = K and (U = ◇ or U ends with ','))
match2 ≡ (T.4 = U T.5 V and |U| = K)

figure 1.

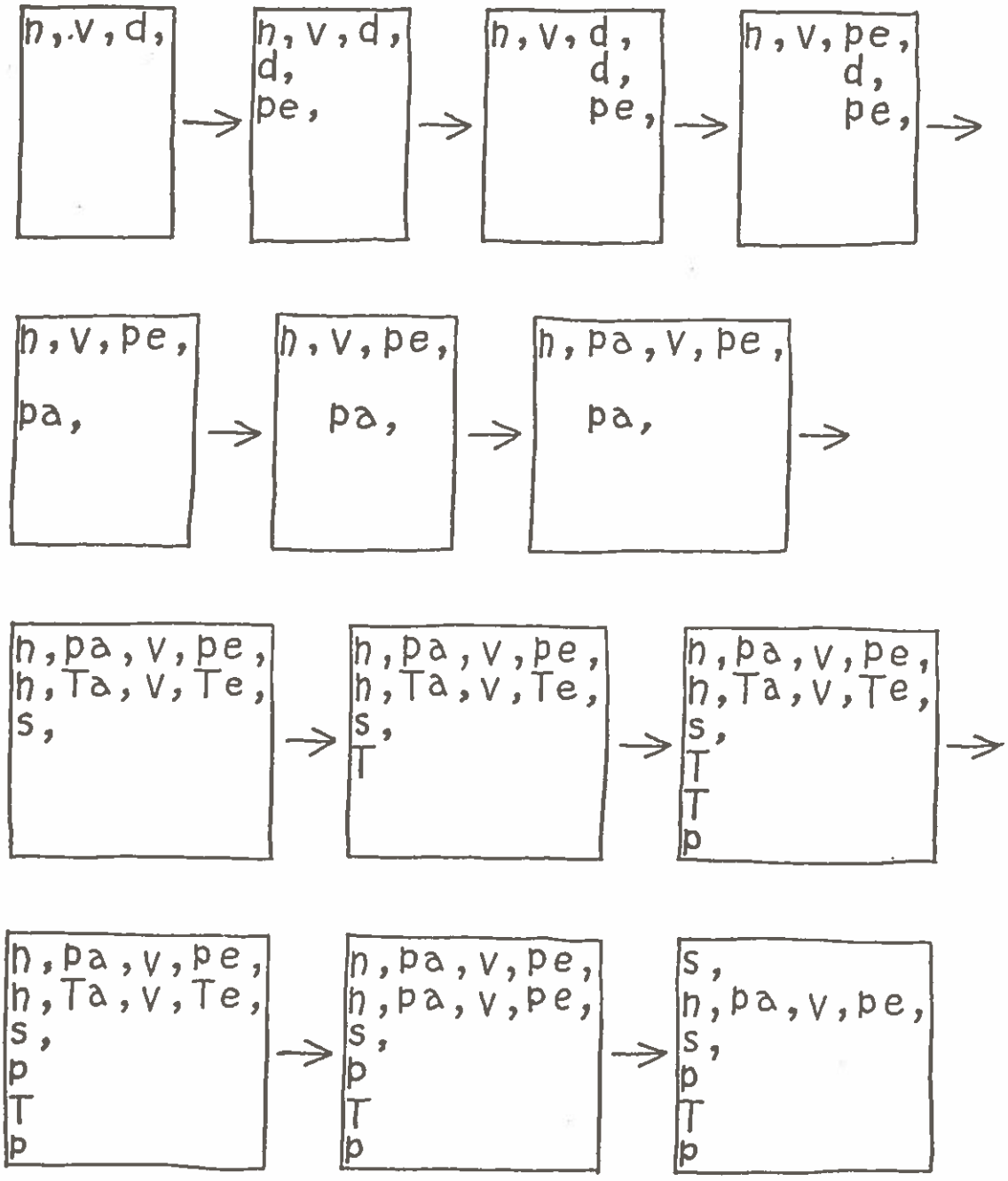
n, v, d,
n, v, d,
d,
pe,
n, v, d,
d,
pe,
n, v, pe,
d,
pe,
n, v, pe,
pa,
n, v, pe,
pa,
n, pa, v, pe,
pa,
n, pa, v, pe,
n, Ta, v, Te,
s,
n, pa, v, pe,
n, Ta, v, Te,
s,
T
n, pa, v, pe,
n, Ta, v, Te,
s,
T
T
p
n, pa, v, pe,
n, Ta, v, Te,
s,
p
T
p
n, pa, v, pe,
n, pa, v, pe,
s,
p
T
p
s,
n, pa, v, pe,
s,
p
T
p

figure 2.

the programming language, informally

Knuth and Bigelow (1967) have defined, in detail, a programming language for the class of deterministic stack automata (dsa). A program in this language has the general appearance of a one-block algol program, with lisp-style conditional statements. The language defined is a programming language in the sense that it permits an intuitively orderly, easily documentable, and reasonably compelling presentation of an algorithm. It is a language for the class of dsa in the sense that each program in it is mechanically translatable into a usual sort of specification of a dsa.

The language to be defined in the present paper will follow the general outlines of that of Knuth and Bigelow (1967). It will differ in detail from the latter to the end that each program will not specify a dsa, but rather a nondeterministic tm and a nondeterministic lba. It will also differ in detail in that some familiar 'programming techniques' for automata, namely variables and multiple tracks, are explicitly provided for in the language design. Another difference arises from the differing mathematical intents of the two papers. Knuth and Bigelow (1967) used their programming language to specify one particular dsa (diagonalizing the lba), whereas the present paper will use its programming language to show how to specify a tm or lba for each w∈g. I.e. the present paper requires a notation for tm- or lba- schemata.

Figure 3 exposes most of the ideas and constructions of the tm-lba programming language to be defined here.

```
      begin
⟨3.1⟩ tracks: 2; alphabets: ({ a_i | i ε {1, ..., n} | , }), (_, +);
⟨3.2⟩ variable x ε ({ a_i | i ε {1, ..., n} | , });
      procedure rewind;
        begin [ ¢ => R ; => L, repeat ] end rewind;
⟨3.3⟩ { procedure bump_i(var);
          begin [ var = a_i => var ← a_i+1 ] end bump_i
⟨3.4⟩ | i ε {1, ..., n-1} | ; } ;
⟨Choose k ≥ 1 . Underline the first k symbols with '+'.⟩
⟨3.5⟩ R, [ => 2.+, R, repeat or rewind ],
⟨Check that the underlined symbols are all a_i (i = 1 initially).⟩
⟨3.6⟩ S: [ 2.+ => [ 1.x => R ], repeat ; => ],
⟨If '$' follows the underlined symbols and the a_n have been checked, accept.⟩
⟨3.7⟩ [ $ => [ x = a_n => accept ] ;
⟨If a_i+1 follows the underlined symbols, set i ← i+1 .⟩
⟨3.8⟩ { 1.a_i+1 => bump_i(x) | i ε {1, ..., n-1} | ; } ], L,
⟨Shift the '+' to underline the next group of k symbols, then go to 3.6 .⟩
      [ 1.x => go to S ; => [ 2._ => 2.+, [ ¢ => R ; 2._ => R ; => L, repeat ],
                                                          2._, R ;
                                => R, repeat ], repeat ]
      end
```

Figure 3.

Figure 3 is a program schema which becomes a program upon fixing an integer $n \geq 1$ and symbols $(a_1, a_2, \ldots, a_n)$. For such a fixed choice, it is a program specifying a tm and an lba accepting $\{a_1^k a_2^k \ldots a_n^k \mid k \geq 1\}$.

The portions of figure 3 in broken brackets ('<', '>') are commentary, and are ignored in parsing and translating the program.

The construction $\{ X(u) \mid Y(u) \mid Z \}$, which occurs in lines 3.1, 3.2, 3.3-4, and 3.8, realizes the schematic character of the example. $Y(u)$ is a formal statement (a predicate), possibly involving u as a free variable, whose content is fixed by the choice which turns the schema into a program. For example, in line 3.1, where $u = i$ and $Y(i) \equiv i \in \{1, \ldots, n\}$, the choice $n = 3$ would fix the content of $Y(i) \equiv i \in \{1, 2, 3\}$. For each choice turning the schema to a program, it must be the case that each expression $Y(u)$ occurring in the program is satisfied by finitely many u. $X(u)$ is an expression which, for each choice of u satisfying $Y(u)$, is a string of symbols of the programming language. Z is a string of symbols, used as a delimiter. If the choice turning the schema to a program fixes $Y(u)$ so that it is satisfied only by $\{u_1, u_2, \ldots, u_m\}$, then $\{ X(u) \mid Y(u) \mid Z \}$ is deemed, for purposes of the translation, to be replaced in the schema by $X(u_1)ZX(u_2)Z\ldots ZX(u_m)$. The order of the solutions to $Y(u)$ is only fixed if they are all integers, in which case they are taken in their natural order. (If an expression $Y(u)$ with non-integral solutions is used in a schema, the schema generally fails to have a unique expression as a program, even when $Y(u)$ is specified. It is intended that in this case the language be so used that the automata specified by various orderings of the solutions to $Y(u)$ are strongly isomorphic. This is so in the main construction of the present paper.)

In the present example, suppose $n = 3$ and $(a_1, a_2, a_3) = (a, b, c)$ are fixed. Then '{ a_i | i ε {1, ..., n} | , }' in lines 3.1 and 3.2 is replaced by 'a, b, c' . Also,

'{ procedure $bump_i$(var);
begin [var = a_i => var <- a_{i+1}] end $bump_i$
| i ε {1, ..., n-1} | ; }'

in lines 3.3-4 is replaced by

'procedure $bump_1$(var);
begin [var = a => var <- b] end $bump_1$;
procedure $bump_2$(var);
begin [var = b => var <- c] end $bump_2$;' .

Also, '{ 1.a_{i+1} => $bump_i$(x) | i ε {1,..., n-1} | ; }' in line 3.8 is replaced by '1.b => $bump_1$(x) ; 1.c => $bump_2$(x)' .

From a logical point of view, the replacements just described must be made prior to any steps in the translation of the program, since, until the replacements are made, there is no program, only a schema. Furthermore, elimination of the { X | Y | Z } construction may not be sufficient to specify the program. For example, it is also necessary in the present case to replace 'a_n' in line 3.7 by 'c' .

The first step of the translation process itself is the elimination of the declared procedures and of all references to them. As in Knuth and Bigelow (1967), the procedures are really macros. They are eliminated by being copied, with arguments suitably replaced, in place of their calls.

In the present example, still supposing $(a_1, \ldots, a_n) = (a, b, c)$, 'rewind' is replaced in line 3.5 by '[¢ => R ; => L, repeat]', '$bump_1(x)$' is replaced by '[x = a => x <- b]', and '$bump_2(x)$' is replaced by '[x = b => x <- c]'. For this example, then, figure 6 is the program to be translated.

```
      begin
⟨4.1⟩ tracks: 2; alphabets: (a, b, c), (_, +);
⟨4.2⟩ variable  x ε (a, b, c);
⟨4.3⟩ R, [ => 2.+, R, repeat or [ ¢ => R ; => L, repeat ] ],
⟨4.4⟩ S: [ 2.+ => [ 1.x => R ], repeat ; => ],
⟨4.5⟩ [ $ => [x = c => accept ] ;
⟨4.6⟩ 1.b => [x = a => x<- b ] ;
⟨4.7⟩ 1.c => [x = b => x<- c ] ], L,
⟨4.8⟩ [ 1.x => go to S ; => [ 2._ => 2.+, [ ¢ => R ; 2._ => R ; => L, repeat ],
⟨4.9⟩                                                          2._, R ;
⟨4.10⟩                                => R, repeat ], repeat ]
      end
```

Figure 4.

The remaining declarations (in lines 4.1 and 4.2) specify that the machine's tape is to be considered as divided into two tracks, one to be marked with the symbols {a, b, c}, the other with {_, +}, and that its finite control is to be used to store as the value of a variable x an element of {a, b, c}, initially 'a', the first one listed.

By convention, the input is the initial contents of track 1 (the one whose alphabet is {a, b, c}), and the remaining track is initially to be filled with the first symbol of its alphabet, '_' [2]. The left- and right- end-marks, '¢' and '$' , respectively, do not lie on any track, but each occupies one whole tape square. The initial scan point is '¢' .

The rest of the program (after the declarations) consists of a list of actions of one sort or another, separated by commas. The basic flow of control in the program is from the first of these ('R') to the second ('[=> 2.+ ...]]'), and so on. There are, however, labels, such as 'S' in line 4.4, and jumps, such as 'go to S' in line 4.8, which modify the basic flow in an obvious way.

The square-bracketed portions of the program, which are lisp-style conditional statements, provide a further modification of the control flow. These conditional statements consist of a sequence of expressions, each dominated by '=>', separated by semi-colons. The portion of each such expression left of '=>' is a condition which may be satisfied or not. The action of the whole conditional statement is to examine the conditions, one at a time, from left to right, until one which is satisfied is found; then to perform the list of actions, which may be empty, right of the '=>' associated with that condition; then to go on with the next action following the whole conditional statement. (But see the description of 'repeat', below.) If no condition is satisfied, the effect is that the machine halts without accepting the input.

---

2. This convention is adopted for the sake of the present example. Its effect is incorporated into the design of A (see theorem 2) in the main construction, so that it need not be assumed there.

Some possible conditions are: that a particular symbol is being scanned ('¢' in line 4.3, '$' in line 4.5); that the tape square being scanned has a particular symbol on a particular track ('2.+' in line 4.4, '1.b' in line 4.6); that the tape square being scanned has, on a particular track, the symbol which is the current value of a particular variable ('1.x' in line 4.4); that a particular symbol is the current value of a particular variable ('x = c' in line 4.5); the empty condition, which is always satisfied (twice in line 4.3, four places elsewhere).

An additional effect on the flow of control due to conditional statements is provided by the atomic action 'repeat'. Its action is to cause the smallest conditional statement of which it is a constituent to be performed again.

Additional atomic actions are: to write a particular symbol on a particular track of the currently scanned tape square ('2.+' in lines 4.3 and 4.8, '2._' in line 4.9); to shift the scan point left or right ('L' and 'R' respectively); to set the current value of a particular variable ('x ← b' in line 4.6); to halt the machine, accepting the input ('accept' in line 4.5).

The reader should now be able to verify the following descriptions: The effect of the inner nested conditional on line 4.3 is to position the scan point just right of '¢'. The effect of the innermost nested conditional on line 4.8 is to position the scan point just right of the rightmost square which is left of or at the present scan point, and which has '_' on track 2; or, if there is no such square, then just right of '¢'.

The effect of the intermediately nested conditional on lines 4.8-10 is to replace the leftmost '_' on track 2 which is at or right of the present scan point with a '+', and to replace the leftmost '+' on track 2 which is at or left of the new '+' with a '_' (Recall that only '_' and '+' occur on track 2.), all unless there is no '_' at or right of the present scan point, in which case the effect is to fail to accept the input. (In the lba translation, the failure is due to an 'attempted' right shift scanning '$'. In the tm translation, the failure is due to an interminably repeated right shift, since '$' is the natural tape blank.) The effect of the conditional in lines 4.5-7 is given by the following table:

value of x	contents of tape square under scan							
	(a, _)	(a, +)	(b, _)	(b, +)	(c, _)	(c, +)	¢	$
a	fail	fail	x ← b	x ← b	fail	fail	fail	fail
b	fail	fail	fail	fail	x ← c	x ← c	fail	fail
c	fail	fail	fail	fail	fail	fail	fail	accept

The nondeterminism of the machine specified by this program is due to the 'or' in line 4.3. In general, an action can be a list of actions separated by 'or's, in which case any of the listed actions may be chosen. In particular, the effect of line 4.3 (noting that it is at the beginning of the program) is to write one or more '+'s on track 2, beginning at its left end, then to position the scan point just right of '¢', i.e. at the first '+' written.

The reader should now be able to verify the descriptions given as commentary to figure 3, and that it specifies an acceptor for $\{ a_1^k a_2^k \ldots a_n^k \mid k \geq 1 \}$.

the programming language, formally

The present section specifies which strings of symbols are (syntactically) valid

(i) program schemata,

(ii) programs (with procedures), and

(iii) procedure-free programs,

and what tm [resp. lba] are specified by such strings. Logically, the definition for (iii) is prior to that for (ii) and similarly (ii) prior to (i). The definitions are presented here in the reverse of this logical order, to provide better motivation.

Definition 11: A program schema S depending on formal parameter P is valid over a set $\underset{\sim}{P}$ if and only if each choice of $P \varepsilon \underset{\sim}{P}$ turns S into a valid program (with procedures) X by the process outlined in the preceding section. For such S, $\underset{\sim}{P}$, X, the tm [resp. lba] corresponding to $P \varepsilon \underset{\sim}{P}$ specified by S is the tm [resp. lba] specified by X.

For example, if $(a_1, a_2, \ldots)$ is some fixed non-repeating sequence of symbols, then figure 3 is a program schema depending on formal parameter n valid over the set of positive whole numbers. Also: the main construction of the present paper is a program schema depending on formal parameter G (asserted to be) valid over any set of vdg.

The lack of formality in definition 11 ('Turns into' is ill-defined.) is justified by the fact that it applies to objects explicitly presented, indeed, for the main purpose of the present paper, to just one such object, namely the program schema depending on G for the acceptor A of theorem 2.

The alternative to this style of presentation is the formal introduction of another metalinguistic level, the details of which would be more distracting than enlightening.

Knuth and Bigelow (1967) give a syntax for procedure declarations and statements (calls) which requires only changes in character set to serve here, and which will not be repeated here. It is sufficient to note that such a syntax serves to ensure that the procedure constructs have the general appearance of the corresponding algol 60 constructs, and have the following properties:

Each procedure call occurs in a textual setting in which a ⟨statement list⟩ is valid. (See below. The syntax of ⟨simple statement⟩ is expanded to include procedure calls.) Each actual parameter of each procedure call is a string of symbols in which commas only occur within properly nested parentheses and brackets, and in which all parentheses and brackets are properly nested. The actual parameters of each procedure call agree in number with the formal parameters of the associated procedure declaration, and, if the actual parameters are substituted for the corresponding formal parameters throughout the body of the procedure declaration, the resulting text between 'begin' and 'end' is a valid ⟨statement list⟩.

Definition 12: A program (with procedures) X is valid if and only if the following transformation process terminates when applied to X, with the result a valid procedure-free program Y. For such X, Y, the tm [resp. lba] specified by X is the tm [resp. lba] specified by Y.

Transformation Process (Elimination of procedure constructs):

Step 1: If there are no procedure calls in the text (as transformed so far), erase all procedure declarations and halt. Otherwise go on to step 2.

Step 2: Transform the text by copying the appropriate procedure body (between 'begin' and 'end'), with actual parameters substituted throughout for formal parameters, in place of the first procedure call in the text. Go on to step 3.

Step 3: Replace each label and each reference to it throughout the modified procedure body just copied in step 2 by a new label (not occurring elsewhere in the text). Return to step 1.

The selection and transformation process implicit in definitions 11 and 12 is exemplified in the preceding section: If $(a_1, a_2, a_3, \ldots) = (a, b, c, \ldots)$, then the tm [resp. lba] corresponding to $(n =) 3$ specified by figure 3 (a program schema) is the tm [resp. lba] specified by figure 4 (a procedure-free program).

The remainder of the present section defines (syntactic) validity for procedure-free programs, and defines a transformation of valid procedure-free programs into quadruples specifying tm [resp. lba] in the manner indicated in the introduction to the present paper. Validity is defined in terms of a backus-form grammar with restrictions (stated in prose). The transformation to quadruples is defined in terms of three functions, whose arguments are strings of programming language symbols and some bookkeeping quantities. These restrictions and function definitions are interspersed with the backus-form syntax equations.

An informal description of the functions used to define the transformation to quadruples follows:

quad(X, p, q, s) is the set of quadruples corresponding to the occurrence of program string X starting at position number p (in the program string), ending at position number q, with s the 'first new available' position number just left of X.

Cquad(X, p, q, s, E) is the set of quadruples corresponding to X, p, q, s as for quad. X must be a part of a conditional statement (in fact, a <conditional tail>), and E is the set of conditions specified left of X.

next(X, s) is the 'first new available' position number right of the program string X, supposing s to be the 'first new available' position number just left of X.

No formal statement of the possible values of the arguments of these functions needs to be made, since the functions are usefully evaluated only in response to the requirements of definition 13 and their 'recursive calls' on one another. For the same reason, the order of their evaluation (with various arguments) in transforming a specific valid procedure-free program is, to some extent, fixed. Accordingly, the reader may wish to regard them as specifying a mechanical translation procedure (dynamic), as well as a transformation (static) of strings to sets of quadruples.

The following definition is given out of order to provide better motivation. Logically, the backus-form grammar and the definition of the three functions described above are prior to it.

Definition 13: A procedure-free program is valid if and only if it is a terminal production of ⟨program⟩. If a ⟨program⟩ has ⟨statements⟩ L and ⟨declarations⟩ specifying variable set V and initial variable v_0, then it specifies the tm [resp. lba] with quadruples quad(L, 1, -1, 2) [3] whose states are (a subset of) $\{-1, 0, 1, \ldots, \text{next}(L, 2) - 1\} \times V$, whose initial state is $(1, v_0)$, and whose set of accepting states is $\{0\} \times V$.

⟨program⟩ → begin ⟨declarations⟩ ⟨statements⟩ end

⟨declarations⟩ → ⟨track declaration⟩ ⟨alphabet declaration⟩
⟨variable declarations⟩ ⟨track variable declaration⟩
⟨class declarations⟩

⟨statements⟩ → ⟨statement list⟩

⟨track declaration⟩ → tracks : ⟨unsigned integer⟩ ;

For the remainder of the present description, the value of ⟨unsigned integer⟩ in the ⟨track declaration⟩ will be referred to as k.

⟨alphabet declaration⟩ → alphabets : ⟨alphabets⟩ ;

⟨alphabets⟩ → ⟨repeated alphabet⟩ | ⟨alphabets⟩ ,
⟨repeated alphabet⟩

⟨repeated alphabet⟩ → ⟨set⟩ | ⟨unsigned integer⟩ × ⟨set⟩

---

3. The usefulness of '1' and '2' here should be clear from the informal description of quad. The '-1' is a technical device to permit quad to be applied even to strings including the end of the program. No quadruples corresponding to moves from states of the form (-1, v) are generated by definition 13, so the effect of the '-1' on control flow is that 'falling through' the program is a failure to accept the input.

The alphabet specified by the ⟨alphabet declaration⟩

$$\underline{\text{alphabets}} : i_1 \times (a_{11}, a_{12}, \ldots, a_{1j_1}), i_2 \times (a_{21}, a_{22}, \ldots, a_{2j_2}), \ldots, i_z \times (a_{z1}, a_{z2}, \ldots, a_{zj_z}) ;$$

is (If an 'i_e ×' is omitted, '1 ×' is assumed.) :

$$\{\text{'}\not{\text{c}}\text{'}, \text{'\$'}\} \cup (\{a_{11}, a_{12}, \ldots, a_{1j_1}\} \times \ldots \times \{a_{11}, a_{12}, \ldots, a_{1j_1}\}_{i_1 \text{ times}} \times \{a_{21}, a_{22}, \ldots, a_{2j_2}\} \times \ldots \times \{a_{21}, a_{22}, \ldots, a_{2j_2}\}_{i_2 \text{ times}} \times \ldots \times \{a_{z1}, a_{z2}, \ldots, a_{zj_z}\} \times \ldots \times \{a_{z1}, a_{z2}, \ldots, a_{zj_z}\}_{i_z \text{ times}}) .$$

It is required that $\sum_{e=1}^{z} i_e = k$. For the remainder of the present description, the alphabet specified will be referred to as $A = \{\text{'}\not{\text{c}}\text{'}, \text{'\$'}\} \cup (A_1 \times A_2 \times \ldots \times A_k)$.

⟨variable declarations⟩ → ⟨empty⟩ | ⟨variable declaration⟩ ⟨variable declarations⟩

⟨variable declaration⟩ → variable ⟨identifier⟩ ε ⟨set⟩ ;

⟨track variable declaration⟩ → track variable ⟨identifiers⟩ ;

The variable set specified by the ⟨variable declarations⟩ and ⟨track variable declaration⟩

$\underline{\text{variable}}\ \hat{x}_1, \hat{x}_2, \ldots, \hat{x}_{i_1} \varepsilon (a_{11}, a_{12}, \ldots, a_{1j_1})$;

$\underline{\text{variable}}\ \hat{x}_{i_1+1}, \hat{x}_{i_1+2}, \ldots, \hat{x}_{i_2} \varepsilon (a_{21}, a_{22}, \ldots, a_{2j_2})$; ... ;

$\underline{\text{variable}}\ \hat{x}_{i_{z-1}+1}, \hat{x}_{i_{z-1}+2}, \ldots, \hat{x}_{i_z} \varepsilon (a_{z1}, a_{z2}, \ldots, a_{zj_z})$;

$\underline{\text{track variable}}\ \hat{t}_1, \hat{t}_2, \ldots, \hat{t}_n$;

is:

$$\underbrace{\{a_{11}, a_{12}, \ldots, a_{1j_1}\} \times \ldots \times \{a_{11}, a_{12}, \ldots, a_{1j_1}\}}_{i_1 \text{ times}}$$
$$\times \underbrace{\{a_{21}, a_{22}, \ldots, a_{2j_2}\} \times \ldots \times \{a_{21}, a_{22}, \ldots, a_{2j_2}\}}_{i_2 \text{ times}} \times \ldots$$
$$\times \underbrace{\{a_{z1}, a_{z2}, \ldots, a_{zj_z}\} \times \ldots \times \{a_{z1}, a_{z2}, \ldots, a_{zj_z}\}}_{m = i_z \text{ times}}$$
$$\times \underbrace{\{1, 2, \ldots, k\} \times \ldots \times \{1, 2, \ldots, k\}}_{n \text{ times}} .$$

For the remainder of the present description, the variable set specified will be referred to as $\underline{V} = X_1 \times X_2 \times \ldots \times X_m \times \{1, 2, \ldots, k\}^n$, and the variables declared referred to as $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m, \hat{t}_1, \hat{t}_2, \ldots, \hat{t}_n$.

The initial variable specified by the same ⟨variable declarations⟩ and ⟨track variable declaration⟩ is:

$$(\underbrace{a_{11}, \ldots, a_{11}}_{i_1 \text{ times}}, \underbrace{a_{21}, \ldots, a_{21}}_{i_2 \text{ times}}, \ldots, \underbrace{a_{z1}, \ldots, a_{z1}}_{i_z \text{ times}}, \underbrace{1, \ldots, 1}_{n \text{ times}}) .$$

⟨class declarations⟩ → ⟨empty⟩ | ⟨class declaration⟩ ⟨class declarations⟩

⟨class declaration⟩ → class ⟨identifiers⟩ = ⟨set⟩ ;

The class set specified by the ⟨class declarations⟩

$$\underline{\text{class}}\ \hat{K}_1, \hat{K}_2, \ldots, \hat{K}_{i_1} = (a_{11}, a_{12}, \ldots, a_{1j_1})\ ;$$
$$\underline{\text{class}}\ \hat{K}_{i_1+1}, \hat{K}_{i_1+2}, \ldots, \hat{K}_{i_2} = (a_{21}, a_{22}, \ldots, a_{2j_2})\ ;\ \ldots\ ;$$
$$\underline{\text{class}}\ \hat{K}_{i_{z-1}+1}, \hat{K}_{i_{z-1}+2}, \ldots, \hat{K}_{i_z} = (a_{z1}, a_{z2}, \ldots, a_{zj_z})\ ;$$

is:

$$\underbrace{\{a_{11}, a_{12}, \ldots, a_{1j_1}\} \times \ldots \times \{a_{11}, a_{12}, \ldots, a_{1j_1}\}}_{i_1 \text{ times}}$$
$$\times \underbrace{\{a_{21}, a_{22}, \ldots, a_{2j_2}\} \times \ldots \times \{a_{21}, a_{22}, \ldots, a_{2j_2}\}}_{i_2 \text{ times}} \times \ldots$$
$$\times \underbrace{\{a_{z1}, a_{z2}, \ldots, a_{zj_z}\} \times \ldots \times \{a_{z1}, a_{z2}, \ldots, a_{zj_z}\}}_{m' = i_z \text{ times}}\ .$$

For the remainder of the present description, the class set specified will be referred to as $K_1 \times K_2 \times \ldots \times K_{m'}$, and the class variables declared referred to as $\hat{K}_1, \hat{K}_2, \ldots, \hat{K}_{m'}$.

⟨identifiers⟩ → ⟨identifier⟩ | ⟨identifier⟩ , ⟨identifiers⟩

No ⟨identifier⟩ is permitted to occur more than once in the ⟨declarations⟩.

⟨statement list⟩ → ⟨nondeterministic list⟩ |
⟨nondeterministic list⟩ , ⟨statement list⟩

If L_1 = N ',' L_2 , N is a ⟨nondeterministic list⟩, and L_2 is a ⟨statement list⟩, then

$$\underline{quad}(L_1, p, q, s) = \underline{quad}(N, p, s, s+1) \cup \underline{quad}(L_2, s, q, \underline{next}(N, s+1)) \quad \text{and}$$

$$\underline{next}(L_1, s) = \underline{next}(L_2, \underline{next}(N, s+1)) .$$

⟨nondeterministic list⟩ → ⟨statement⟩ |
⟨statement⟩ or ⟨nondeterministic list⟩

If N_1 = S 'or' N_2 , S is a ⟨statement⟩, and N_2 is a ⟨nondeterministic list⟩, then

$$\underline{quad}(N_1, p, q, s) = \{((p,v), a, a, (\hat{s},v)) \mid a \in \underline{A} \text{ and } v \in \underline{V} \text{ and } \hat{s} \in \{s, s+1\}\} \cup \underline{quad}(S, s, q, s+2) \cup \underline{quad}(N_2, s+1, q, next(S, s+2)) \quad \text{and}$$

$$\underline{next}(N_1, s) = \underline{next}(N_2, \underline{next}(S, s+2))$$

⟨statement⟩ → ⟨identifier⟩ : ⟨statement⟩ | (⟨statement list⟩) |
[⟨conditional tail⟩ | ⟨simple statement⟩

In the first case, the ⟨identifier⟩, which must not occur anywhere else in the ⟨program⟩ except in a ⟨jump⟩, is said to be a (valid) label.

If S_1 = I ':' S_2 , I is an ⟨identifier⟩, and S_2 is a ⟨statement list⟩, then

$$\underline{quad}(S_1, p, q, s) = \underline{quad}(S_2, p, q, s) \quad \text{and} \quad \underline{next}(S_1, s) = \underline{next}(S_2, s) .$$

In this case, any state (p, v) , $v \in \underline{V}$, is said to correspond to I .

If S = '(' L ')' and L is a ⟨statement list⟩, then

$\underline{quad}(S, p, q, s) = \underline{quad}(L, p, q, s)$ and $\underline{next}(S, s) = \underline{next}(L, s)$.

If S = '[' T and T is a ⟨conditional tail⟩, then

$\underline{quad}(S, p, q, s) = \underline{Cquad}(T, p, q, s, \emptyset)$ and
$\underline{next}(S, s) = \underline{next}(T, s)$.

⟨conditional tail⟩ →] | ; ⟨conditional tail⟩ |
⟨condition⟩ => ⟨statement list⟩ ⟨conditional tail⟩

$\underline{Cquad}(']', p, q, s, E) = \emptyset$ and $\underline{next}(']', s) = s$

If T_1 = ';' T_2 and T_2 is a ⟨conditional tail⟩, then

$\underline{Cquad}(T_1, p, q, s, E) = \underline{Cquad}(T_2, p, q, s, E)$ and
$\underline{next}(T_1, s) = \underline{next}(T_2, s)$.

If T_1 = C '=>' L T_2 , C is a ⟨condition⟩, L is a ⟨statement list⟩, and T_2 is a ⟨conditional tail⟩, then

$\underline{Cquad}(T_1, p, q, s, E) = \{((p,v), a, a, (s,v)) \mid a \in \underline{A}$ and
$v = (x_1, \ldots, x_m, t_1, \ldots, t_n) \in \underline{V}$ and $(a,v) \in D(C) \setminus E \}$
$\cup\ \underline{quad}(L, s, q, s+1)$
$\cup\ \underline{Cquad}(T_2, p, q, \underline{next}(L, s+1), E \cup D(C))$ and
$\underline{next}(T_1, s) = \underline{next}(T_2, \underline{next}(L, s+1))$, where
$D(C) = \{(a,v) \mid a \in \underline{A}$ and $v = (x_1, \ldots, x_m, t_1, \ldots, t_n) \in \underline{V}$
and $P_0(C) \}$ and P_0 is given below, in table 1.

⟨condition⟩ → ⟨identifier⟩ ⟨relation⟩ ⟨expression⟩ |

⟨track expression⟩ . ⟨class expression⟩ |

⟨whole tape symbol⟩ | ⟨empty⟩

In the first case: (i) If the ⟨identifier⟩ occurred in a ⟨track variable declaration⟩, then the ⟨relation⟩ must be '=' or '≠', and the ⟨expression⟩ a ⟨track expression⟩. (ii) If the ⟨identifier⟩ occurred in a ⟨variable declaration⟩, then: If the ⟨expression⟩ is not a ⟨symbol expression⟩, it must be a ⟨class expression⟩ and the ⟨relation⟩ must be 'ε' or 'ℰ'. If the ⟨expression⟩ is a ⟨symbol expression⟩, the ⟨relation⟩ must be '=' or '≠'. (i) or (ii) must be satisfied.

C	$\hat{t}_i = j$	$\hat{t}_i \neq j$	$\hat{t}_i = \hat{t}_j$	$\hat{t}_i \neq \hat{t}_j$	$\hat{x}_i \varepsilon \hat{K}_j$	$\hat{x}_i \notin \hat{K}_j$	$\hat{x}_i = c$
$P_0(C)$	$t_i = j$	$t_i \neq j$	$t_i = t_j$	$t_i \neq t_j$	$x_i \varepsilon K_j$	$x_i \notin K_j$	$x_i = c$

C	$\hat{x}_i \neq c$	$\hat{x}_i = \hat{x}_j$	$\hat{x}_i \neq \hat{x}_j$	$i.\hat{K}_j$	$i.c$	$i.\hat{x}_j$	$\hat{t}_i.\hat{K}_j$
$P_0(C)$	$x_i \neq c$	$x_i = x_j$	$x_i \neq x_j$	$a = (a_1, \ldots, a_k)$ and			
				$a_i \varepsilon K_j$	$a_i = c$	$a_i = x_j$	$a_{t_i} \varepsilon K_j$

C	$\hat{t}_i.c$	$\hat{t}_i.\hat{x}_j$	ℰ	\$	$(c_1, \ldots, c_k)$	◇
$P_0(C)$	$a = (a_1, \ldots, a_k)$ and		$a = $ 'ℰ'	$a = $ '\$'	$a = (c_1, \ldots, c_k)$	true
	$a_{t_i} = c$	$a_{t_i} = x_j$				

Table 1. (Quotation marks omitted.)

If S is a ⟨simple statement⟩, then

next(S, s) = s .

⟨simple statement⟩ → ⟨write⟩ | ⟨store⟩ | ⟨shift⟩ | ⟨jump⟩

⟨write⟩ → ⟨track expression⟩ . ⟨symbol expression⟩ |
⟨whole tape symbol⟩

If W is a ⟨write⟩, then

quad(W, p, q, s) = {((p,v), a, b, (q,v)) | {a, b} ⊂ $\underline{A}$ and v ε $\underline{V}$ and $P_1(W)$ } , where P_1 is given in table 2.

W	$P_1(W)$	
¢	b = '¢'	[and a ≠ '$']
$	b = '$'	[and a ≠ '¢']
$(c_1, \ldots, c_k)$	$b = (c_1, \ldots, c_k)$	[and a ∉ {'¢', '$'}]
i.c	$b_i = c$	and $a = (a_1, \ldots, a_k)$ and $b = (b_1, \ldots, b_k)$ and $\forall_{e=1}^{k}\ e \neq i \Rightarrow b_e = a_e$
$i.\hat{x}_j$	$b_i = x_j$	
$t_i.c$	$b_{t_i} = c$	
$t_i.\hat{x}_j$	$b_{t_i} = x_j$	

Table 2.

⟨store⟩ → ⟨identifier⟩ ← ⟨symbol expression⟩ | ⟨identifier⟩

← ⟨track expression⟩ . | ⟨identifier⟩ ← ⟨track expression⟩

In the first two cases, the ⟨identifier⟩ must have occurred in a ⟨variable declaration⟩. In the third case, the ⟨identifier⟩ must have occurred in a ⟨track variable declaration⟩.

If V is a ⟨store⟩, then

$\underline{quad}(V, p, q, s) = \{((p,(x_1,\ldots,x_m,t_1,\ldots,t_n)), a, a, (q,(y_1,\ldots y_m,u_1,\ldots,u_n))) \mid a \in \underline{A}$ and $\forall_{e=1}^{m}\{x_e,y_e\} \subset X_e$ and $\forall_{e=1}^{n}\{t_e,u_e\} \subset \{1,\ldots,k\}$ and $P_2(V)\}$, where P_2 is given in table 3.

V	$P_2(V)$	
$\hat{x}_i \leftarrow c$	$y_i = c$	
$\hat{x}_i \leftarrow \hat{x}_j$	$y_i = x_j$	and $\forall_{e=1}^{m}\ e \neq i \Rightarrow y_e = x_e$
$\hat{x}_i \leftarrow j.$	$a = (a_1, \ldots, a_k)$ and $y_i = a_j$	and $\forall_{e=1}^{n}\ u_e = t_e$
$\hat{x}_i \leftarrow \hat{t}_j.$	$a = (a_1, \ldots, a_k)$ and $y_i = a_{t_j}$	
$\hat{t}_i \leftarrow j$	$u_i = j$	and $\forall_{e=1}^{m} y_e = x_e$ and $\forall_{e=1}^{n}\ e \neq i \Rightarrow u_e = t_e$
$\hat{t}_i \leftarrow \hat{t}_j$	$u_i = t_j$	

Table 3.

⟨shift⟩ → $\underline{L}$ | $\underline{R}$

If M is a ⟨shift⟩, then

$\underline{quad}(M, p, q, s) = \{((p,v), a, M, (q,v)) \mid a \in \underline{A}$ and $v \in \underline{V}$ [and $M = \text{'}\underline{L}\text{'} \Rightarrow a \neq \text{'}\not\in\text{'}$ and $M = \text{'}\underline{R}\text{'} \Rightarrow a \neq \text{'}\$\text{'}$] $\}$.

⟨jump⟩ → ⟨empty⟩ | accept | go to ⟨identifier⟩ | repeat

The ⟨identifier⟩ following 'go to' must be a valid label. (See syntax for ⟨statement⟩.)

'repeat' must lie within a ⟨conditional statement⟩.

If J is a ⟨jump⟩, then

quad(J, p, q, s) = {((p,v), a, a, (r,v)) | a ε A and v ε V } ,

where:

If J = ◇ , then r = q .

If J = 'accept' , then r = 0 .

If J = 'go to' , then necessarily, in the evaluation of the function quad applied to the program, the function quad is applied exactly once with first argument I ':' S for some ⟨simple statement⟩ S . r is the second argument of quad in that application.

If J = 'repeat' , then necessarily J lies in some ⟨conditional tail⟩. Let T be the shortest ⟨conditional tail⟩ including (the present instance of) J . In the evaluation of the function quad applied to the program, the function Cquad is applied with first argument T . r is the second argument of Cquad in its application to (the present instance of) T..

⟨expression⟩ → ⟨class expression⟩ | ⟨track expression⟩

⟨class expression⟩ → ⟨identifier⟩ | ⟨symbol expression⟩

If the ⟨identifier⟩ is not a ⟨symbol expression⟩, it must have occurred in a ⟨class declaration⟩.

⟨symbol expression⟩ → ⟨symbol⟩ | ⟨identifier⟩

The ⟨identifier⟩ must have occurred in a ⟨variable declaration⟩.

⟨track expression⟩ → ⟨unsigned integer⟩ | ⟨identifier⟩

The ⟨unsigned integer⟩ must have a value ε {1, ..., k} . The ⟨identifier⟩must have occurred in a ⟨track variable declaration⟩.

⟨whole tape symbol⟩ → ¢ | $ | ⟨set⟩

The ⟨set⟩ must have k components.

⟨relation⟩ → = | ≠ | ε | ∉

⟨set⟩ → (⟨symbols⟩)

⟨symbols⟩ → ⟨symbol⟩ | ⟨symbol⟩ , ⟨symbols⟩

The syntax for ⟨symbol⟩ varies with the application.

The syntax for ⟨identifier⟩, ⟨unsigned integer⟩, and ⟨empty⟩ is usual.

Figures 5 and 6 illustrate the application of definition 13 to the procedure-free program of figure 4. In figure 5, the program is marked so as to show the program positions corresponding to the first component of the tm [resp. lba]states. Figure 6 shows the quadruples of which the tm [resp. lba] corresponding to figure 5 is composed, tabulated as a 'move function'. Entries marked '[]' are to be included [resp. omitted]. The table is abbreviated, showing the second (variable set) component of a state only where pertinent, and omitting tape symbols in the body of the table unless they indicate a change in tape contents.

	1	2	3	4	5	6	7	8	9	10	11
a_	2,R	4	16	5,a+	6,R	7 or 8	2	10	3,R	11,L	8
a+	2,R	4	13	5	6,R	7 or 8	2	10	3,R	11,L	8
b_	2,R	4	16	5,b+	6,R	7 or 8	2	10	3,R	11,L	8
b+	2,R	4	13	5	6,R	7 or 8	2	10	3,R	11,L	8
c_	2,R	4	16	5,c+	6,R	7 or 8	2	10	3,R	11,L	8
c+	2,R	4	13	5	6,R	7 or 8	2	10	3,R	11,L	8
¢	2,R	4	16		6,R	7 or 8	2	9	3,R	11,L[]	8
$	2,R[]	4	16		6,R[]	7 or 8	2	10	3,R[]	11,L	8

	12	13a b c	14	15	16	17	18a b c	19	20a b c
a_		15	3	14,R	12	24,L	19	0	21
a+		15	3	14,R	12	24,L	19	0	21
b_	20	15	3	14,R	12	24,L	19	0	21
b+	20	15	3	14,R	12	24,L	19	0	21
c_	22	15	3	14,R	12	24,L	19	0	21
c+	22	15	3	14,R	12	24,L	19	0	21
¢			3	14,R	12	24,L[]	19	0	21
$	18		3	14,R[]	12	24,L	19	0	21

	21	22a b c	23	24a b c	25	26	27	28	29	30
a_	17b	23	17c	25 26 26	3	28	24	29,a+	32	35
a+	17b	23	17c	25 26 26	3	36	24	29	33	35,a_
b_	17b	23	17c	26 25 26	3	28	24	29,b+	32	35
b+	17b	23	17c	26 25 26	3	36	24	29	33	35,b_
c_	17b	23	17c	26 26 25	3	28	24	29,c+	32	35
c+	17b	23	17c	26 26 25	3	36	24	29	33	35,c_
¢	17b	23	17c	26 26 26	3	36	24		31	
$	17b	23	17c	26 26 26	3	36	24		33	

Figure 6. (page 1/2)

```
begin
tracks: 2; alphabets: (a, b, c), (_, +);
variable x ε (a, b, c);
⟨1⟩ R, ⟨2⟩ [ => ⟨4⟩ 2.+, ⟨5⟩ R, ⟨6⟩ ⟨7⟩ repeat or ⟨8⟩ [ ∉ => ⟨9⟩ R ;
                                                    =>⟨10⟩ L, ⟨11⟩ repeat ] ],
⟨3⟩ S: [ 2.+ => ⟨13⟩ [ 1.x => ⟨15⟩ R ], ⟨14⟩ repeat ; => ⟨16⟩ ],
⟨12⟩ [ $ => ⟨18⟩ [ x = c => ⟨19⟩ accept ] ;
     1.b => ⟨20⟩ [ x = a => ⟨21⟩ x <- b ] ;
     1.c => ⟨22⟩ [ x = b => ⟨23⟩ x <- c ] ], ⟨17⟩ L,
⟨24⟩ [ 1.x => ⟨25⟩ go to S ; => ⟨26⟩ [ 2._ => ⟨28⟩ 2.+, ⟨29⟩ [ ∉ => ⟨31⟩ R ;
                                                    2._ => ⟨32⟩ R ;
                                                => ⟨33⟩ L, ⟨34⟩ repeat ],
                                                    ⟨30⟩ 2._, ⟨35⟩ R ;
                                            => ⟨36⟩ R, ⟨37⟩ repeat ],
                                                    ⟨27⟩ repeat ]
⟨-1⟩
end
```

Figure 5.

	31	32	33	34	35	36	37
a_	30,R	30,R	34,L	29	27,R	37,R	26
a+	30,R	30,R	34,L	29	27,R	37,R	26
b_	30,R	30,R	34,L	29	27,R	37,R	26
b+	30,R	30,R	34,L	29	27,R	37,R	26
c_	30,R	30,R	34,L	29	27,R	37,R	26
c+	30,R	30,R	34,L	29	27,R	37,R	26
¢	30,R	30,R	34,L[]	29	27,R	37,R	26
$	30,R[]	30,R[]	34,L	29	27,R[]	37,R[]	26

Figure 6. (page 2/2)

the program schema

The present section consists mainly of a program schema depending on $G = (M, N, R_M, B, R, s)$, a vWg, in the language of the preceding sections.

Even for fixed G , the ultimate transformations of the schema into tm and lba will differ. Accordingly, as a schema, it has two sets of properties. Matter (in the commentary) enclosed in square brackets refers to its properties as an lba-language program schema.

Variations on 'y is pushed right on track t to make room for $x \mapsto \hat{x}$' occur in the commentary. Their meaning is: 'If the contents of track t is $xy(\_)^k$ and $|\hat{x}| \geq |x|$, then the present action is [to fail if $|\hat{x}| - |x| > k$, otherwise] to place $\hat{x}y(\_)^{\hat{k}}$ on track t , where $\hat{k} = \max\{0, k - (|\hat{x}| - |x|)\}$'.

Some other notations used in this and the following section are:

T.i denotes (as in preceding sections) the string on track i left of the leftmost occurrence on track i of '_' .

If x is(declared to be) a variable, then (x) denotes its value.

The notations c and I (as at (3)) are retained.

begin

tracks: 6 ;

alphabets: 6 × ({ a | a ε {_, +} ∪ M ∪ N or a = , | , }) ;

variable x1, x2 ε ({ a | a ε {_, +} ∪ M ∪ N or a = , | , }) ;

variable x3 ε ({ a | a ε M | , }) ;

track variable t ;

class M = ({ a | a ε M | , }) ;

```
procedure rewind;
    <Positions tape scan just right of '¢'.>
    begin [ ¢ => R ; => L, repeat ] end rewind;
procedure wind;
    <Positions tape scan just left of '$' end-mark.>
    begin [ $ => L ; => R, repeat ] end wind;
procedure clear(track);
    <Fills track with blanks.>
    begin
    rewind, [ $ => ; => track._, R, repeat ]
    end clear;
procedure endtape;
    <If entered scanning '$', writes a six-track blank, otherwise fails.
    Recall that '$' is the natural tape blank to see this adds one all-
    blank square at the right end of information if the '$' scanned is
    the right end-mark.  [Resp. always fails.]>
    begin [ $ => (_, _, _, _, _, _) ] end endtape;
{procedure check_x;
    <Fails unless the string  x  is on track 1, just right of the scan
    point, in which case tape contents are unchanged.>
    begin
    { [ 1.a_i => R ] | x = a_1...a_k  and  i ε {1,...,k} | , }
    end check_x
| x ε B ∪ {s} | ; } ;
```

```
procedure check;
    ⟨Fails unless the contents of track 1 are  c(X)  for some  X ε B* ,
    in which case tape contents are unchanged.⟩
    begin
    rewind,
    [ $ => ; => { check_x | x ε B | or } , [ 1., => R ], repeat ]
    end check;
procedure test;
    ⟨Accepts if  T.1 = 's','  , otherwise does not affect tape contents.⟩
    begin
    rewind,
    check_s, [ 1., => R ], [ $ => accept or ; 1._ => R, repeat ; => ]
    end test;
{procedure enter_x(track);
    ⟨If the specified track has a string of  |x|  blanks just right of
    the scan point, this enters the string  x  there.  It suffices
    [resp. does not suffice] for the specified track to have only blanks
    between the scan point and '$' end-mark.⟩
    begin
    { [ track._ => track.a_i, R ; => endtape, repeat ]
    | x = a_1...a_k  and  i ε {1,...k} | , }
    end enter_x
| x = ,  or  ∃_{yε(1UN)*} ((x → y) ε R_M  or (y → x) ε R_M)
        or  ∃_{rεR} ∃_{iε{0,...,|r|-1}}  x = r_i | ; } ;
```

```
{procedure load_r;
     <Sets  T.2 = r_1',' ... r_|r|-1',' and T.3 = r_0',' [space permitting].>
     begin
     clear(2), clear(3),
     rewind, { enter_r_i(2), enter_,(2) | i ε {1,...,|r|-1} | , } ,
     rewind, enter_r_0(3), enter_,(3)
     end load_r
| r ε R | ; } ;
procedure fwdsrc(class, fail);
     <Jumps to fail if no member of class occurs on tracks 2 or 3, otherwise positions tape scan and sets track variable t so that the symbol scanned on track (t) is the leftmost (with track 2 left of track 3) occurrence of a member of class on tracks 2 or 3.>
     begin
     t <- 2, rewind,
     [ t.class => ; $ => [ t = 3 => go to fail ;
                           t = 2 => t <- 3, rewind ], repeat ;
                      => R, repeat ]
     end fwdsrc;
procedure bwdins(stop, track, insert);
     <If there is an occurrence of stop on the tape, this places the symbol insert on the specified track just right of the rightmost occurrence of stop and leaves the scan point there, having pushed that track right to make room for the insertion.>
```

```
        begin
        wind,
        [ track._ => L, [ stop => R, track.insert, go to B ;
                                => xl <- track.  , track._, R, track.xl, L ],
                                                                repeat ;
                  => R, endtape, repeat ], B:
        end bwdins;
    procedure findall(class, action);
        <Repeatedly performs action, each time having positioned the scan
        point and set track variable  t  so that the symbol scanned on
        track  (t)  is the leftmost (with track 2 left of track 3) occurrence
        of a member of class on tracks 2 or 3.>
        begin
        [ => fwdsrc(class, F), action, repeat ], F:
        end findall;
{procedure metaload_r;
        <Sets  T.5 = x  and  T.6 = y , where  r = (x → y) [space permitting].>
        begin
        clear(5), clear(6),
        rewind, enter_x(5),
        rewind, enter_y(6)
        end metaload_r
| (x → y) = r ε R_M | ; } ;
```

```
procedure shift(mtc, src);
    ⟨Pushes all of tracks mtc and src right to make room for a string of
    blanks on each.⟩
    begin
    [ => (bwdins(¢, mtc, _), bwdins(¢, src, _), repeat) or ]
    end shift;
procedure setscan(src);
    ⟨Sets the scan point to the leftmost non-blank square on track src.⟩
    begin rewind, [ src._ => R, repeat ; => ] end setscan;
procedure subs(dst, mtc, src);
    ⟨If the portion of track dst right of the scan point is  ux , the
    portion of track mtc right of the scan point is  uy , the portion of
    track  src right of the scan point is  vz , each of  y  and  z  is
    either empty or begins with a blank, and  u  and  v  include no
    blanks, then this pushes  x  right to make room for the substitution
    of  v(+)^i  for  u , where  i = max { 0, |u| - |v| } . ⟩
    begin
    [ $ => ;
    mtc._ => [ src._ => ;
                    => x2 <- src. , L, [ ¢ => bwdins(¢, dst, x2) ;
                                       => src.+, bwdins(src.+, dst, x2) ],
                                    R, repeat ] ;
    {mtc.a => [ dst.a => [ src._ => dst.+ ;
                        {src.b => dst.b | b ε M ∪ N ∪ {,} | ; } ] ], R, repeat
    | a ε M ∪ N ∪ {,} | ; } ]
    end subs;
```

```
procedure apply2;
    ⟨Nondeterministically: fails or, if T.4 = u T.5 v , replaces T.5
    with T.6 in T.4 , pushing v right to make room.⟩
    begin shift(5, 6), setscan(6), subs(4, 5, 6) end apply2;
procedure metagen;
    ⟨Nondeterministically sets T.4 ε L(M, N, R_M, (x3)) [space permitting]
    or fails. See lemma 2.⟩
    begin
    M0: clear(4), rewind, 4.x3,
    [ => rewind, M1: [ 4.M => ; 4._ => go to M5 or ; $ => go to M5 or ;
                                    => R, repeat ],
        M2: {metaload_r | r ε R_M | or} , M3: apply2, M4: repeat ], M5:
    end metagen;
procedure copy;
    ⟨Pushes the portion of track (t) right of the symbol initially
    scanned right to permit the substitution of T.4 for the symbol
    initially scanned. Fails if T.4 = ◇ .⟩
    begin
    t.+, x2 ← _ ,
    [ => rewind,
        [ 4._ => 4.x2, C1: L, x2 ← 4. , 4._, L,
                [ ∉ => R, 4.x2, [ t.+ => t.x2 ;
                                    => R, repeat ], go to C2 ;
                    => bwdins(t.+, t, x2) ] ;
            $ => go to C1 ;
              => R, repeat ], repeat ], C2:
    end copy;
```

```
procedure generate;
    ⟨Nondeterministically: fails or, for each u ε M , replaces each
    occurrence of u on tracks 2 and 3 with some fixed element of
    L(M, N, R_M, u), pushing those tracks right as necessary.⟩
    begin findall(M, (x3 ← t. , metagen, findall(x3, copy))) end generate;
procedure smash;
    ⟨Repeatedly, if T.1 = u '+' v and u includes no '+'s , this
    sets T.1 = uv . Hence this removes all '+'s from track 1.⟩
    begin
    rewind,
    [ $ => ;
    1.+ => R, [ $ => L, 1._ ;
                => x1 ← 1. , L, 1.x1, R, R, repeat ], rewind, repeat ;
        => R, repeat ]
    end smash;
procedure apply1;
    ⟨Nondeterministically: fails or, if T.1 = u T.2 v , and u = ◇
    or u ends with ',', replaces T.2 with T.3 in T.1 , pushing
    v right to make room.⟩
    begin
    shift(2, 3), setscan(3), L, [ 1., => ; ¢ => ], R, subs(1, 2, 3), smash
    end apply1;
⟨main line of program (See lemmata 2 and 3.)⟩
P0: check, [ => P1: test, P2: {load_r | r ε R | or}, P3: generate, P4: apply1,
            P5: repeat ]
end
```

properties of the program schema

It remains to be argued that, for given [cs] vWg $G = (M, N, R_M, B, R, s)$, the tm [resp. lba] A corresponding to G specified by the program schema of the preceding section has the property (4). It is not the intent of the present paper to give a rigorous proof of this fact, or even to develop the machinery to do so. The proofs of the propositions of the present section depend on the accuracy and completeness of the descriptions given as commentary in the program schema.

Two remarks are in order concerning the commentary given in the program schema: First, each assertion like

(6) 'This procedure affects the tape in manner F.'

may, in view of its role in a formal proof, be considered as an abbreviation for one like

(7) 'If the state part of an instantaneous description I_1 of A corresponds to the initial position of an instance of this procedure, and the state part of an instantaneous description I_2 of A corresponds to the final position of the same instance of this procedure, and the state parts of I_1, I_2 do not differ otherwise, then there is a sequence of computational steps of A leading from I_1 to I_2 if and only if the tape part of I_2 can be derived from the tape part of I_1 in manner F .' ,

and similarly for comments describing jumps, acceptance, rejection, and

the assignment of values to variables. Second, except possibly in the case where a lemma is provided here, the reader should not find it difficult to convince himself of the truth of the assertions so abbreviated, since the programming techniques used are not complicated.

In accordance with the general intent of the present paper and, in particular, with the fact that the translation (6) ↦ (7) is not formally specified, the language of the present section will have formality between that of (6) and that of (7). Instantaneous descriptions will be (partially) specified as:

(8) $\quad L(C)$,

where 'L' is a label occurring in the program schema and C is a condition on tape contents and values of variables. Each occurrence of the notation (8) in an assertion may be taken to abbreviate some complete instantaneous description whose state satisfies the condition C and corresponds to the label 'L' , and whose tape contents satisfy the condition C . The existence of a sequence of computational steps of A leading from one instantaneous description, $L_1(C_1)$ to another, $L_2(C_2)$ is asserted thus:

$$L_1(C_1) \vdash L_2(C_2) \quad .$$

In the remainder of the present section, let $G = (H, M, R_M, B, R, s)$ be a fixed [cs] vWg satisfying (5). Lemmata 2, 3, and 4 refer to the tm [resp. lba] A corresponding to G specified by the program schema of the preceding section.

Lemma 2: For $U \in M$, let $G_U = (M, N, R_M, U)$. In the procedure metagen, $M0((x3) = U) \vdash M5(T.4 = x)$ if and only if $x \in L(G_U)$ [and $|x| \leq |X|$, where X is the input to A].

Proof: The conditional statement at M1 assures that if $M1(T.4 = x) \vdash M5$, then $x \in N^*$. Conversely, the same conditional statement permits $M1(T.4 = x) \vdash M5$. In each case, the passage $M1 \vdash M5$ entails no change in tape contents. Accordingly, it suffices to show $M0((x3) = U) \vdash M1(T.4 = x)$ if and only if $U \overset{*}{\underset{G_U}{\Rightarrow}} x$ [and $|x| \leq |X|$].

($U \overset{*}{\underset{G_U}{\Rightarrow}} x$ [and $|x| \leq |X|$] $\Rightarrow M0((x3) = U) \vdash M1(T.4 = x)$): By induction on the length of a derivation $U \overset{*}{\underset{G_U}{\Rightarrow}} x$: If length is 0, then $x = U$. Clearly, $M0((x3) = U) \vdash M1(T.4 = U)$ [and $|U| = 1 \leq |X|$ unless $X = \Diamond$, in which case ($U \overset{*}{\underset{G_U}{\Rightarrow}} x$ and $|x| \leq |X|$) is impossible by the assumption that G satisfies (5).] If length is > 0, then $\exists_{y \in (M \cup N)^*}$ $U \overset{*}{\underset{G_U}{\Rightarrow}} y$ and $y \Rightarrow x$ via some $(\hat{y} \to \hat{x}) \in R_M$. By inductive hypothesis [($|y| \leq |x| \leq |X|$)], $M0((x3) = U) \vdash M1(T.4 = y)$. Always $M1 \vdash M2$ with no change in tape contents, so $M1(T.4 = y) \vdash M2(T.4 = y)$. [$|\hat{y}| \leq |\hat{x}| \leq |x| \leq |X|$, so] $M2(T.4 = y) \vdash M3(T.4 = y$ and $T.5 = \hat{y}$ and $T.6 = \hat{x})$ via $\text{metaload}_{(\hat{x} \to \hat{y})}$. Finally, $M3(T.4 = y$ and $T.5 = \hat{y}$ and $T.6 = \hat{x}) \vdash M1(T.4 = x)$ via a suitable shift during application of apply2.

($M0((x3) = U) \vdash M1(T.4 = x) \Rightarrow U \overset{*}{\underset{G_U}{\Rightarrow}} x$ [and $|x| \leq |X|$]): By induction on the smallest number of occurrences of $M4()$ states in a computational sequence $M0((x3) = U) \vdash M1(T.4 = x)$: If smallest number of occurrences is 0, then $x = U$. If smallest number of occurrences is

> 0 , then $\exists_{y \in (M \cup N)^*}$ M0((x3) = U) |- M1(T.4 = y) and M1(T.4 = y) |- M1(T.4 = x) with exactly one occurrence of an M4() state. By inductive hypothesis, $U \overset{*}{\underset{G_U}{\Rightarrow}} y$. The necessary passage M3 |- M1 assures that an applicable rule was selected at M2 , so $y \overset{}{\underset{G_U}{\Rightarrow}} x$. [In any case, arrival at M1(T.4 = x) insures $|x| \leq |X|$.]

Lemma 3: In the main line of the program, if $Z \in (N^*)^*$, then P1(T.1 = x) |- P1(T.1 = c(Z)) if and only if $\exists_{X \in (N^*)^*}$ x = c(X) and $Z \overset{*}{\underset{G}{\Rightarrow}} X$.

Proof: (($\exists_{X \in (N^*)^*}$ x = c(X) and $Z \overset{*}{\Rightarrow} X$) => P1(T.1 = x) |- P1(T.1 = c(Z))) : By induction on the length of a derivation $Z \overset{*}{\underset{G}{\Rightarrow}} X$: If length is 0 , then X = Z , x = c(Z) . If length is > 0 , then $\exists_{Y \in (N^*)^*}$ $Z \overset{*}{\underset{G}{\Rightarrow}} Y$ and $Y \overset{}{\underset{G}{\Rightarrow}} X$ <u>via</u> some strict rule $(Y_1 \rightarrow X_1)$ derived from some $(Y_2 \rightarrow X_2) \in R$. [Because G satisfies (5), $|c(Y_2)| \leq |c(Y_1)|$ and $|c(X_2)| \leq |c(X_1)|$. Because G is cs, $|c(Y_1)| \leq |c(X_1)|$. Hence, by definition of '=>', $|c(Y_1)| \leq |c(Y)|$, $|c(X_1)| \leq |c(X)|$, and $|c(Y)| \leq |c(X)|$. Therefore space permits representation of all of Y, Y_1, Y_2, X_1, X_2 .] Now P1(T.1 = x) |- P2(T.1 = x) |- P3(T.1 = x and T.2 = $c(X_2)$ and T.3 = $c(Y_2)$) |- P4(T.1 = x and T.2 = $c(X_1)$ and T.3 = $c(Y_1)$) |- P1(T.1 = c(Y)) by stated properties of procedures (with appropriate choices). By inductive hypothesis, P1(T.1 = c(Y)) |- P1(T.1 = c(Z)) .

(P1(T.1 = x) |- P1(T.1 = c(Z)) => ($\exists_{X \in (N^*)^*}$ x = c(X) and $Z \overset{*}{\underset{G}{\Rightarrow}} X$)) : By induction on the smallest number of occurrences of P5() states in a computational sequence P1(T.1 = x) |- P1(T.1 = c(Z)) : If smallest number of occurrences is 0 , then x = c(Z) . If smallest number is

references

Baker, J. L. (1970). The syntax of algol 68, property grammars, and context-sensitive languages. Unpublished. (Submitted to Information and Control.)

Hopcroft, J. E. and J. D. Ullman (1969). Formal languages and their relation to automata. Addison-Wesley. Reading, Mass.

Knuth, D. E. and R. H. Bigelow (1967). Programming languages for automata. J ACM 14 , 615-635.