J Comput Virol (2009) 5:9–87
DOI 10.1007/s11416-008-0115-z

ORIGINAL PAPER

On self-reproducing computer programs

Master Thesis in Computer Science University of Dortmund

Jürgen Kraus
Hagen, Germany

Contents

Electronic supplementary material The online version of this article (doi:10.1007/s11416-008-0115-z) contains supplementary material, which is available to authorized users.

This thesis has been translated from the German and edited by Daniel Bilar and Eric Filiol with the kind permission of Jürgen Kraus.

D. Bilar
University of New Orleans, New Orleans, USA
e-mail: dbilar@uno.edu

E. Filiol (✉)
ESIEA, Laval, France
e-mail: efiliol@wanadoo.fr; filiol@esiea.fr

1 Introduction

1.1 Motivation

Life in copious forms abounds on Earth. Some lifeforms have remained unchanged over millennia, while others have died out. Differences in fate and appearance non-withstanding, they are all made up of highly complex biochemical units called *cells* which are the building blocks of life. The first cell, the starting point of biological evolution, was itself a product of Earth's preceding chemical evolution. How the cell came to be in the first place is an interesting question that ultimately can be answered probabilistically in vitro [11]. Furthermore, the existence and evolution of life itself may be the inevitable consequence of the complex conditions of Earth's early years. If this is true—if complexity gives rise to life—then it may be possible for other complex "worlds" to bear life,[1] or at the very least offer a real possibility to support certain life forms.

Computer technology has made immense strides in the last two decades. Newer and ever-more-powerful electronic elements enable the construction of digital mainframes whose capabilities would have been considered utopian just a few years ago. These mainframe capabilities can be further augmented by clustering and local to wide-area networking [21], yielding a system that is barely tractable for the user; in fact, helper computers have been proposed to administer these networks. Hence, there exist computing systems today that resemble a universe, comprised of integrated circuits and bits, whose complexity is reminiscent of the conditions in Earth's early years. Again, if complexity gives rise to life, it is possible to speculate about life existing on and/or arising from computer systems. The only guideline we have as to how this kind of life would look like is biological life, so far the only known life. Earlier on, we introduced the cell as life's building block. Without foreshadowing Chap. 7 too much, we shall highlight two distinctive characteristics of a living cell: *autoreproduction* and *mutation*, the former denoting autonomous flawless cell reproduction; the latter, error-prone cell reproduction. The electronic cell-equivalent in computer systems are *self-reproducing programs*, which we define in Sect. 1.2 as programs which can modify their code autonomously at runtime, without any external "blueprints". Since computer systems exhibit a small but non-zero error rate, there inherently exists the possibility of faulty reproduction, i.e. mutations. Hence, we consider self-reproducing programs to be contenders for the existence of lifeforms on computer systems.

The purpose of this thesis is not only to offer an existence proof of self-reproducing programs (see Chap. 2), but to construct concrete examples in various programming languages (Chaps. 3 and 6) and study their properties (Chaps. 4 and 5). Reproduction and mutation are two key evolutionary traits. Selection constitutes the third component enabling evolution. The magnificent cornucopia of species on Earth emanates from evolutionary processes; applied to self-reproducing programs, we should expect to encounter new programs with wildly different properties. Prior to introducing some models to evolve self-reproducing programs in Chaps. 8 and 9, however, we shall examine in Chap. 7 how apt the analogy between self-reproducing programs in a "computing" environment and biological living cells truly is.

1.2 Towards a definition of self-reproducing programs

Since this thesis strives to analyze programs and the programming languages in which they are written, it behooves us to give a precise definition of program *syntax*, the rules of writing a program, and program *semantics*, its concrete interpretation on a concrete computer system [9,14]. An in-depth discussion is surely outside the scope of this text, yet we shall give at least a rudimentary exposition using the abstract programming language `PL` in Chap. 2. We shall list the appropriate syntax references when we present concrete programming languages. Furthermore, for the case of program semantics, we shall prove as sufficient to refer to the program's realized function (see Sect. 5.1). Lastly, we pre-suppose an adequate familiarity with standard computer science programming languages and constructs.

Concrete programming languages may be categorized as either assembly or high-level languages. In the context of our discussion of self-reproducing programs, we list the relevant characteristics.

Assembly languages Assembly languages are machine-architecture dependent. Hence, several architectural elements are implied by its concomitant assembly language, among

[1] The definition of "Life" is contentious and will be discussed thoroughly in Chap. 7.

Table 1 Language differences

High-level languages	Assembly languages
⇓	⇓
Programs cannot read memory	Programs can read memory and hence their own machine code
⇓	⇓
Self-reproduction ≡ High-level language source code generation	Self-reproduction ≡ generation of faithful machine code copy
⇓	⇓
Translation required	No translation of copy required

them the memory layout, accessible by the program. Since executable assembly code also resides in memory, assembly programs can refer to and interpret their constitutive machine code.

High-level languages These languages are removed from the machine architecture, and as such architecture-independent. As a corollary, programs written in high level languages cannot directly access memory. As such, they lack the ability to read from or interpret their memory-resident machine code.

We shall equivalently refer to programs and their realized functions. In the context of this thesis, we emphasize an additional aspect: *Programs are finite character strings, i.e. texts*. Throughout its interpretative lifetime in the computer system, the same program may take the form of different texts over various alphabets. The textual representation of an assembly language program differs from its translated machine code: While assembly languages use an alphanumeric representation, machine code is restricted to 16 hexadecimal (`0-F`) characters. High-level languages are similar in that they potentially introduce one or more additional intermediate text representations between the source and the machine code.

The differences between assembly and high-level languages are reflected in the definition of self-reproducing programs in the respective languages.

Let S be a high level programming language. Then

Definition 1.1 Let π be a syntactically correct program in S.

1. If there is no input to π, we denote π as (strictly) self-reproducing if π reproduces (exactly) its program text in S.
2. If there is input to π, we denote π as (strictly) self-reproducing if, given any input, π reproduces (exactly) its program text in S.

Hence, Definition 1.1 allows self-reproducing programs to receive input. However, it stipulates that no input information may be used for reproduction; in fact, self-reproduction requires input independence.

Let M be a architecture-dependent assembly language. The following definition is informed by the ability of assembly language to read its own machine code.

Definition 1.2 Let π be a syntactically correct program in M.

1. If there is no input to π, we denote π as (strictly) self-reproducing if π can reproduce its machine code (exactly) or copy its machine code (exactly) in memory.
2. If there is input to π, we denote π as (strictly) self-reproducing if, given any input, π can reproduce its machine code (exactly) or copy its machine code (exactly) in memory.

In contrast to higher-level languages, the copies of self-reproducing assembly language programs do not have to be translated into machine code prior to execution. Table 1 summarizes the differences in this self-reproducing context.

Since assembly programs can read their own machine code, it is relatively straightforward - given some basic assembly language knowledge—to posit the existence of self-reproducing programs and to subsequently construct concrete examples (see Sect. 3.4). In higher-level languages, however, both existence of and construction procedures for self-reproducing programs are far from intuitive. In Chap. 2, we shall offer a rigorous theoretical existence proof. Concrete examples pose a challenge, as well; in general, circumstances are more dire in higher-level languages. As such, we shall almost exclusively focus on self-reproducing programs in high-level languages in Chaps. 3, 4 and 5.

2 Existence of self-reproducing programs

2.1 Introduction

In this chapter, we set out to prove the theoretical existence of self-reproducing programs written in high level programming languages. However, we will not base our argument on the many features of real programming languages such as PASCAL, SIMULA, ALGOL, and the like. Rather, we will define and use – as far as it is possible in this context – our own simple programming language. Its distinguishing features are particularly simple data types, as well as the implementation of programming constructs usually found in any high level programming languages. We will call this programming language the PL language. Its seeming simplicity will not

vitiate its power; PL will boast the same "computability capabilities" as any other common programming language. PL will prove its worth as a very suitable entry-level tool to the theory of "computable functions". We emphasize that we shall draw from aforementioned theory only insofar it is necessary and useful to prove the existence of self-reproducing PL programs.

Since other common programming languages have the same "computability capabilities" as the PL language, the existence of self-reproducing programs written in PL implies the existence of self-reproducing programs in other common programming languages—high level languages, as well as assembly languages.

2.2 Definition of the programming language PL(A)

In the first instance, we lay the foundation through an arbitrary, but fixed alphabet $A = \{a_1, a_2, \ldots, a_n\}$ with $n \in \mathbb{N}$. The set A^* of all words built on A represents the *data space* of the PL language. We consider the empty word $\epsilon \in A^*$ a datum, as well.

Definition 2.1 (Expressions)

1. The *constants* in PL are the elements of A^*.
2. The *variables* $X_1, X_2, \ldots, Y, Z, W$ are the elements of a fixed set VR. Any variable may take any value in A^*.
3. The *operations* are Xa and $\rho(X)$ with $X \in VR$ and $a \in A$.
   Meaning: Xa equals xa whenever $x \in A^*$ equals X. $\rho(X)$ equals $x \in A^*$, whenever X equals xa for a given element $a \in A$. Otherwise $\rho(X) = \epsilon$.
4. The *conditions* are of the form $\omega(X) = a$ or $X = \epsilon$ with $X \in VR$ and $a \in A$.
   Meaning: $\omega(X) = a$ is true whenever the ending letter of the word value of X is the letter a. $X = \epsilon$ is true whenever X equals ϵ.

Definition 2.2 (Basic instructions)

The *basic instructions* of the PL language are

- The void instruction $\gamma_1 : \bar{\epsilon}$,
- and the assignations :
  - $\gamma_2 : X := \epsilon$,
  - $\gamma_3 : X := Xa$,
  - $\gamma_4 : X := Y$,
  - $\gamma_5 : X := \rho(X)$,

  for all variables X and Y in VR and $a \in A$.

Definition 2.3 (Structures of control)
The *control structures* in the PL language are

- $\mathcal{X}_1 : P; Q$.
  Meaning: Sequential execution of instructions. Similar to common programming languages.
- $\mathcal{X}_2$: if p then goto L.
  Meaning: $\mathcal{X}_2$ represents a conditional jump where p is a (logical) condition (see Definition 2.1, item 4). L is a label (see Definition 2.4). Otherwise similar to common programming languages.
- $\mathcal{X}_3$: if p then P else Q fi.
  Meaning: IF branch. p is a (logical) condition. Instructions p and Q represent the two alternatives. Compare to common programming languages.
- $\mathcal{X}_4$: while $X = \epsilon$ do P od.
  Meaning: while loop; the instructions of the form $\omega(X) = a$, $X \in VR$, $a \in A$ are not allowed. P is an instruction. The rest is similar to common programming languages.
- $\mathcal{X}_5$: loop Xcase $a_1 \to P_1$
  $\vdots \quad \vdots$
  $a_n \to P_n$
  end
  Meaning: $\mathcal{X}_5$ represents a CASE branch with a selection of different cases. First, variable X is copied internally. Then, the value of X is read from the left to right. For every possible letter a_j (i.e. elements in A) corresponding to the value of X, the corresponding instruction P_j is run. Should, for some letter a_j, the directive $a_j \to P_j$ be missing, we proceed with $a_j \to \bar{\epsilon}$, for any $j \in [n]$.

Definition 2.4 (Labels)
Labels are elements of a fixed set $M = \{L_1, L_2, \ldots\}$. A label may reference any instruction P in the following way $L : P$.

Definition 2.5 (Instructions)
An *instruction* in PL is either a base instruction or is composed of base instructions linked by control structures $\mathcal{X}_1$ to $\mathcal{X}_5$.

Definition 2.6 (PL programs)
A PL program π has the form of

$$\begin{aligned} \pi = \ & \text{input} \quad X_1, \ldots, X_r \\ & \text{Instruction}_\pi ; \\ & \text{output} \quad Z_1, \ldots, Z_s \end{aligned}$$

where $r \geq 0$, $s \geq 0$ and where Instruction_π is an instruction. The pairwise distinct variables $X_1, \ldots, X_r$ in VR are the *input variables*. The pairwise distinct variables $Z_1, \ldots, Z_s$ in VR are the *output variables*.

Should control structure X_2 : if p then goto L occur in Instruction_π, then label L may only appear once under the form $L : P$ in Instruction_π; in other words, we proceed to label L and run instruction P again.

Definition 2.7 (Execution of a PL program)
The execution of a PL program π begins by creating and initializing the input variables $X_1, \ldots, X_r$ with their input values. Any other internal variable is set to ϵ. Then Instruction_π is executed and program execution results are put into the output variables $Z_1, \ldots, Z_s$. Should the program never stop, the result of π is undefined.

As a matter of principle, we did not so much define a specific programming language PL as much as a class of programming languages. This is because we have retained the freedom of choice with respect to the sets VR and L, as well as and especially to the alphabet A. While the elements from VR and M describe the internal states of programs only, the alphabet A represents the data set on which a PL program operates. Depending on the choice of finite alphabet A, we will denote our programming language PL(A), using our Definitions 2.1 to 2.7.

Remark Strictly speaking, we have not formally defined PL(A). Errors of interpretation are possible. If our definition had been exact, we would have had to formally detail both the syntax as well as the semantics of PL(A). The description of the semantics is particularly tedious, and moreover would have been beyond the scope of our framework. We shall define at least the syntax of PL(A) by a context-free (formal) grammar.

2.3 A context-free grammar for PL(A)

The following context-free grammar $G(A)=(V_T, V_N, S_0, P)$ generates all valid PL(A) programs for a fixed alphabet A. Unfortunately, the grammar may generate valid, as well as invalid PL programs. This follows from Definition 2.6 which lists various colloquial rules such as "*label L may only appear once under the form* $L : P$ *in* Instruction_π". Such rules cannot be conceived by context-free grammars, but prove to be useful in differentiating valid from invalid programs generated by $G(A)$. 'Real' programming languages generated from context-free grammars share this limitation, since colloquial natural language rules are bound to be formulated there, as well.

Example 2.1 In SIMULA: "*Jumps into the body of a* WHILE *loop are forbidden*" [7,17].

2.3.1 Description of the $G(A) = (V_T, V_N, S_0, P)$ grammar

The set of terminal symbols V_T is given by

$$V_T = A \cup M \cup VR \cup \{\text{input, output, if, then, goto, else, fi, while, do, od, loop, case, end, :, =, } \rightarrow, ;, ,, \sqcup, (,), \rho, \omega, \epsilon, \bar{\epsilon}\}$$

The last set in the previous union set is made of the base symbols.

The set of non-terminal symbols V_N is

$$V_N = \{< \text{program} >, < \text{statement} >, < \text{simple statement} >, < \text{identifier} >, < \text{label} >, < \text{identifier list} >, < \text{condition} >\}$$

The starting symbol S_0 is $<$ program $>$.

The set P contains the following production rules:

1. $< \text{program} > \rightarrow$ input $< \text{identifier list} >$; $< \text{statement} >$; output $< \text{identifier list} >$;
2. $< \text{identifier list} > \rightarrow < \text{identifier list} >, < \text{identifier} >$
3. $< \text{identifier list} > \rightarrow < \text{identifier} >$
4. $< \text{identifier} > \rightarrow X$ for all $X \in VR$
5. $< \text{identifier} > \rightarrow \epsilon$
6. $< \text{statement} > \rightarrow < \text{label} >; < \text{statement} >$
7. $< \text{statement} > \rightarrow < \text{statement} >; < \text{statement} >$
8. $< \text{statement} > \rightarrow$ if $< \text{condition} >$ then goto $< \text{label} >$.
9. $< \text{statement} > \rightarrow$ if $< \text{condition} >$ then $< \text{statement} >$ $< \text{statement} >$ else $< \text{statement} >$ fi
10. $< \text{statement} > \rightarrow$ while $< \text{identifier} > = \epsilon$ do $< \text{statement} >$ od
11. $< \text{statement} > \rightarrow$ loop $< \text{identifier} >$ case $a_1 \rightarrow < \text{statement} >$, $\vdots$ $a_n \rightarrow < \text{statement} >$, end
12. $< \text{statement} > \rightarrow < \text{simple statement} >$
13. $< \text{label} > \rightarrow L$, for all $L \in M$
14. $< \text{condition} > \rightarrow \omega(X) = a$ for all $a \in A, X \in VR$
15. $< \text{condition} > \rightarrow X = \epsilon$ for all $X \in VR$
16. $< \text{simple statement} > \rightarrow \bar{\epsilon}$
17. $< \text{simple statement} > \rightarrow X := \epsilon$ for all $X \in VR$
18. $< \text{simple statement} > \rightarrow X := Xa$ for all $X \in VR$, $a \in A$
19. $< \text{simple statement} > \rightarrow X := X'$ for all $X, X' \in VR$
20. $< \text{simple statement} > \rightarrow X := \rho(X)$ for all $X \in VR$

We can establish the followings correspondence between these production rules and the previous definitions:

- Productions 1–5 $\widehat{=}$ Definition 2.6.
- Production 6 $\widehat{=}$ Definition 2.4.
- Production 7–13 $\widehat{=}$ Definition 2.3 and 2.5.
- Production 14–20 $\widehat{=}$ Definition 2.1 and 2.2.

The afore-mentioned limitations of colloquial human language rules remain unaddressed by $G(A)$.

2.4 PL(A)-computable functions—Church's thesis

Let us now consider a finite alphabet A and a program $\pi \in PL(A)$. The program π contains $r \geq 0$ input variables and $s \geq 0$ output variables. During the program execution, input variables assignment yields the assignment of output variables, provided the program terminates. If the latter indeed halts—which cannot be pre-supposed—the program results represent the final output variable assignments. Should the program never halt, the results are undefined. In both cases, we do not care about any possible intermediate variable assignment during program execution. This viewpoint corresponds to Definition 2.8.

Definition 2.8 Let $\pi \in PL(A)$ be a program. The *function computed* by π is $\varphi_\pi : (A^*)^r \to (A^*)^s$, with $s \geq 0$ and $r \geq 0$. φ_π assigns to every starting initialization $(x_1, \ldots, x_r)$, $x_i \in A^*, i \in [r]$[2] a result $(z_1, \ldots, z_s) = \varphi_\pi(x_1, \ldots, x_r)$, $z_j \in A^*$, $j \in [s]$, provided the program π indeed halts. If it does not halt, $\varphi_\pi(x_1, \ldots, x_r)$ is undefined.

Remark From Definition 2.8, it follows that:

1. φ_π is generally a partial function.
2. The special cases $r = 0$ and $s = 0$ are implicitly accepted. Their meaning has however to be clarified. It describes the null t-uple ( ):

   (a) $\varphi_\pi : (A^*)^r \to (A^*)^0, r \geq 1$ assigns the null t-uple ( ) to every r-uple $(x_1, \ldots, x_r) \in (A^*)^r$, provided that the program π halts on the initial r-uple $(x_1, \ldots, x_r)$.

   $$\varphi_\pi(x_1, \ldots, x_r) = \begin{cases} ( ) & \text{if } \pi \text{ halts} \\ \text{undefined} & \text{otherwise} \end{cases}$$

   (b) $\varphi_\pi : (A^*)^0 \to (A^*)^s, s \geq 1$ associates the null t-uple ( ) to a s-uple $(z_1, \ldots, z_s) \in (A^*)^s$ provided that the program π halts.

   $$\varphi_\pi( ) = \begin{cases} (z_1, \ldots, z_s) & \text{if } \pi \text{ halts} \\ \text{undefined} & \text{otherwise} \end{cases}$$

   (c) $\varphi_\pi : (A^*)^0 \to (A^*)^0$ associates the null t-uple ( ) to the the null t-uple ( ), provided that the program π halts.

   $$\varphi_\pi( ) = \begin{cases} ( ) & \text{if } \pi \text{ halts} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 2.9 Let be A a fixed alphabet.

1. A word function $f : (A^*)^r \to (A^*)^s, r, s \geq 0$ is said $PL(A)$-*computable* (or *computable* for short) if there exists a program $\pi_f \in PL(A)$ with $\varphi_{\pi_f} = f$.
2. The set $\mathcal{P}(A) = \{\varphi_\pi | \pi \in PL(A)\}$ is called the *set of* $PL(A)$-*computable functions*.

Again and again, attempts have been made to define classes of computable functions to refine the intuitive notion of computability. In the end, all these attempts produced the same sets of "computable functions". For example, the set of "Turing machines computable" functions has been proven identical to the set of "partial recursive" functions. In our case, for a fixed alphabet A, the set $\mathcal{P}(A)$ is also identical to both previous sets. These set similarities gave birth to Church's thesis.

Definition 2.10 (Church's thesis)
Every intuitively computable function is $PL(A)$-computable and conversely.

From Definition 2.10 it follows that if A_1 and A_2 are two pairwise distinct finite alphabets, we can obviously differentiate the sets $\mathcal{P}(A_1)$ and $\mathcal{P}(A_2)$ from one another. Consequently we will abandon the *differentiation based on the alphabet* and will simply use $\mathcal{P}$ to describe the set of computable or partial recursive functions (see also Definition 2.12). The following extension to the Church's thesis is essential, as well.

Definition 2.11 (Extension to the Church's thesis)
To every computable function f and for any finite alphabet A, we can effectively associate a program $\pi \in PL(A)$ with $f = \varphi_\pi$.

Definition 2.12 1. $\mathcal{P}_s^r = \{f \in \mathcal{P} | f : (A^*)^r \to (A^*)^s, \ r, s \geq 0\}$.
2. $\mathcal{R} = \{f \in \mathcal{P} | f \text{ is total}\}$ is the set of the total recursive functions.
3. $\mathcal{R}_s^r = \mathcal{R} \cap \mathcal{P}_s^r, \ r, s \geq 0$.

2.5 Coding and "Gödel Numbering" of $\mathcal{P}$

Definition 2.13 Let A be a finite alphabet. A set $B \subseteq (A^*)^r$, $r \geq 0$ is said to be *decidable* or equivalently to be *recursive* if there exists a function $\chi_B : (A^*)^r \to A^*$ such that

$$\chi_B(x_1, \ldots, x_r) = \epsilon \Leftrightarrow (x_1, \ldots, x_r) \in B.$$

Definition 2.14 Let A_1 and A_2 be two finite alphabet. A function $\xi : A_1^* \to A_2^*$ is called a *coding* of A_1^* with respect to A_2^* if

1. $\xi \in \mathcal{R}$,
2. ξ is an injective function,
3. $\xi(A_1^*)$ is decidable,
4. $\xi^{-1} \in \mathcal{P}$.

[2] Notation: $[r] = \{1, \ldots, r\}$ for any $n \in \mathbb{N}$. This notation is not to be mistaken with a bibliographical reference.

Let $A_0 = \{1\}$. Then we can identify A_0^* with the natural numbers including the null element – this set being denoted $\mathbb{N}_0$ – in the following way:

$$\epsilon \mathrel{\widehat{=}} 0$$
$$\underbrace{1111\ldots111}_{n \text{ ones}} \mathrel{\widehat{=}} n \in \mathbb{N}$$

Definition 2.15 A coding $\xi : A^* \to A_0^* = \{1\}^* \mathrel{\widehat{=}} \mathbb{N}_0$ is called a *Gödel numbering*. $\xi(\omega)$ is called the Gödel number of ω for all $\omega \in A_0^*$.

In the rest of this work, we will specify the associate alphabet to any Gödel numbering of a $PL(A)$ program. We will at the same time give a Gödel numbering of $\mathcal{P}$. Let $A = \{a_1, \ldots, a_n\}$ be a fixed alphabet. Let set B list all special symbols as well as constitutive letters of the word symbols "input, if, fi..." of $PL(A)$.

$$B = \{:, =, \epsilon, \bar{\epsilon}, \to, ;, ,, \sqcup, \omega, \rho, (,),$$
$$a, c, d, e, f, g, h, i, l, n, o, p, s, t, u, w\}.$$

The set of labels and variables in $PL(A)$ programs are $M = \{L_1, L_2, \ldots\}$ and $VR = \{V_1, V_2, \ldots\}$, respectively. Let $L_i, i \geq 1$ ($V_j, j \geq 1$, respectively) be arbitrary names for labels or variables, respectively. It is worth mentioning the fact we hitherto did not need to restrict the choice of these names. However, we have to make sure in our forthcoming consideration that we work on a finite alphabet. Thus, we are going to standardize the naming as follows:

- The name L_1 is textually the same as "L_1".
- The name L_2 is textually the same as "L_2" and so on...
- And in a similar way, the name V_1 is textually the same as "V_1" and so on...

Hence we have

$$M \subset \{L, 0, 1, \ldots, 9\}^*, \qquad VR \subset \{V, 0, 1, \ldots, 9\}^*.$$

Let us now consider the set $C = A \cup B \cup \{V, 0, 1, \ldots, 9\}$. Every program $\pi \in PL(A)$ can be interpreted as a word from the set C^* and $PL(A)$ itself, as well as a subset of C^*. We then extensively define an injective mapping $H : C \to \{1\}^* \mathrel{\widehat{=}} \mathbb{N}_0$ as follows:

$$\begin{array}{llll}
: \longmapsto 0 & c \longmapsto 13 & u \longmapsto 26 & \\
= \longmapsto 1 & d \longmapsto 14 & w \longmapsto 27 & \\
; \longmapsto 2 & e \longmapsto 15 & L \longmapsto 28 & \\
, \longmapsto 3 & f \longmapsto 16 & V \longmapsto 29 & \\
(\longmapsto 4 & g \longmapsto 17 & 0 \longmapsto 30 & \\
) \longmapsto 5 & h \longmapsto 18 & \ldots &
\end{array}$$

$$\begin{array}{llll}
\sqcup \longmapsto 6 & i \longmapsto 19 & \ldots & \\
\to \longmapsto 7 & l \longmapsto 20 & 0 \longmapsto 39 & \\
\epsilon \longmapsto 8 & n \longmapsto 21 & a_1 \longmapsto 40 & \\
\bar{\epsilon} \longmapsto 9 & o \longmapsto 22 & \ldots & \\
\rho \longmapsto 10 & p \longmapsto 23 & \ldots & \\
\omega \longmapsto 11 & s \longmapsto 24 & a_n \longmapsto 39 + n & \\
a \longmapsto 12 & t \longmapsto 25 &
\end{array}$$

The injective mapping H can also be extended to the injective mapping $H^* : C^* \to \mathbb{N}_0^*$:

$$H^*(\epsilon) \longmapsto \epsilon$$
$$H^*(\bar{x}y) \longmapsto H^*(\bar{x})H(y), \forall y \in C, \forall \bar{x} \in C^*$$

Lemma 2.1 *H^* is a coding of C^* with respect to $\mathbb{N}_0^*$.*

Proof 1. From the definition of H and H^*, H^* is intuitively computable, and according to Church's thesis, it is decidable, as well. H^* is defined for every element from C^*. H^* is also total and totally from $\mathcal{R}$.

2. H^* is trivially injective.

3. Let us consider $D : H^*(C^*)$. D is a subset of $\mathbb{N}_0^*$. Let $\bar{i} \in \mathbb{N}_0^*$. $\bar{i}$ has a finite size[3] $l(\bar{i})$, $\bar{i} = m_1 m_2 \ldots m_{l(\bar{i})}$ with $m_j \in \mathbb{N}_0$ for $j \in [l(\bar{i})]$. $\bar{i}$ is precisely from set D when every m_j has an antecedent in C, with respect to H. In order to determine whether $\bar{i} \in D$ or not, we need to perform $l(\bar{i})^*|C|$ tests at most. There also exists a total recursive function

$$\chi_D : \mathbb{N}_0^* \to \mathbb{N}_0^*,$$

with

$$\begin{array}{rl}
& \chi_D(\bar{i}) = \epsilon \\
\Longleftrightarrow & \text{(Every element of } \bar{i} \text{ has an antecedent in } C \\
& \text{with respect to } H) \\
\Longleftrightarrow & \bar{i} \in D.
\end{array}$$

Then $D = H^*(C^*)$ is decidable.

4. In item 3, it has been shown that for every $\bar{i} \in H^*(C^*)$ we can effectively identify an antecedent in C^*. Hence $(H^*)^{-1}$ is computable for every element in which it is defined and thus $(H^*)^{-1} \in \mathcal{P}$.

From all of this, it follows that H^* is a coding. □

We now consider the total recursive function $f : \mathbb{N}_0^* \to \mathbb{N}_0$ with

$$\bar{i} \longmapsto \begin{cases} 0 & \text{if } \bar{i} = \epsilon \\ P_1^{m_1} \ldots P_{l(\bar{i})}^{m_{l(\bar{i})}+1} - 1 & \text{if } \bar{i} = m_1 \ldots m_{l(\bar{i})} \end{cases}$$

where P_j is the j-th prime integer.

[3] The notation l describes with respect to a base alphabet B the length function of elements from B^*.

Proposition 2.1 *f is a bijective function.*

Proof 1. f is injective due to the unicity of factorization into prime numbers.
2. f is surjective since every number $m \in \mathbb{N}$ such that $m > 1$ can be factored into prime numbers. □

When coding C^* with H^* with respect to $\mathbb{N}_0^*$, and when considering $f \in \mathcal{R}$ as a bijective function from $\mathbb{N}_0^*$ to $\mathbb{N}_0$, we can establish Lemma 2.2.

Lemma 2.2 *The function $g := f \circ H^* : C^* \to \mathbb{N}_0$ is a Gödel numbering of C^* with respect to $\mathbb{N}_0$.*

Proof With every word w from C^* and thus every program from $PL(A)$ an integer $f \circ H^*(w)$ from $\mathbb{N}_0$ is is unequivocally associated. Since H^* is injective and f is bijective, g is a Gödel numbering of C^*. □

Lemma 2.3 *The set $T = \{g(x)|x \in PL(A)\} \subset \mathbb{N}_0$ is decidable.*

Proof 1. $i = 0$ is not in T since the null (void) word is not a program.
2. Let be $i \in \mathbb{N}$. We thus can represent it as follows:

$$i = P_1^{m_1} \dots P_k^{m_k+1} - 1, k \geq 1.$$

There is at least one m_j, $j \in [k]$ which does not have an antecedent with respect to H. This i is not in the image set of g and thus is not in T. Let us now consider every m_j from the image space of H. Then, there exists a word $w \in C^*$ with $g(w) = i$. By considering the grammar defined in Sect. 2.3 and colloquial human language rules such as "The input variables are pairwise distinct", we can design a program (compare "formal languages" with "compiler design"), which halts on every input $w \in C^*$ and precisely outputs ϵ whenever $w \in PL(A)$. There also exists a total recursive function whose result is also ϵ whenever the antecedent of an element in $\mathbb{N}_0$, if any, is a valid $PL(A)$ program. Then T is decidable. □

Definition 2.16 If the set B is the domain of a partial recursive function, then $B \subset (A^*)^q, q \geq 0$ is said to be *enumerable*.

Lemma 2.4 *$B \subset (A^*)^q, q \geq 0$ decidable $\Rightarrow$ B is enumerable.*

Proof Let us suppose that $B \subset (A^*)^q, q \geq 0$ is decidable. From Definition 2.13, there exists a total recursive function

$$\chi_B : (A^*)^q \to A^* \text{ such that } \chi_B(x_1, \dots, x_q) = \epsilon \Leftrightarrow (x_1, \dots, x_q) \in B.$$

From χ_B we can draw a partial recursive application

$$f_B : (A^*)^q \to A^*,$$

such that

$$f_B(x_1, \dots, x_q) = \begin{cases} \epsilon & \text{if } \chi_B(x_1, \dots, x_q) = \epsilon \\ \text{is undefined} & \text{otherwise} \end{cases}.$$

Then, the application $g_B : (A^*)^q \to (A^*)^q$ with[4] $g_B(\bar{x}) = \prod_1^2(\bar{x}, f_B(\bar{x}))$, $\bar{x} \in (A^*)^q$, is thus a partial recursive function and its image set is precisely the set B. □

Corollary 2.1 *The set T is enumerable.*

Proof Consequence from Lemma 2.4 since T is decidable. □

Similar to the proof of Lemma 2.3, we can prove that for every $m, k \geq 0$, the set

$$T_{m,k} = \{g(\pi)|\pi \in PL(A), \pi \text{ has exactly } m \text{ input variables and } k \text{ output variables}\} \subset \mathbb{N}_0$$

is decidable. We need to consider the decision algorithm used in the proof of Lemma 2.3 to determine whether a program $v \in PL(A)$ has exactly m input variables and k output variables. From the decidability of the set $T_{m,k}$, it follows that $T_{m,k}$ is enumerable as well, and thus that there exists a partial recursive function from $\mathbb{N}_0$ to $\mathbb{N}_0$, whose image space is the set $T_{m,k}$. Since $T_{m,k} \neq \emptyset$, it follows that there even exists a total recursive function (see [5, p. 82]) given by

$$t_{m,k} : \mathbb{N}_0 \to \mathbb{N}_0, \text{ with } t_{m,k}(\mathbb{N}_0) = T_{m,k}.$$

It makes also sense to speak of the i-th element of $T_{m,k}$. For every $m, k \geq 0$, we can generate the set $\{\pi_0, \pi_1, \pi_2, \dots\}$ of all programs having m input variables and k output variables. We have with π_j a $PL(A)$ program such that $t_{m,k}(j) = g(\pi_j)$.

In general, there also exists for every $m, k \geq 0$ a total recursive function

$$\gamma_{m,k} : \mathbb{N}_0 \to \{\pi|\pi \in PL(A), \pi \text{ has exactly } m \text{ input variables and } k \text{ output variables}\} =: W,$$

with the inverse function $\bar{\gamma}_{m,k} : W \to \mathbb{N}_0$ such that

$$\bar{\gamma}_{m,k}(\pi) = \min\{j|\gamma_{m,k}(j) = \pi\}.$$

Along with the set $T_{m,k}$, of course the set $\mathcal{P}_k^m$ for every $m, k \geq 0$ is decidable and thus enumerable. We also can write the set $\mathcal{P}_k^m$ under the form $\{f_0, f_1, f_2, \dots\}$, where for every f_j we have $f_j = \varphi_{\pi_j}$. The Gödel number of π_j also relates to f_j. In the above-mentioned enumeration of $\mathcal{P}_k^m$, every function of $\mathcal{P}_k^m$ is counted several times. This means that more than one Gödel number correspond to these functions. Thus we have the following Lemma.

[4] The notation $\prod_i^n$ generally describes the projection onto the ith component when considering a n-tuple.

Lemma 2.5 *For every $f \in \mathcal{P}_k^m, m, k \geq 0$, f has more than one Gödel number, with respect to the Gödel numbering g.*

Proof Let be $m, k \geq 0$, $f \in \mathcal{P}_k^m$. f is computed through the following program:

$$\begin{aligned} \pi_0 = \ & \text{input} \quad X_1, \quad \ldots, X_m; \\ & \quad\ \text{AW}_{\pi_0}; \\ & \text{output} \quad Z_1, \quad \ldots, \ Z_k \end{aligned}$$

We have $\varphi_{\pi_0} = f$. But f can also be computed with the programs $\pi_1, \pi_2, \pi_3, \ldots$ defined by

$$\begin{aligned} \pi_i = \ & \text{input} \quad X_1, \ldots, \ X_m; \\ & \quad\ \text{AW}_{\pi_0}; \\ & \quad\ \underbrace{\bar{\epsilon}; \ldots; \bar{\epsilon};}_{i \text{ times}} \\ & \text{output} \ Z_1, \ldots, \quad Z_k \end{aligned}$$

We thus have $f = \varphi_{\pi_1} = \varphi_{\pi_2} = \varphi_{\pi_3} = \ldots$

Since $g(\pi_1) \neq g(\pi_2) \neq g(\pi_3) \neq \ldots$, hence f has an infinite number of Gödel numbers. □

Definition 2.17 Let $\mathcal{F}$ be a set of word functions and let be

$$\mathcal{F}_s^r = \{f : (A^*)^r \to (A^*)^s | f \in \mathcal{F}\}, r, s \geq 0.$$

A function $\psi \in \mathcal{F}_s^{r+1}$ is called *universal* for $\mathcal{F}_s^r$ when[5]

$$\mathcal{F}_s^r = \{\lambda \bar{y}[\psi(x, \bar{y})] | x \in A^*, \bar{y} \in (A^*)^r\}.$$

Proposition 2.2 *For every $m, k \geq 0$, there exists an universal function $\psi_{m,k} \in \mathcal{P}_k^{m+1}$ for $\mathcal{P}_k^m$.*

Proof (By using Church's thesis)
The set of all program having m input variables and k output variables exists under an enumerated form, say by using $\gamma_{m,k}$:

$\pi_0, \pi_1, \pi_2, \ldots$

Then the function

$$\psi_{m,k} = \lambda x, \bar{y}[\varphi_{\pi_{\bar{\gamma}_{m,k}(x)}}(\bar{y})], \bar{y} \in (A^*)^m$$

is universal for $\mathcal{P}_k^m$, and $\psi_{m,k}$ is intuitively computable. From the Church's thesis, there exists $\pi_u^{m,k} \in PL(A)$ with $m+1$ input variables and k output variables such that $\psi_{m,k} = \varphi_{\pi_u^{m,k}}$. □

Remark We can also prove Proposition 2.2 by constructing $\pi_u^{m,k}$ directly. The proof is, however, more complex.

2.6 Lexicographic order of A^*

At this point, we want to introduce a broader Gödel numbering of A^*. to this end, we will first explain what we mean by lexicographic order of A^*.

[5] For the Lambda notation, refer to [20, p. 13].

Definition 2.18 Let be $A = \{a_1, \ldots, a_n\}$. The successor function $\nu : A^* \to A^*$ is defined as follows:

- $\nu(\epsilon) = a_1$,
- $\nu(xa_i) = xa_{i+1}$,
- $\nu(xa_n) = \nu(x)a_1$, for $i \in [n-1], x \in A^*, a_i \in A$.

Lemma 2.6 *The successor function $\nu : A^* \to A^*$ is bijective.*

Proof On the one hand, ν is injective since two elements of A^* can have the same successor only if they are themselves equal. On the other hand, ν is surjective since by successively applying the function ν, every word from A^* can be reduced to the null (void) word. Thus, $\{\nu^i(\epsilon) | \in \mathbb{N}\} = A^*$. □

Since $w_i = \nu^i(\epsilon)$ for every element w_i from A^*, we obtain an order on A^* [5].

Definition 2.19 Let $w_i = \nu^i(\epsilon)$ and $w_j = \nu^j(\epsilon)$ two elements from A^*. Then $w_i \leq w_j$ whenever $i \leq j$. This order is called the *lexicographic order* on A^*.

We can also list all the words from A^* in a unique lexicographic sequence:

$\epsilon = w_0 < w_1 < w_2 \ldots$

Proposition 2.3 *Let be $A = \{a_1, \ldots, a_n\}$ and $A_0 = \{1\}$. Let be $C_p : A^* \to \mathbb{N}_0 \widehat{=} \{1\}^*$ the application which lists, for every $x \in A^*$, the number of x in a sequence of lexicographical ordering:*

$x \longmapsto i$ *with* $\nu^i(\epsilon) = x$.

Then C_p is a bijective Gödel numbering.

Proof 1. C_p is total since C_p is defined on the whole set A^*. For every x, we can effectively obtain $C_p(x)$ by applying the definition of ν a finite number of times. So $C_o \in \mathcal{R}$.
2. C_p is injective since no two elements of A^* have the same location (index) in the sequence in lexicographic order.
3. Since $C_p(A^*) = \mathbb{N}_0$, then $C_p(A^*)$ is naturally decidable.
4. Starting from the void word, we can for every $i \in \mathbb{N}_0$ effectively generate by means of ν all the words from A^* up to the i-th word in the lexicographic order sequence. This i-th word is the antecedent of i. So C_p^{-1} is computable and since $C_p(A^*) = \mathbb{N}_0$, it is also total. Hence $C_p^{-1} \in \mathcal{R}$.

It follows from these four points that C_p is a Gödel numbering. From the second point and from the fact that $C_p(A^*) = \mathbb{N}_0$, C_p is bijective. □

Lemma 2.7 *let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$ be two alphabets. The function*

$\xi_{m,n} = C_m^{-1} \circ C_n : A^* \to B^*$

is a coding of A^* *with respect to* B^*.

Proof Obvious. □

2.7 Reduction with respect to input and output variables

We would like to show that it is possible to transform every program $\pi \in PL(A)$ with r input variables and s output variables into an equivalent program π' having only one input variable and only one output variable. Due to the equivalence between $\mathcal{P}$ and $PL(A)$, it actually suffices to just consider the functions from $\mathcal{P}_1^1$ to treat the whole set $\mathcal{P}$. The universal function with respect to $\mathcal{P}_1^1$ is then also universal with respect the whole set $\mathcal{P}$.

Let $\pi \in PL(A)$ be a program with r input variables and s output variables, $r, s \geq 1$. Then π has the following structure:

$$\begin{aligned} \pi = \ & \text{input} \quad X_1, \ldots, X_r; \\ & \quad\ \ \mathrm{AW}_\pi; \\ & \text{output} \quad Z_1, \ldots, Z_s \end{aligned}$$

Every element $\bar{x} = (x_1, \ldots, x_r)$ from $(A^*)^r$ can be an instance of π. In other words, $\bar{x}$ can also be represented by

$$\bar{x} = x_1|x_2|\ldots|x_r \in (A \cup \{|\})^*, \quad \text{where } | \notin A.$$

Under this form, $\bar{x}$ is treated as an input to a program π' (equivalent to π) having only one input and only one output variable.

$$\begin{aligned} \pi' = \ & \text{input } X; \\ & [X_1 := x_1; \ldots; X_r := x_r]; \\ & \mathrm{AW}_\pi; \\ & [Z_1 := z_1|\ldots|z_s]; \\ & \text{output } Z \end{aligned}$$

The part of the program between the square brackets can be made explicit in the following way. The first part $[X_1 := x_1; \ldots; X_r := x_r]$; is realized by:

```
X_1 := ϵ; ...; X_r := ϵ
loop X case a_1 → X_r := X_r a_1,
            ⋮       ⋮
            a_n → X_r := X_r a_n,
            |   → X_1 := X_2;
                  X_2 := X_3;
                  ⋮
                  X_{r-1} := X_r;
                  X_r := ϵ;
end
```

and the second part $[Z_1 := z_1|\ldots|z_s]$; is realized by:

```
Z := ϵ;
loop Z_1  case a_1 → Z := Z a_1,
               ⋮       ⋮
               a_n → Z := Z a_n,
          end;
Z := Z|;
loop Z_2  case a_1 → Z := Z a_1,
               ⋮       ⋮
               a_n → Z := Z a_n,
          end;
Z := Z|;
⋮
Z := Z|;
loop Z_s  case a_1 → Z := Z a_1,
               ⋮       ⋮
               a_n → Z := Z a_n,
          end;
```

π' is not an element from $PL(A)$, since the working alphabet is not A but $A \cup \{|\} \neq A$. But one can obtain from π' a program $\pi'' \in PL(A)$ (equivalent to π and containing only one input and only one output variable) by coding the set $(A \cup \{|\})^*$ with respect to A^*. For coding, we can consider the function

$$\xi_{n+1,n} = C_n^{-1} \circ C_{n+1}$$

where the function C_n^{-1} and C_{n+1} have been defined in Sect. 2.6.

Essentially, all programs from $PL(A)$ can be reduced to only one input and only one output variable (should such variables exists). It follows from Church's thesis that it suffices to consider the set $\mathcal{P}_1^1$ instead of the set $\mathcal{P}$. The universal function $\psi_{1,1}$ with respect to $\mathcal{P}_1^1$ is in this context universal for the whole set $\mathcal{P}$, as well. In order to illustrate this point more clearly, we are going to introduce the following notation.

Definition 2.20 Let $\psi_{1,1}$ be a universal function with respect to $\mathcal{P}_1^1$.

$\varphi_x^k = \lambda y_1, \ldots, y_k[\psi_{1,1}(x), \xi_{n+1,n}(y_1|\ldots|y_k)]$,

with $x, y_1, \ldots, y_k \in A^*$. When then have

$\varphi_x^k : (A^*)^k \to A^*$.

2.8 Recursion Theorem - s-m-n-Theorem

In this section, we are going to prove the recursion theorem which will enable us to demonstrate the existence of self-reproducing $PL(A)$ programs. Let us fix an alphabet A such that every $PL(A)$ program lies in A^*.

Proposition 2.4 *(s-m-n-Theorem)*
For every $m, n \in \mathbb{N}$ there exists a function $s_n^m \in \mathcal{R}_1^{m+1}$, which for every $x \in A^*$, $\bar{y} \in (A^*)^m$, $\bar{z} \in (A^*)^n$ satisfies the following equation:

$$\varphi_x^{m+n}(\bar{x}, \bar{y}) = \varphi_{s_n^m(x,\bar{y})}^n(\bar{z}).$$

Proof Let us fix $m, n \in \mathbb{N}$.

Case 1. x is not a valid program text. Then φ_x^{m+n} is undefined. In this case, $\varphi_{s_n^m(x,\bar{y})}^n$ must be also undefined. Thus, we set $s_n^m(x, \bar{y}) = \epsilon$. Now $s_n^m(x, \bar{y})$ is not a valid program text as well, and so

$$\varphi_x^{m+n}(\bar{x}, \bar{y}) = \varphi_{s_n^m(x,\bar{y})}^n(\bar{z}) = \text{undefined}.$$

Case 2. x is a valid program text, so $x \in PL(A)$. Then x has the following structure:

$$\begin{aligned} x = \text{input}\ & Y_1, \ldots, Y_m, Z_1, \ldots, Z_n; \\ & \text{AW}_x; \\ \text{output}\ & W \end{aligned}$$

When setting $\bar{y} = (y_1, y_2, \ldots, y_m)$, we have

$$\begin{aligned} s_n^m(x, \bar{y}) = \text{input}\ & Z_1, \ldots, Z_n; \\ & [Y_1 := y_1]; \ldots; [Y_m := y_m]; \\ & \text{AW}_x; \\ & \text{output}\, W \end{aligned}$$

The parts of the program between the square brackets can be easily realized. This can be shown, for instance, with $[Y_1 := y_1]$:
y_1 is an element from A^* and thus has a finite size $l(y_1) \in \mathbb{N}_0$

$$y_1 = a_{1_1} \ldots a_{1_{l(y_1)}}, \text{ with } a_{1_j} \in A^*.$$

Therefore $[Y_1 := y_1]$ will be realized as follows:

$$\begin{aligned} & Y_1 := \epsilon; \\ & Y_1 := Y_1 a_{1_1}; \\ & \quad \vdots \\ & Y_1 := Y_1 a_{1_{l(y_1)}}; \end{aligned}$$

From Church's thesis, we have $s_n^m \in \mathcal{R}_1^{m+1}$. □

Corollary 2.2 *There exists a $h \in \mathcal{R}_1^m$ such that for every $f \in \mathcal{P}_1^{m+n}$ we have:*

$$f(\bar{x}, \bar{y}) = \varphi_{h(\bar{y})}^n(\bar{x}), \quad \forall \bar{x} \in (A^*)^m, \quad \forall \bar{y} \in (A^*)^n.$$

Proof Since f is a computable function, there exists a program $\pi_0 \in A^*$, with $f = \varphi_{\pi_0}^{m+n}$. From Proposition 2.4, hence we have

$$f(\bar{x}, \bar{y}) = \varphi_{\pi_0}^{m+n}(\bar{x}, \bar{y}) = \varphi_{s_n^m(\pi_0,\bar{y})}^n(\bar{x}).$$

Hence the corollary follows by setting $h = \lambda \bar{y}[s_n^m(\pi_0, \bar{y})]$. □

Remark Let be $g \in \mathcal{P}_1^{k+1}$. Then there exists a program $x \in A^*$ with

$$g = \varphi_x^{k+1}.$$

Let us introduce the following notation:

$$g_x = \varphi_{s_k^1(x,y)}^k.$$

With this notation, we have:

$$g_x(\bar{y}) = g(x, \bar{y}), \forall \bar{y} \in (A^*)^k.$$

Let be $h \in \mathcal{P}_1^r$ and suppose that $h(\bar{x})$ in undefined for a $\bar{x} \in (A^*)^r$. According to our notation, the function $g_{h(\bar{x})}$ is undefined everywhere.

We can now prove the following formulation of Kleene's recursion theorem:

Proposition 2.5 (*Recursion theorem as formulated in* [5])
For every function $f \in \mathcal{P}_1^1$ there exists a program text $x \in A^*$ which satisfies

$$\varphi_x = \varphi_{f(x)}.$$

Proof The function $g = \lambda y, x[\varphi_{\varphi_y(y)}(x)]$ lies in $\mathcal{P}_1^2$ with $x, y \in A^*$. We have proved in Corollary 2.2 that there exists a $h \in \mathcal{R}_1^1$ such that

$$\varphi_{h(y)} = g_y = \varphi_{\varphi_y(y)} \forall y \in A^*. \tag{1}$$

Let us now consider $f \in \mathcal{P}_1^1$. Since $h \in \mathcal{R}_1^1$ we can apply one after the another the function f and h: $f \circ h$ lies in $\mathcal{P}_1^1$ as well. From Church's thesis we can effectively associate a program $\pi \in PL(A)$ to $f \circ h$ with

$$\varphi_\pi = f \circ h. \tag{2}$$

From Eqs. (1) and (2), hence we have:

$$\varphi_{h(\pi)} \overset{(1)}{=} \varphi_{\varphi_\pi(\pi)} \overset{(2)}{=} \varphi_{f(h(\pi))}.$$

We have the value of x we were looking for:

$$x = h(\pi).$$

□

Definition 2.21 Let be $f \in \mathcal{P}_1^1$. An element $x \in A^*$ is called a *fixed point value* if we have

$$\varphi_x = \varphi_{f(x)}.$$

Corollary 2.3 *For every function $g \in \mathcal{P}_1^2$, there exists a program $x_0 \in A^*$ such that*

$$\varphi_{x_0} = g_{x_0}.$$

Proof From Corollary 2.2, there exists a function $h \in \mathcal{R}_1^1$ with $\varphi_{h(y)} = g_y$, for every $y \in A^*$. The function h has a fixed point value x_0 from the recursion theorem, with

$$\varphi_{x_0} = \varphi_{h(x_0)} = g_{x_0}.$$

Hence the result. □

Proposition 2.6 *There exists in $\mathcal{P}_1^1$ a function with a program source $x_0 \in A^*$, which produces on every input $y \in A^*$ its own source code x_0.*

Proof The function $g \in \prod_1^2 : (A^*)^2 \to A^*$ with $g(x, y) = x$,
for every $x, y \in A^*$ is trivially in $\mathcal{P}_1^2$.

From Corollary 2.3, it follows that the equation $\varphi_x = g_x$ has a solution x_0. Hence, there exists a $x_0 \in A^*$ such that

$$\varphi_{x_0} = g_{x_0} = \lambda y[x_0] \Rightarrow \varphi_{x_0}(y) = x_0, \quad \forall y \in A^*.$$

φ_{x_0} is thus a function which produces its own source code x_0 for every input $y \in A^*$. By considering that $\varphi_{x_0} \in \mathcal{P}_1^1$ (obvious), we conclude the proof of the proposition. □

The next proposition is a generalization of Proposition 2.6.

Proposition 2.7 *Let $f : A^* \to A^*$ from $\mathcal{P}_1^1$. Then there exists a function in $\mathcal{P}_1^1$, whose program source $x_0 \in A^*$ produces the value $f(x_0)$ on every input.*

Proof Let be $f \in \mathcal{P}_1^1$. The function $g : (A^*)^2 \to A^*$ such that

$$g(x, y) = \prod_1^2 (f(x), y) = f(x), \quad x, y \in A^*,$$

lies in $\mathcal{P}_2^1$. This follows from the completeness of $\mathcal{P}$ with respect to the functional substitution and composition (see [5]). From Corollary 2.3 it follows that the equation $\varphi_x = g_x$ has a solution x_0. Thus, there exists a $x_0 \in A^*$ such that

$$\varphi_{x_0} = g_{x_0} = \lambda y[f(x_0)] \Rightarrow \varphi_{x_0}(y) = f(x_0), \quad \forall y \in A^*.$$

Since φ_{x_0} is a constant function, it is obvious trivial that φ_{x_0} lies in $\mathcal{P}_1^1$. Hence, φ_{x_0} is the solution we were looking for. □

From Proposition 2.7, it follows that there exist $PL(A)$ programs which do not just reproduce their own source code once, but many times.

Corollary 2.4 *For every $i \in \mathbb{N}$ there exists a function with a program source $x_{i_0} \in A^*$, which on every input $y \in A^*$ produces it own source code x_{i_0} successively i times.*

Proof Let be $i \in \mathbb{N}$. Then the proof follows from the proof of Proposition 2.7 by considering that $f = f_i : A^* \to A^*$ and

$$f_i(x) := x^i \quad (= \underbrace{x \ldots x}_{i \text{ times}}).$$

□

Remark Proposition 2.6 is established as a special case of Proposition 2.7 by considering that $f = \mathrm{id}$.

With the previous propositions, we have proved the theoretical existence of self-reproducing $PL(A)$ programs. Though we offered constructive proofs, effective construction of such programs is not straightforward. As such, these proofs cannot be used directly to generate actual self-reproducing $PL(A)$ programs. In Sect. 2.5, we have seen that the set of $PL(A)$ programs is enumerable. When enumerating this set lexicographically, Proposition 2.6 guarantees the existence of a number $i_0 \in \mathbb{N}_0$ such that π_{i_0} is a self-reproducing program. However, the magnitude of number i_0 all but precludes using exhaustive enumeration as a way to find self-reproducing $PL(A)$ programs.

We reiterate the purpose of Chap. 2: The quest for self-reproducing programs in high level programming languages is not futile. Such programs do indeed exist, and can be constructed.

Remark Section 4.3 deals with cyclically self-reproducing programs (see Definition 4.4). The existence of cyclically self-reproducing programs can likely be deduced from Kleene's recursion theorem, as well.

3 Self-reproducing program examples in high-level and assembly languages

3.1 Introduction

Chapter 2 offered concrete self-reproducing program examples in the fictitious PL(A) language. Since concrete programming languages match PL(A)'s expressive power, such self-reproducing examples must exist in those languages, as well. We shall offer some practical parsimonious specimens in SIMULA and PASCAL. Some such self-reproducing programs can be implemented easily on concrete computer systems; others, despite their syntactic correctness, cannot for a variety of reasons. Whenever possible, we shall distill implementable examples from the later set. Section 3.4 offers some machine-dependent examples in the SIEMENS assembly language.

3.2 Self-reproducing programs in SIMULA [19]

This chapter develops self-reproducing programs in the SIMULA language. We chose SIMULA as an example of a block-oriented language; in Sect. 3.3 we contrast examples

Table 2 Naive π_0

```
π0 =   begin OUTTEXT("......................") end;
                          ⇑
       π0's program text has to appear here, e.g.:
       begin OUTTEXT("..............") end;
                          ⇑
                       see above
                          ⋮
```

Table 3 Recursive expansion of π_0

```
π0 =   begin   OUTTEXT("BEGIN OUTTEXT("BEGIN")
               OUTTEXT("BEGIN OUTTEXT(".......
               ..............................
               ..............END") END") END")
       end
```

in PASCAL, which is not block-oriented. For our purposes, however, the availability of text variables proves to be the more important differentiating characteristic: while PASCAL just recognizes simple text constants, SIMULA in contrast allows for real text variables. Hence, in conjunction with SIMULA's class concept, we fruitfully leverage this difference in Sect. 3.2.5 by processing variables of type `text`.

3.2.1 Naive approach

We motivate the difficulties writing a self-reproducing SIMULA program π through the following naive approach:

```
x = begin OUTTEXT("x") end                    (3)
```

In essence, π contains an output instruction which constitutes π's complete program text. We express this approach in π_0 (Table 2). We can expand the recursion into an equivalent program (see Table 3). Of course, since π_0 isn't finite, it is no longer a program. The insolvability of Eq. 3 also demonstrates the "impossibility" of π_0: Since text `x` and the two constants `begin OUTTEXT(" ` and `)"end` have unequal lengths, they cannot be equal!

3.2.2 Text decomposition algorithm

It follows from Sect. 3.2.1 that a self-reproducing program π written in SIMULA cannot simply print out its own text en bloc with a simple output instruction. Hence, π has to somehow build its text step-wise from partial strings. We have to

(i) decompose π

Since we are unaware of a specific procedure, we shall at first decompose π completely into constitutive single characters. The result is shown in Table 4. Since π_1 represents an infinite text, it too cannot constitute a program: If a text consisting of 13 characters is required to output one character,

Table 4 π_1: Decomposition of π in single characters

```
π1 =   begin   OUTTEXT("BEGIN OUTTEXT("B")
               OUTTEXT("BEGIN OUTTEXT("E")
               OUTTEXT("BEGIN OUTTEXT("G")
               OUTTEXT("BEGIN OUTTEXT("I")
               OUTTEXT("BEGIN OUTTEXT("N")
               OUTTEXT("BEGIN OUTTEXT(" ")
               OUTTEXT("BEGIN OUTTEXT("O")
                                  ⋮
```

Table 5 π_2: Char array `C`

```
character array    C[0 : maxchar]
          ⋮
C[0] := "A";
C[1] := "B";                }          Letters
          ⋮
C[25] := "Z";
          ⋮
C[26] := "0";
          ⋮                 }          Digits
C[35] := "9";
          ⋮
C[36] := ";";
C[37] := ":";
          ⋮                 }          Special characters
C[maxchar] := "*";
```

π_1 cannot be finite. As such, a purely sequential (character-by-character) decomposition is not possible. Thus, for our purposes, we have to chose a structured

(ii) algorithm

in order to construct concrete examples of self-reproducing programs. (i) and (ii) represent the two most important aspects of self-reproducing programs. The subsequent construction challenges will depend on the artful decomposition of the program and finding a suitable output algorithm.

3.2.3 An array-based approach

π_2 picks up on Sect. 3.2.2's full decomposition of π into single characters, the difference being π_2's implicit algorithmic—rather than π_1's explicitly accessible—decomposition. Algorithmically, we construct π's text through single characters. Valid characters for this algorithm are accessible through an array `character array C[0 : maxchar]` The array `C` (Table 5) contains sufficient elements to match every character used to construct SIMULA programs. The algorithm construct the program text π_2 from successive array `C` elements (see Table 6). We rewrite program π_2 (Table 7) as

Table 6 π_2: Sketch of algorithm

```
while not p do                 | I's type is integer
begin <compute new I> ;        | p is a predicate
OUTCHAR(C[I]);                 | that stops the algorithm
<set p>;                       | as soon as text π2 is printed
end
```

Table 7 π_2: Decomposition into 4 parts

$\pi_2 =$

`begin integer I;` ⋮	Ia

`C[0] := "A";`
 `C[1] := "B";`
 ⋮
 `C[maxchar] := "*";`
	II

`while not p do`
 `begin <compute new I> ;`
 `OUTCHAR(C[I]);`
 `<set p>;`
 `end`
	III

`end`	Ib

a decomposition into four parts: The prologue and epilogue (`Ia , Ib`), the character array initialization (`II`) and the algorithm that realizes the construction (`III`). Implementation of (`Ia, Ib, II`) is straightforward, even the algorithm (`III`) looks manageable. All that remains to be written is the SIMULA equivalent of `<compute new I>` and `<set p >`, where the former will prove to be the harder task. As a start, we want to examine further what is expected in general of the algorithm in `III` and of `<compute new I>` in particular.

Definition 3.1 Let D be the set of all valid characters in SIMULA programs:

$D \in \{a, b, \ldots, z, 0, 1, \ldots, 9, ;, :, \ldots, *\}$

Then $\forall \alpha \in D$, let $i_\alpha \in \mathbb{N}_0$ be the index of char $\alpha \in D$ in array `C`:

$\alpha = C[i_\alpha]$

Lemma 3.1 *D^* is encoded by $\mathbb{N}_0^*$ through the mapping $\delta : D^* \to \mathbb{N}_0^*$, where $\delta(\epsilon) = \epsilon$, $\delta(w\alpha) = \delta(w)i_\alpha$ $\forall w \in D^*$, $\alpha \in D$.*

Proof (i) $\delta(x)$ is defined for every $x \in D^*$; hence, δ is total. δ is trivially computable.

(ii) δ is injective since every character in D is stored exactly once in array C.

(iii) Let $\overline{j} = j_1 \ldots j_n \in \mathbb{N}_0^*$. Then $\overline{j}$ is in $\delta(D^*)$ when every $j_k, k \in [n]$ is from $\{0, \ldots, maxchar\}$. Hence $\delta(D^*)$ is decidable.

(iv) Let $\overline{j} = j_1 \ldots j_n \in \delta(D^*)$. For every $j_k, k \in [n]$, the character in D can be selected through j_k and character array C. A maximum of $n * maxchar$ comparisons are necessary to recover the inverse image of $\overline{j}$ under δ. Hence, the inverse δ^{-1} is computable.

It follows from (i) to (iv) that δ is an encoding (see Definition 2.14). □

Table 8 π_2: Generate i_j

set i_1; $i_{j+1} := F(i_j);\ j \in [l(\pi_2) - 1]$

Remark Every SIMULA program π, as a finite string in D^*, has an encoding $\delta(\pi)$ in $\mathbb{N}_0^*$.

A new value for `I` is calculated in every iteration of π_2's while loop. Hence, during execution, `I` takes on a series of values. These values can be represented as a string in $\mathbb{N}_0^*$:

$$I = i_1, i_2, \ldots, i_{l(\pi_2)}, \quad i_j \in \mathbb{N}_0 \quad \text{for all} j \in [l\,(\pi_2)]\,. \tag{4}$$

In order for π_2 to output its own text, the following equality must hold:

$$C[i_1]\, C[i_2] \ldots C[i_{l(\pi_2)}] \stackrel{!}{=} \pi_2. \tag{5}$$

Thus, we state that

- `<compute new I>` computes the incremental encoding of π_2 through δ,
- the purpose of the entire π_2 algorithm is the decoding of the encoding, i.e. the realization of δ^{-1}

We are still left with the problem of computing $i_j, j \in [l(\pi_2)]$, which has to be done iteratively through a function $F : \mathbb{N}_0 \to \mathbb{N}_0$ (Table 8):

Function `F` can be written as a procedure and merged into part `Ia` of π_2. The statement `<compute new I>` is rewritten as $I := F(I)$.

Since we strive to implement a self-reproducing SIMULA program on a concrete computer systems, we required the following from function `F`:

a) `F` has to be computable in reasonable time.
b) The intermediate computation results of `F` cannot exceed the numerical range of the computer system.

It is possible for a given `F` to be independent of `I`.

3.2.4 Choosing iteration function F

This section discusses two viable candidate functions for `F`.

A modulo-based iteration function

We extend encoding δ through "Gödelization" (see Definition 2.15). We define the mapping f_δ.

Table 9 π_2: Procedure F

```
integer procedure F;
begin   F := X mod (maxchar+1);
        X := X // (maxchar+1);
end
```

Table 10 π_2: Specifying iteration step r

```
X := q;
Y := 1;
  while Y ≤ l("π2 without string q") do
  begin I:=F;
        OUTCHAR(C[I]);
        if Y=r then OUTINT(q,...);
        Y:=Y+1;
   end
```

Definition 3.2 Define the mapping $f_\delta : \mathbb{N}_0^* \to \mathbb{N}_0$ as follows:

$$f_{\delta(\bar{i})} = \begin{cases} 0, & \text{if } \bar{i} = \epsilon \\ \sum_{j=1}^{k} i_j (maxchar+1)^{j-1}, & \text{if } \bar{i} = i_1 \ldots i_k \end{cases} \quad (6)$$

If $\bar{i} \in \delta(D^*)$, then each i_j, $j \in [k]$ is smaller or equal to $(maxchar + 1)$. Hence, the injective restriction of f_δ on $\delta(D^*)$ maps the elements of $\delta(D^*)$ to $\mathbb{N}_0$. We have:

Lemma 3.2 *Given $f := f_\delta \circ \delta$, the mapping $f : D^* \to \mathbb{N}_0$ represents the Gödelization of D^*.*

Proof Self-evident. □

Of interest to us is the fact that $f(x)\ \forall x \in D^*$ can be effectively used to recover $\delta(x)$. This is *a fortiori* the case for $f(\pi_2)$. Table 9 sketches an iteration procedure for $\delta(\pi_2)$'s construction, provided X's starting point is chosen as $f(\pi_2)$. // denotes integer division. Since we disallowed reading in X's starting value of $f(\pi_2)$ from an external source, π_2 has to contain the assignment $X := f(\pi_2)$. As a whole number, π_2 can contain $f(\pi_2)$ in its program text. It is albeit unknown at construction time of π_2, but can be subsequently computed. At construction time, we omit listing $f(\pi_2)$ in the statement $X := f(\pi_2)$;. Only after construction can we compute $q := f(\text{“}\pi_2 \text{ without string } f(\pi_2)\text{”})$. With number q as starting value for X, π_2 can only reproduce its program text without q. We remedy this by altering π_2.

We can pinpoint with relative ease at which step the algorithm has to output $f(\pi_2)$. Let this be the r^{th} iteration step, and the number r becomes part of the program. The new algorithm for π_2, as well as the final π_2 is given in Tables 10 and 11.

Array C need not contain any characters that do not appear in π_2, as to keep its size as small as possible. Values for r, q, and l("π_2 without string q") are computed after construction of the program. It is easy to see that π_2 is self-reproducing.

Although syntactically correct, π_2 cannot be implemented on a concrete computer. The value of q is too large to be represented on common computer systems. We show this by calculating a lower bound: At a minimum, π_2 contains at least 32 characters, i.e. maxchar ≥ 32. The minimum length of π_2 including the necessary blank spaces as delimiters amounts to

$$\texttt{l("}\pi_2\texttt{ without string q")} > 700$$

It follows from Definition 3.2 and the definition of q that $q > 32^{700-1}$. From the raw estimate, it is apparent that q's size exceeds the numerical range of common computers.

Table 11 π_2: Amended with C, F and r

```
π2 = begin
     integer              I,X,Y
     character            array       C[1:maxchar];
     integer              procedure F;
                          begin       F := X mod (maxchar+1);
                                      X := X // (maxchar+1);
                          end
     C[0] := "A";
             ⋮
     C[maxchar] := "*";
     X := q;
     Y := 1;
     while                Y ≤         l("π2 without string q") do
                          begin       I:=F;
                                      OUTCHAR(C[I]);
                                      if Y=r then OUTINT(q,...);
                                      Y:=Y+1;
                          end
     end
```

A Gödelization-based iteration function

Section 2.5 introduced the Gödelization $g : C^* \to \mathbb{N}_0$. Similarly, $g_D : D^* \to \mathbb{N}_0$ can be constructed by substituting D for C and the mapping $H_D : D \to \mathbb{N}_0$ for $H : C \to \mathbb{N}_0$, with $H_D(\alpha) := i_\alpha, \forall \alpha \in D$. Since for every Gödel number $g(w)$, $w \in C^*$, we can effectively specify the inverse image w, this is applicable for every Gödel number $g_D(v)$, $v \in D^*$, as well. Along the lines of Sect. 3.2.4, we can construct a procedure F' to iteratively compute $\delta(\pi_2')$, with $q' := g_d(\text{“}\pi_2' \text{ without number } g_D(\pi_2')\text{”})$ as the iteration's starting value. Subsequently, we obtain π_2', a self-reproducing program along the lines of Sect. 3.2.4, with the identical reservation that despite its syntactic correctness, it is not implementable because of the magnitude of q'. Again, we show this by calculating a lower bound. The minimum length of π_2' is 650 characters. Then

$$q > p_1^{i_B} p_2^{i_E} p_3^{i_G} p_4^{i_I} p_5^{i_N} \ldots p_{588}^{i_\alpha+1} - 1, \quad \alpha \in D$$

Since the fifth prime number 11 is already greater than 10, and character a with $i_a = 0$ occurs rarely in π_2', we surely have $q' > 10^{600}$, which again exceeds the numerical range of common computers.

3.2.5 A string-based SIMULA *program* π_3

Section 3.2.4 introduced syntactically correct, albeit unrealizable, self-reproducing programs π_2 and π_2'. We move from constructing realizable iterator functions and concomitant start values to focussing our attention on the programs's text decomposition and algorithm. We change π_2 to π_3 as follows:

(i) Decomposition: In contrast to π_2, the decomposition units in π_3 are partial strings, not individual characters. We substitute `text array C[1:maxtext]` for `character array C[1:maxchar]`. This introduces the SIMULA `text` concept for our purposes.

(ii) Algorithm: The algorithm of π_3 strives to reconstruct the program text of π_3 from partial strings stored in field `C`. Every field component is referred to by its index. Hence, π_3 can be encoded by successive indices:

$$\pi_3 \mapsto i_1, \ldots, i_k \ , \ i_j \in \{1, \ldots, maxtext\}$$

The index series is written into text variable `X`. We can access individual i_j's in `X` through the standard[6] `text` procedures `SUB` and `GETINT`. Hence π_3's algorithm needs merely to iterate sequentially through text `X` and output the text $C[i_j]$ for every i_j.

[6] See [19].

Table 12 π_3: Specifying iteration step r

```
X := COPY("i1,...,ik");
for I:=1 step 1 until k do
begin
      <compute next ij in X>;
      OUTCHAR(C[ij]);
      if I=r then OUTTEXT(X);
end
```

By leveraging SIMULA's text concept for π_3, we avoid the integer representation difficulties of q and q' that plagued π_2 and π_2', respectively. `X` does not contain its own encoding, however; for this reason, output of text `X` is handled separately. Taking our cue from Sect. 3.2.4.1—where the number `X` caused us similar grief—we use an easily computable number r to output `X`. Tables 12 and 13 illustrate this extension. The whitespace in line 19) serves human readability, it does not affect program behavior. Lines 21) and 23) induce the scanning of text `X`.

Verifying π_3

The algorithmic section of π_3 sequentially works through exactly 105 numbers stored in `X`. Each number j is used to output a text `C[j]`. At first, `C[1]` and `C[2]` are printed, which represent the first two program lines. Lines 3)–18) are printed by the next 96 numbers, which consist of 16 groups. Each group is delimited by the pair `21,...,23` which denotes exactly one program line. The group makeup is given in Table 14. This makeup corresponds exactly to the general program line makeup of lines 3)–18). Hence, it can be seen in conjunction with the program's encoding that these numeric groups will print lines 3)–18).

After these 96 numbers have been parsed, we process the number 3 which prints `x:=copy(")`. Concurrently, the `for` loop variable `I` takes on the value 99, since we have printed out 99 numbers so far. For this reason, it is now the turn of text `X` to be printed. Processing number 23, results in printing line 19); lines 20)–24) are outputted with the numbers 4,11,12,13 and 14. The loop variable has then reached 105 and the algorithm terminates.

Improving π_3

(i) Text variables `C[1]...C[33]` as well as `X` contain the partial strings of the program. Recurring strings are of particular importance; we list them in Table 15. These strings in their entirety constitute the structural 'building units' of π_3. The strings in Table 15 are essential and cannot be done without, but the composition of the other strings seems a bit arbitrary. It is not clear, for instance, why `C[1]` and `C[2]` cannot be combined.

Table 13 π_3: Partial strings construction

```
π3 =
1)   begin     integer      I,S,Z; text X;
2)             text array   C[1:34];
3)   C[1] :=   COPY("BEGIN INTEGER I,S,Z; TEXT X;");
4)   C[2] :=   COPY("TEXT ARRAY C [1:34];");
5)   C[3] :=   COPY("X:=COPY(""")");
6)   C[4] :=   COPY("FOR I :=1 STEP 1 UNTIL 105 DO ");
7)   C[11] :=  COPY("BEGIN S :=X.SUB(Z+1,2).GETINT;");
8)   C[12] :=  COPY("OUTTEXT(C[S])");
9)   C[13] :=  COPY("Z:=(IF S<10 THEN 2 ELSE 3)+Z;");
10)  C[14] :=  COPY("IF I=99 THEN OUTTEXT(X) END END");
11)  C[21] :=  COPY("C[");
12)  C[22] :=  COPY(" ]:=COPY(""");
13)  C[23] :=  COPY(""");");
14)  C[24] :=  COPY("""");
15)  C[31] :=  COPY("1");
16)  C[32] :=  COPY("2");
17)  C[33] :=  COPY("3");
18)  C[34] :=  COPY("4");
19)  COPY("1,2,
               21, 31, 22, 1, 23,
               21, 32, 22, 2, 23,
               21, 33, 22, 3, 24, 23,
               21, 34, 22, 4, 23,
               21, 31, 31, 22, 11, 23,
               21, 31, 32, 22, 12, 23,
               21, 31, 33, 22, 13, 23,
               21, 31, 34, 22, 14, 23,
               21, 32, 31, 22, 21, 23,
               21, 32, 32, 22, 22, 24, 23,
               21, 32, 33, 22, 24, 23, 23,
               21, 32, 34, 22, 24, 24, 23,
               21, 31, 31, 22, 31, 23,
               21, 32, 31, 22, 32, 23,
               21, 33, 31, 22, 33, 23,
               21, 34, 31, 22, 34, 23,
               3,23,4,11,12,13,14,");
20)  for       I:=1 step 1  until 105 do
21)  begin     S:=X.SUB(Z+1,2).GETINT;
22)            OUTTEXT(C[S]);
23)            Z:=(if S<10 then 2 else 3)+Z;
24)            if I=99 then OUTTEXT(X) end end
```

Table 14 π_3: Group makeup

```
      21,     ≡     C[
[number,]     ≡     digit 1 or 2 or 3
  number,     ≡     digit 1 or 2 or 3 or 4
      22,     ≡     ]:=copy("
    [24,]     ≡     "
  number,     ≡     text
    [24,]     ≡     "
      23,     ≡     ");
```

Table 15 π_3: Recurring strings

```
C[21]   =   C[
C[22]   =   ]:=copy("
C[23]   =   ");
C[24]   =   "
C[31]   =   1
C[32]   =   2
C[33]   =   3
C[34]   =   4
```

In general, we specify the largest partial strings that do not contain a `"`. For instance, `xxx"xxx` would have to be specified as

k) `C[j] := copy("xxx""xxx");`

To be sure, the number j in text `X` prints `xxx"xxx`, but no sequence of numbers can be specified to print line k).

$21, \ldots, 22, j, 23$

Table 16 π_3': Logical improvements

$\pi_3' =$

```
1)   begin     integer      I,S,Z;
2)             text array   C[1:31];
3)   C[1] :=   COPY("BEGIN INTEGER I,S,Z; TEXT ARRAY[1:31];C[1]:=COPY("""");
4)   C[2] :=   COPY("C[");
5)   C[3] :=   COPY(" ]:=COPY("""");
6)   C[11] :=  COPY("""");");
7)   C[12] :=  COPY("""");
8)   C[13] :=  COPY("1");
16)  C[21] :=  COPY("2");
17)  C[22] :=  COPY("3");
18)  C[23] :=  COPY("1,1,12,11,
                2, 21, 3, 2, 11,
                2, 22, 3, 3, 12, 11,
                2, 13, 13, 3, 12, 11, 11,
                2, 13, 21, 3, 12, 12, 11,
                2, 13, 22, 3, 13, 11,
                2, 13, 13, 3, 21, 11,
                2, 13, 21, 3, 22, 11,
                2, 21, 22, 3, 23, 11,
                2, 21, 22, 3, 31, 11,
                2, 22, 13, 3, 31, 11, 31,");
19)  C[31] :=  COPY("FOR I:=1 STEP 1 UNTIL 60 DO BEGIN S:=C[23].SUB(Z+1,2).GETINT;
                OUTTEXT(C[S]);Z:=(IF S<10 THEN 2 ELSE 3)+Z END END");
20)  for       I:=1 step 1  until 60 do
21)  begin     S:=C[23].SUB(Z+1,2).GETINT;
22)            OUTTEXT(C[S]);
23)            Z:=(if S<10 then 2 else 3)+Z;
24)    end
25)  end
```

prints

C[j] := copy("xxx"xxx");

There is a " missing in the middle of "xxx"xxx". It cannot be inserted by printing C[24] since it is missing in the middle of C[j]'s text. A successful insertion of number 24 into X is only possible if the single quote appears at the beginning or at the end of C[j] (compare lines 12)–14) and the corresponding numeric groups on text X).

(ii) π_3 makes use of a text array C and a text X, both of which only contain text constants. Hence, there is no justifying special treatment for X. We subsequently extend field C with component C[x].C[x] is assigned X and hence we jettison the extraneous algorithmic special case for X (the if condition in the for loop). C[x] is printed when number x is processed; x is itself an element of C[x].

(iii) Points (i) and (ii) hint at a parsimonious makeup of C. Fewer components require fewer digits to address them unambiguously. We may even eliminate line 18), since digit 4 is not used for addressing.

Program π_3' (Table 16) incorporates the changes listed in points i)-iii). Component C[1] and C[31] are particularly conspicuous examples of largest partial strings alluded to in i). From a logical functionality point of view, there are no more improvements to be made in π_3. However, "textually" we can shorten the program further with the following changes:

(iv) C components can be addressed in such a way that commonly recurring addresses in C[23] are as short as possible (Table 17). In order to minimize length, the three shorter, one-digit addresses should be used most frequently. This is not the case: Address 1 occurs just twice, whereas double-digit address 11 occurs ten times. We optimize our addressing scheme by interchanging the contents of C[1] and C[11] and by reflecting this change in C[23]. We thus save 8 characters.

3.2.6 Implementing π_3

Program π_3 dumps its program text into the standard stream SYSOUT. Since the program text is transmitted in one single uninterrupted string, we have to increase the size of the SYSOUT buffer, lest we incur a runtime exception during execution. We augment π_3 with the following statement

SYSOUT.IMAGE := BLANKS(200);

Table 17 π_3': Length improvements

Address	Occurrence in C[23]	Total characters
1	2	1*3=2
2	10	1*10=10
3	10	1*10=10
11	10	2*10=20
12	4	2*4=8
13	6	2*6=12
21	8	2*8=16
22	5	2*5=10
23	1	2*1=2
31	2	2*2=4

and adjust the text constant `C[31]` accordingly (see Appendix A.1 in Electronic Supplementary Material). Our SIMULA compiler's input length is capped at 72 characters. The output of π_3 and π_3' would thus have to be subdivided into blocks of 72 characters to be compilable, which is not the case for either program. Appendix A.2 in ESM presents a version π_3'' of π_3 whose output is blocked into 72-character lines through a complicated assignment set. π_3'''s output is compilable and constitutes an executable SIMULA program equivalent to π_3''.

3.2.7 A procedure-based program π_4

Section 3.2.5 introduced a self-reproducing program π_3 containing its own text stored in the form of partial strings, which then had to be printed out in the right order. The self-reproducing SIMULA program π_4 combines the storage and printing of said partial strings. Instead of π_3's

```
C[address] := copy("text");
```

π_4 contains

```
procedure name; OUTTEXT("text");
```

The decomposition of the program text remains the same, but the algorithm in π_4 consists of consecutive procedure calls. Thus, we no longer need to use the SIMULA programming language's text concept: We merely need the ability to use text constants as print statement arguments (Table 18).

Verifying π_4

The first statement of π_4 (`AA`) prints out the first program line. The subsequent nine program lines are printed by lines 12)–20) which can easily be checked by tracing the visually ordered procedure calls in π_4. The last statement `AB` induces the output of lines 11)–22), since procedure `AB`'s text constant contains the procedure calls—the algorithmic part–of π_4. With the execution of `AB`, the output of π_4 catches up with the execution of π_4.

Remark The self-reproducing SIMULA program π_4 requires the use of string constants as data, and structurally the ability to consecutively execute procedures. These aren't just

Table 18 π_4: Procedural decomposition

```
π4 =
1)  begin
2)  procedure AA;  OUTTEXT("BEGIN");
3)  procedure C;   OUTTEXT("PROCEDURE");
4)  procedure A;   OUTTEXT(";OUTTEXT(""");
5)  procedure B;   OUTTEXT("");");
6)  procedure AC;  OUTTEXT("""");
7)  procedure BA;  OUTTEXT("A");
8)  procedure BB;  OUTTEXT("B");
9)  procedure BC;  OUTTEXT("C");
10) procedure AB;  OUTTEXT("AA;C;BA;BA;A;AA;B;C;BC;A;C;B;C;BA
                   ;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;AC;AC;B;C;BB;BA;A;
                   BA;B;C;BB;BB;A;BB;B;C;BB;BC;A;BC;B;C;BA;BB;A;AB;B;AB
                   END");
11) AA;
12) C; BA; BA; A; AA; B;
13) C; BC; A; C; B;
14) C; BA; A; A; AC; B;
15) C; BB; A; AC; B; B;
16) C; BA; BC; A; AC; AC; B;
17) C; BB; BA; A; BA; B;
18) C; BB; BB; A; BB; B;
19) C; BB; BC; A; BC; B;
20) C; BA; BB; A; AB; B;
21) AB;
22) end
```

SIMULA-specific idioms, but available in most higher-level languages. As such, self-reproducing programs similar to π_4 must exist in almost all other high level languages.

3.2.8 Implementing π_4

The buffer size limitation of the SIMULA compiler (Sect. 3.2.6) has to be taken into account here, as well. An executable program π_4' can be built from π_4 by including the assignment

```
SYSOUT.IMAGE := BLANKS(200);
```

into the text constant of procedure AB. We proceed—analogous to Sect. 3.2.6—to derive an executable program π_4'' by again taking into account the compiler output restrictions. We do this by

- adding procedure `procedure Q; OUTIMAGE;`
- decomposing the text constant of procedure AB into procedures AB, CA, CB, CC, AAA and AAB.

We incur some statement bloat in π_4'' to accommodate the output of procedures CA to AAB. Appendices A.3 and A.4 present π_4's derived programs π_4' and π_4'', respectively.

3.3 Self-reproducing programs in PASCAL [10]

This section introduces self-reproducing programs in PASCAL, a non-block oriented language that recognizes text constants, but not text variables. As such, it is not self-evident how to adapt the SIMULA program π_3 into an equivalent, self-reproducing PASCAL program. However, keeping previous remarks in mind, we will have no problems implementing a PASCAL version of π_4 (Sect. 3.2.6).

3.3.1 A string-based PASCAL *program* π_5

Despite the lack of text variables, we shall try to port program π_3 into a self-reproducing PASCAL program π_5. There are several ways to overcome this language restriction, of which we present two:

(i) We use character arrays to store the π_3 strings. Field C becomes two-dimensional

```
var C : array[1..maxtext, 1..maxlength]
  of char;
```

where `maxlength` denotes the length of the largest partial string that arises during the decomposition of π_5. Every row in C contains exactly one string from the decomposition of π_5. Having successfully approached the issue of string storage, we turn our attention to the algorithmic part of π_5. Since we have no strings and therefore no GETINT-equivalent procedure, we introduce a mapping from every "string" `C[j,...]` to a letter in the alphabet (Table 19). Using C's rows, we can construct and therefore describe π_5's program text with a finite sequence of letters. This sequence is stored in `C[maxtext,...]`, the content of which merely has to be processed for printing by π_5's algorithm (Table 20). Unfortunately, we find the left hand side of the `case` structure rife with single quotes '. The single quote ' in PASCAL is equivalent to the SIMULA double quote ". This has structural ramifications in that the algorithm would have to be decomposed into a lot of partial strings (see Sect. 3.2.5), resulting in a unwieldy program. Fortunately, we can negotiate this problem with PASCAL procedure ORD which maps char to integer (Table 21). There remain a few more obstacles towards the realization of π_5.

Table 19 π_5: Mapping 'strings' to letters

`C[1,...]`	$\mapsto$	`a`
`C[2,...]`	$\mapsto$	`b`
`C[3,...]`	$\mapsto$	`c`
`C[4,...]`	$\mapsto$	`d`
$\vdots$		$\vdots$
`C[maxtext,...]`	$\mapsto$	$\text{maxtext}^{\text{th}}$ `letter`

Table 20 π_5: Algorithm for `C[maxtext,...]`

```
for I:=1 to <length of C[maxtext,...]> do
begin case C[maxtext,I] of
      'a': <output C[1,...]>
      'b': <output C[2,...]>
                        ⋮
end
```

Table 21 π_5: Improved algorithm with ORD

```
for I:=1 to <length of C[maxtext,...]> do
begin HELP := ORD(C[maxtext,I])
      case HELP of
      <ORD(a)>: <output C[1,...]>
      <ORD(b)>: <output C[2,...]>
                        ⋮
end
```

- π_5 needs to contain a special output procedure to handle statements of the type `<length of C[I,...]>`
- The rows of C contain in general many blanks, the output of which we try to avoid, if possible.

All in all, we could construct a syntactically correct, albeit unmanageable program π_5.

Table 22 π_5: ORD letter mapping

```
ORD(a) = 193
ORD(b) = 194
ORD(c) = 195
ORD(d) = 196
ORD(e) = 197
ORD(f) = 198
ORD(g) = 199
ORD(h) = 200
ORD(i) = 201
ORD(j) = 202
ORD(k) = 203
```

Table 23 π_5: Mapping procedures to letters

```
procedure A ↦ a
procedure B ↦ b
procedure C ↦ c
procedure AA ↦ d
procedure AB ↦ e
procedure AC ↦ f
procedure BA ↦ g
procedure BB ↦ h
procedure BC ↦ i
procedure CA ↦ j
procedure CB ↦ k
```

(ii) In order to circumvent the difficulties presented in (i), we abandon the idea of storing π_5's partial strings in a two-dimensional character array. We turn to a scheme similar to Sect. 3.2.7's SIMULA program π_4 in that we implicitly store the strings in output procedures. π_5's algorithm's, given in (i), is modified in that now, we map the procedure instead of the "strings" to the alphabet. These changes make π_5 much more manageable. The specific PASCAL compiler-implementation dependent ORD letter values and procedure mapping are given in Tables 22 and 23. The revised π_5 is presented in Table 24.

Verifying π_5

We discussed the purpose of the `for` loop. Every character in `X` maps to one alternative in the `case` statement which prints out a partial string of π_5's program text. Through parsing of `X`, it is easy to verify (akin to π_3's verification) that π_5 does reproduce itself.

3.3.2 Implementing π_5

We implement π_5 such that the number of characters per printed line is limited to 132 characters. Thus, we insert the procedure

```
procedure Q; begin WRITELN end;
```

into π_5. Procedure `A` and `B`'s longish text constants warrant their breaking up into multiple procedures which lengthen the program text for π_5, forcing procedure `CB` to be broken up, as well (Table 25).

Variable `X` contains π_5's encoding; the added procedures, however, exceed the variable's holding capacity (an artifact of the specific PASCAL implementation used here). We introduce another variable `Y: array [1 .. 68] of char;` to handle the space requirements of π_5's procedural extensions, which in turn necessitate the generation of procedure `CCA`. Appendix A.5 in ESM show the changes to π_5 algorithm in detail. The new procedure encodings are given in Table 26.

3.3.3 A procedure-based PASCAL program π_6

Since all the languages elements used in Sect. 3.2.7's are present in PASCAL, we can translate π_4 into a self-reproducing PASCAL program π_6 (Table 27).

Verifying π_6

Verification follows directly from π_4's verification in Sect. 3.2.7.

3.3.4 Implementing π_6

π_6 writes its program text in one go without blocking into file `OUTPUT`. Again, the string's length exceeds the buffer capacities of both the `SIEMENS` printer and the PASCAL compiler. As such, it can be neither printed nor compiled. We negotiate this limitation by blocking the output into chunks of 132 characters (which corresponds to the `SIEMENS` printer buffer). Hence, we have to repeatedly include the `WRITELN` procedure into π_6. We shorten it as such

```
procedure Q : begin WRITELN end;
```

This change affects the procedural definition part of π_6 with ramifications for procedure `AA`. In order to break up `AB`'s long text constant into two procedures, we need to implement yet another procedure `CA`. Appendix A.6 in ESM shows π_6 implementing these changes.

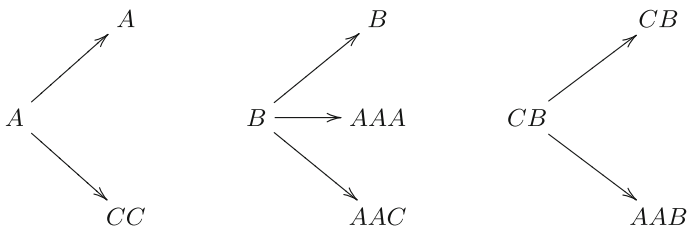
3.4 Self-reproducing program in the SIEMENS assembly language

This section gives a few examples of self-reproducing programs written in an assembly language (here `SIEMENS` assembler). Since assembly programs may directly access and read the memory areas in which they are situated, our task becomes somewhat easier. Self-reproducing programs

Table 24 π_5: Procedural decomposition

```
π5 =
1)  program SELF(OUTPUT)
2)   var I, HELP:  integer;
3)             X:  array[1..72] of char;
4)   procedure A;  begin WRITE('PROGRAM SELF(OUTPUT); VAR I, HELP: INTEGER;
                   X: ARRAY[1..72] OF CHAR;') end;
5)   procedure B;  begin WRITE(''); FOR I:=1 72 DO BEGIN
                   HELP := ORD (X[I]0; CASE HELP OF 193:A; 194:B; 195:C;
                   196:AA; 197:AB;198:AC; 199:BA; 200:BB; 201:BC;
                   202:CA; 203:CB; END; END; WRITELN END.') end;
6)   procedure C;  begin WRITE('PROCEDURE') end;
7)  procedure AA;  begin WRITE(';BEGIN WRITE('') end;
8)  procedure AB;  begin WRITE('') END;') end;
9)  procedure AC;  begin WRITE('"') end;
10) procedure BA;  begin WRITE('A') end;
11) procedure BB;  begin WRITE('B') end;
12) procedure BC;  begin WRITE('C') end;
13) procedure CA;  begin WRITE('BEGIN X :='') end;
14) procedure CB;  begin WRITE('ACGDAECHDFBECIDCECGGDDFECGHDFEE')
                   CGIDFFECHGDGECHHDHECHIDIECIGDJFECIHDKEJKB') end;
15) begin
16) X:='ACGDAECHDFBECIDCECGGDDFECGHDFEECGIDFFECHGDGECHHDHEC
    HIDIECIGDJFECIHDKEJKB';
17) for I:=1 to 72 do
18) begin HELP := ORD(X[I]);
19)                case HELP of
20)                193 : A
21)                194 : B
22)                195 : C
23)                196 : AA
24)                197 : AB
25)                198 : AC
26)                199 : BA
27)                200 : BB
28)                201 : BC
29)                202 : CA
30)                203 : CB
31)                end;
32) end; WRITELN
33) end.
```

Table 25 π_5: Breaking up A, B and CB



needn't output their program text in assembly code; rather, they may write the machine code directly into memory (see Sect. 1.2). All addressing in the subsequent examples are relative to the program counter PCR which guarantees that the copies' functionality as well as their self-reproductive capability are preserved. We explain the programs in as much detail as the scope of this thesis allows. For details on the SIEMENS assembler, the interested reader is referred to [22,23].

Table 26 π_5: New procedure mappings

CC	$\mapsto$	l	AAB	$\mapsto$	n
AAA	$\mapsto$	m	CCA	$\mapsto$	q
AAC	$\mapsto$	p	Q	$\mapsto$	o

with

ORD(l)	=	211	ORD(o)	=	214
ORD(m)	=	212	ORD(p)	=	215
ORD(n)	=	213	ORD(q)	=	216

Table 27 π_6: Procedural decomposition

```
π6 =
1)  program PI6(OUTPUT)
2)  procedure AA;  begin WRITE('PROGRAM PI6(OUTPUT); PROCEDURE AA; BEGIN WRITE(''') end;
3)  procedure C;   begin WRITE('PROCEDURE ') end;
4)  procedure A;   begin WRITE(';BEGIN WRITE(''') end;
5)  procedure B;   begin WRITE(''') END;') end;
6)  procedure AC;  begin WRITE('"') end;
7)  procedure BA;  begin WRITE('A') end;
8)  procedure BB;  begin WRITE('B') end;
9)  procedure BC;  begin WRITE('C') end;
10) procedure AB;  begin WRITE('BEGIN AA;AA;AC;B;C;BC;A;C;B;C;BA
                   ;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;AC;AC;B;C;BB;BA;A;BA;B
                   ;C;BB;BB;A;BB;B;C;BB;BC;A;BC;B;C;BA;BB;A;AB;B;AB;WRITELN END.') end;
11) begin          AA; AA; AC; B;
12)                C; BC; A; C; B;
13)                C; BA; A; A; B;
14)                C; BB; A; AC B; AC; B;
15)                C; BA; BC; A; AC; AC; B;
16)                C; BB; BA; A; BA; B;
17)                C; BB; BB; A; BB; B;
18)                C; BB; BC; A; BC; B;
19)                C; BA; BB; A; AB; B; AB; WRITELN end
```

Table 28 PROG1: Self-reproducing SIEMENS assembly

Line #	name	mnemonic opcode	operands	command format	command length (bytes)
1)	PROG1	START			
2)		BALR	1,00	RR	2
3)		LA	2,2(0,0)	RX	4
4)		SR	1,2	RR	2
5)		LM	4,8,0(1)	RS	4
6)		STM	4,8,64(1)	RS	4
7)		SVC	X'5B'	RR	2
8)		END			
			Program length in bytes		18

Example 3.1 The self-reproducing assembly program PROG1 (Table 28) copies its machine code to the 64th byte relative to PROG1's first statement. Assembly instructions START and END are not assembled into machine code and can be ignored for copying purposes. The programs's first executable statement is

```
BALR 1,00
```

which loads the current program counter PCR into instruction register R1. Since the program counter is incremented by the length of the statement to be executed before actual execution, R1 contains after execution the starting address of PROG1 plus the length of BALR statement. Since BALR is RR, the length is 2. Statement

```
LA 2,2(0,0)
```

loads the literal 2 into register R2. The SR (Subtract Register) command

```
SR 1,2
```

subtracts the content's of R2 from R1. Hence, after execution, R1 contains the program's starting address, and we shall use R1 subsequently as a base register. Line 5 denotes a LM (load multiple) command

```
LM 4,8,0(1)
```

which loads consecutive words in memory into consecutive multi-purpose registers (as such, 16 is the limit). The first two operands denote the first and last register, respectively; and the last operand gives the starting address of the memory location to be copied. In our case it is PROG1's starting address which explains the composition of the third operand: R1 as base address plus an offset of 0. Program PROG1's length is 18 bytes. Since memory word length is 4 bytes, five multi-purpose registers (R4–R8) are sufficient to contain the entire program. The next statement

```
STM 4,8,64(1)
```

Table 29 PROG2: Self-reproducing SIEMENS assembly

Line #	name	mnemonic opcode	operands	command format	command length (bytes)
1)	PROG1	START			
2)		BALR	1,00	RR	2
3)		LA	2,2(0,0)	RX	4
4)		SR	1,2	RR	2
5)		MVC	64(60,1),0(1)	SS	6
6)		SVC	X'5B'	RR	2
7)		END			
			Program length in bytes		16

Table 30 PROG2: Template for copying

```
1) PROGRAM CSECT
2)         BALR        1,00
3)         LA          2,2(0,0)
4)         SR          1,2
5)         MVC         <location offset of copy>(<# bytes to be copied>,1),0,1
6)         ⋮                      program region
7)         ⋮                      to be copied
⋮          ⋮                      (max 2^8 − 16 bytes)
           END
```

is a STM (multiple store) command and serves as the storage counterpart of the LM command. It stores the contents of registers R4 to R8 consecutively at the address specified by the third operand. Again, we use register R1 as a base address and an offset of 64. After execution of STM a copy of PROG1 will be resident in memory, 64 bytes from the base address of the original. The last command

```
SVC X'5B'
```

simply serves to properly terminate PROG1. Appendix B.1 in ESM gives an expansive discussion on PROG1.

The next example PROG2 is 2 bytes shorter than PROG1. These savings are realized by replacing the LM and STM commands by a MVC (move character) statement. In addition to copying itself, PROG2 is also able to copy a certain memory section that follows the program (Table 29).

Example 3.2 Lines 1)–4) are identical to PROG1 in that they load the program base address into register R1. Line 5)'s MVC induces the copying of 60 consecutive bytes, starting at base address R1 plus offset 0 (2nd operand), to the memory address specified by R1 plus offset 64 (1st operand). Since PROG2 is merely 16 bytes long, 44 additional bytes are copied by the MVC statement. Up to 2^8 bytes can be copied using MVC, provided once specifies the range in the appropriate field (60 in our example). PROG2's last two lines 6)–7) correspond to PROG1's lines 7)–8).

PROG2's lines 2)–5) introduces a code snippet that can integrate other assembly code regions or augment entire programs into a self-reproduction format. Table 30 illustrates how to use the program text section length in bytes as an operand for the MVC statement. This way, it is possible to copy up to $2^8 - 16$ bytes' worth of program text into memory. The copy's location offset must be chosen according to the length of the text region to be copied. Our next example PROG3 is a self-reproducing assembly program which hands off control to its copy after execution (Table 31). This is achieved by an unconditional jump to the copy's base address. Since we faithfully copied PROG3, upon execution this process repeats itself; the system's memory is hence iteratively suffused with copies of PROG3. The distance between consecutive copies is constant.

Lines 1)–5) are identical to PROG2 and induce the copying of PROG3 to the 64th byte offset following PROG3's first statement. Line 6)'s LA (load address) command

```
LA 2,64(0,0)
```

loads 64 into R2. The subsequent AR statement

```
AR 1,2
```

serves to increase the contents of base register R1 by 64. After execution of the AR statement, R1 thus contains the starting address of PROG3's copy, control to which is handed off by the BR (uncondition branch) statement

```
BR 1
```

The copy is executed and the process repeats itself with the new copy. Appendix B.3 in ESM illustrates the

Table 31 PROG3: One copy after another

Line #	name	mnemonic opcode	operands	command format	command length (bytes)
1)	PROG3	START			
2)		BALR	1,00	RR	2
3)		LA	2,2(0,0)	RX	4
4)		SR	1,2	RR	2
5)		MVC	64(60,1),0(1)	SS	6
6)		LA	2,64(0,0)	RX	4
7)		AR	1,2	RR	2
8)		BR	1	RR	2
9)		END			
			Program length in bytes		22

Table 32 PROG4: 4 byte chunks

Line #	name	mnemonic opcode	operands	command format	command length (bytes)
1)	PROG4	START			
2)		BALR	1,00	RR	2
3)		LA	2,2(0,0)	RX	4
4)		SR	1,2	RR	2
5)		LA	3,4(0,0)	RX	4
6)		LA	4,48(0,0)	RX	4
7)		LA	10,22(0,0)	RX	4
8)		AR	10,1	RR	2
9)		MVC	64(4,1),0(1)	SS	6
10)		AR	1,3	RR	2
11)		SR	4,3	RR	2
12)		BRP	10	RR	2
13)		SVC	X'5B'	RR	2
14)		END			
			Program length in bytes		36

implementation of PROG3. Since PROG3 never stops, eventually memory exhaustion will lead to abnormal program termination.

The previous examples copied programs en bloc using the MVC, or the LM and STM commands, respectively. The following code example (Table 32) presents a program that copies its code in chunks of 4 bytes, increasing its algorithmic complexity somewhat as well as its program length compared to the previous examples.

Example 3.4 Lines 2)–4) load PROG4's starting address into register R1, which is subsequently used as the base address register. The multipurpose registers R3, R4, and R10 are assigned values of 4, 48, and 22 (lines 5)–7)), respectively; 48 being the number of bytes to be copied into memory by PROG4. The statement

```
AR 10,1
```

adds the start address of PROG4 to the value in register R10. After command AR is executed, R10 holds the branch address used by BRP (branch if positive) in line 12), which corresponds to the address of statement

```
MVC 64(4,1),0(1)
```

in line 9). The MVC statement uses base address register R1 to copy the first 4 bytes of PROG4's machine code into the 64th memory location after PROG4's first statement. R1's value is subsequently increased with statement

```
AR 1,3
```

which adds the content of R3 (which is 4) to R1. The following statement

```
SR 4,3
```

subtracts 4 from register R4, which holds the remaining number of bytes to be copied. Provided the number is positive, some of the 48 bytes remain to be copied, and statement

```
BRP 10
```

branches back to the MVC command in line 9), which in turn copies the next 4 bytes of PROG4, since R1 had already been

Table 33 PROG4: Template for copying

```
1)  PROGRAM CSECT
2)          BALR        1,00
3)          LA          2,2(0,0)
⋮           ⋮
6)          LA          4,<region length>(0,0)
7)          LA          10,22(0,0)
⋮           AR          10,1
⋮           MVC         <location offset of copy>(4,1),0,1
⋮           ⋮
12)         BRP         10
13)         ⋮
⋮           ⋮           } program region
⋮           ⋮           } to be copied
⋮           ⋮
            END
```

increased by 4. The program terminates after 48 bytes have been copied. Since PROG4's length adds up to a mere 36 bytes, 12 consecutive follow-up bytes are copied, as well. As before, statement

```
SVC X'5B'
```

ends the program; for a demonstration, see Appendix B.4 in ESM. Similarly to Example 3.2's PROG2 , lines 1)–12) can be used to integrate large code regions and/or augment entire assembly programs into a self-reproducing program format (see Table 33). In contrast to PROG2's mechanism, however, the region to be copied is no longer limited to $2^8 - 16$ bytes.

4 Variants of self-reproducing programs

Let S in this chapter be a fixed high level language in the usual sense (see Sect. 1.2).

4.1 Motivation

In Sect. 1.2, we gave a definition of self-reproducing programs. The gist of it is: *Let be $\pi \in S$. π is said to be self-reproducing if it outputs its own source code without the help of any input.* If we probe this definition further, the output of any self-reproducing program π in S must satisfy the following requirements:

1. The output of π must contain a syntactically correct program π' in the programming language S.
2. π' must be identical to π.

If we choose to ignore the second condition, then the program generally is no longer self-reproducing; it can at best be called "reproducing".

If π is "reproducing", following possibilities arise:

1. The program π outputs the program π'. The program π' itself produces the program π'' such as $\pi'' = \pi$. Programs π and π'' are certainly not self-reproducing *per se*. However, a certain kind of self-reproduction - one with some intermediary step - is indeed evinced here.
2. The program $\pi = \pi^0$ outputs the program π^1, while π^1 itself outputs the program π^2, and so on... From a general point of view, we have π^i outputs π^{i+1}, $i \geq 0$. For every $i, j \geq 0$, we have $\pi^i \neq \pi^j$ whenever $i \neq j$.

On the other hand, we can tighten our previous definition of self-reproduction by adding additional constraints. All in all, some interesting variants of self-reproduction are conceivable. We will present and study some of them in this chapter. We will also illustrate them by means of examples using real programming languages as SIMULA and PASCAL.

4.2 Infinitively reproducing programs

Definition 4.1 Let π be a (syntactically correct) program in S.

1. a) If π does not consider any input, then π is called (strongly) *reproducing*, if π precisely outputs a syntactically correct program π' in S.
   b) If π considers input, then π is called (strongly) *reproducing*, if π precisely outputs a syntactically correct program π' in S for every valid input value.
2. $R(S)$ describes the set of all reproducing programs in S.

Remark Every (strongly) self-reproducing program is obviously (strongly) reproducing.

Consequently, there exist reproducing programs in the SIMULA and PASCAL programming languages, since there also exist self-reproducing programs in these languages.

Lemma 4.1 *There exist infinitely many reproducing programs which are not self-reproducing in* SIMULA *and* PASCAL *programming languages.*

Proof 1. For every $k \in \mathbb{N}$, the following SIMULA program:

$\pi_{\mathrm{SIM}(k)}$ = begin
OUTTEXT("BEGIN INTEGER I;
$I := k$;
OUTINT(k, < arity of k >)
END")
end

is reproducing, since the text constant, produced by $\pi_{\mathrm{SIM}(k)}$ represents a valid SIMULA program.

2. A similar program can be given in PASCAL programming language:

$\pi_{\mathrm{PAS}(k)}$ = program T(OUTPUT);
begin
WRITE('PROGRAMT(OUTPUT);
BEGIN
WRITE(k)
END.')
end

□

Definition 4.2 Let $(\pi_i)_{i\in\mathbb{N}_0} = \pi_0, \pi_1, \pi_2, \ldots$ be an (infinite) sequence of programs in programming language S. Then, $(\pi_i)_{i\in\mathbb{N}_0}$ is called a *reproduction sequence* if

π_j reproduces π_{j+1}, for every $j \in \mathbb{N}_0$.

From Definition 4.2 it follows that every program in a reproduction sequence can itself be used as the starting program of a new reproduction sequence. We merely need to construct the corresponding subsequence. This fact justifies the following definition.

Definition 4.3 Let $(\pi_i)_{i\in\mathbb{N}_0}$ be a reproduction sequence in the programming language S. Let π_j, $j \in \mathbb{N}_0$ be an element in this sequence.

1. π_j is said to be *infinitely reproducing*.
2. The tail sequence $(\pi_k)^j_{k\in\mathbb{N}_0}$ from $(\pi_i)_{i\in\mathbb{N}_0}$ with $\pi_k = \pi_{j+k}$ for every $k \in \mathbb{N}_0$ is called the *reproduction sequence* of π_j.
3. The set $U(S)$ describes the set of all infinitely reproducing programs in S.

Reproduction sequence of π_0

$$\pi_0 \to \ldots \to \pi_{j-1} \to \pi_j \to \pi_{j+1} \to \ldots$$

Reproduction sequence of π_j

$$\pi_j \to \pi_{j+1} \to \ldots$$

Remark Every self-reproducing programs π is infinitely reproducing. The reproduction sequence of π is constant.

Proposition 4.1 *There exists an infinitely reproducing program in* PASCAL *whose reproduction sequence does not contain any program more than once.*

Proposition 4.1 will be proved by means of the following program example $\overset{\infty}{\pi}_0$ which satisfies the conditions of the proposition.

Example 4.1

$\overset{\infty}{\pi}_0$ = program UR(OUTPUT);
var I, K : integer;
procedure $Z(J$: integer); begin WRITE($J + 1$) end;
procedure AA; begin WRITE('PROGRAM UR(OUTPUT); VA
RI, K : INTEGER; PROCEDURE Z(J : INTEGER): BEG
IN WRITE(J + 1) END; PROCEDURE AA; BEGIN
WRITE(' ') end;
procedure C; begin WRITE('PROCEDURE') end;
procedure A; begin WRITE(';BEGIN WRITE(' ') end;
procedure B; begin WRITE(' ') END;') end;
procedure AC; begin WRITE(' ' ') end;
procedure BA; begin WRITE(' A ') end;
procedure BB; begin WRITE(' B ') end;
procedure BC; begin WRITE(' C ') end;
procedure CA; begin WRITE('BEGIN K:=') end;
procedure AB; begin WRITE(';FOR I := 1 TO K DO BEGI
N WRITELN(I, I*I, I*I*I) END; AA;AA;AC;B;C;BC;A;C
;B;C;BA;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;AC;AC
;B;C;BB;BA;A;BA;B;C;BB;BB;A;BB;B;C;BB;BC;A;BC;
B;C;BC;BA;A;CA;B;C;BA;BB;A;AB;B;CA;Z(K);AB;WRI
TELN END.') end;
begin
$K := 0$;
for $I := 1$ to K do
begin WRITELN(I,I*I,I*I*I) end;
AA;AA:AC;B
C;BC; A; C; B;
C;BA; A; A;AC; B;

```
C;BB;     A;AC; B;     B;
C;BA;BC; A;AC;AC;     B;
C;BB;BA; A;     BA;     B;
C;BB;BB; A;     BB;     B;
C;BB;BC; A;     BC;     B;
C;BC;BA; A;     CA;     B;
C;BA;BB; A;     AB;     B;CA;Z(K);AB;
WRITELN
end.
```

The program $\overset{\infty}{\pi}_0$ is an infinitely reproducing program whose reproduction sequence does not contain any program more than once.

Proof The program $\overset{\infty}{\pi}_0$ corresponds essentially to the self-reproducing programs π_6 presented in Sect. 3.3.4. In order to prevent $\overset{\infty}{\pi}_0$ from being self-reproducing, we increment by 1 the max value of the internal variable I in the copy $\overset{\infty}{\pi}_1$ of $\overset{\infty}{\pi}_0$. Incrementing is done by means of procedure Z, whose source code is positioned at the program head. Thus the copy of $\overset{\infty}{\pi}_0$ is not only formally but also semantically different from $\overset{\infty}{\pi}_0$. Since the copy differs merely in an integer constant of $\overset{\infty}{\pi}_0$, it remains able to reproduce. Similarly the copy $\overset{\infty}{\pi}_2$ of $\overset{\infty}{\pi}_1$ differs from $\overset{\infty}{\pi}_1$. The internal variable I in $\overset{\infty}{\pi}_2$ is increased by 2 compared $\overset{\infty}{\pi}_0$. We thus conclude:
$\overset{\infty}{\pi}_0$ is an infinitely reproducing program. In every element $\overset{\infty}{\pi}_j$, in the reproduction sequence generated from $\overset{\infty}{\pi}_0$, the internal variable I equals j (Fig. 1). □

Remark 1. Programs $\overset{\infty}{\pi}_j$, $j \in \mathbb{N}_0$ are reproducing but not strongly reproducing. However, by deleting the instruction

```
WRITELN(I, I*I, I*I*I)
```

we obtain strongly reproducing programs. Proposition 4.1 could be tightened in this respect.

2. The programs of the reproduction sequence generated from $\overset{\infty}{\pi}_0$ contain the complete self-reproduction mechanisms of program π_6 from Sect. 3.3.4. However, we only asked for the more lenient reproductive property. In order to obtain only reproduction we have weakened the self-reproduction mechanism by adding the following piece of code:

```
for I := 1 to ... do
begin ... end
```

This method appears to be absurd at first glance. It seems, though, that the programs in the reproduction sequence $(\overset{\infty}{\pi}_i)_{i\in\mathbb{N}_0}$ need a self-reproduction mechanism to generate infinitely many pairwise different, syntactically correct programs. It would be desirable to have a reproduction sequence whose programs exhibit mechanisms weaker than self-reproduction. The difficulty in finding such a sequence may be indicative of the necessity of a strong property requirement: Infinitely many successive programs—generated one from the other—must somehow lie densely packed close together. This density seems so important that "quasi self-reproduction" mechanisms may be necessary in order to generate such sequences.
3. Programs $\overset{\infty}{\pi}_0$ only contains language constructs that are also found in the SIMULA programming language. Consequently, Proposition 4.1 extends to this programming language.
4. The first line of Proposition 4.1 has a theoretical meaning. From a practical point of view, there are indeed an infinity of reproducing programs but not the corresponding sequences. Also, when considering a finite memory size, we can only hope to represent finitely many programs only.

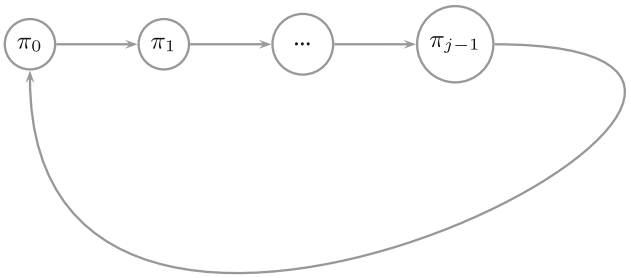


Fig. 1 Cyclically self-reproducing programs

4.2.1 Implementing $\overset{\infty}{\pi}_0$ programs

The program $\overset{\infty}{\pi}_0$ writes its outputs totally unformatted as a single line text. This line is too long not only for the printer buffer but also for the internal buffer of the PASCAL compiler. In order to get a working example for the program $\overset{\infty}{\pi}_0$, it is desirable to interpret the output from program $\overset{\infty}{\pi}_0$, namely the program $\overset{\infty}{\pi}_1$, before executing it. For that purpose, we will proceed as in Sect. 3.3.4 and thus introduce the formatting of the input by means of the procedure Q:

```
procedure Q; begin WRITELN end;
```

The relatively long text constant in procedure `AB` will be divided into four procedures. For that purpose, we will additionally define procedures `AAA`, `CB` and `CC` in the program. Appendix A.7 in ESM gives the modified program $\overset{\infty}{\pi}_0$.

4.3 Cyclically self-reproducing programs

Definition 4.4 Let π_0 be an infinitely reproducing program in the programming language S. Let $(\pi_i)_{i\in\mathbb{N}_0}$ be the reproduction sequence of π_0.

1. If there exists $j \geq 1$ such that $\pi_j = \pi_0$, then the program π_0 is said to be *cyclically reproducing*.
2. If π_0 is cyclically reproducing, the smallest integer $j \geq 1$ such that $\pi_j = \pi_0$ is called the cycle length (*period*) of $\pi_j = \pi_0$.
3. The set of all cyclically reproducing programs in S is denoted $Z(S)$.

Remark Every program π_j in the reproduction sequence of any cyclically self-reproducing program π_0 is itself a cyclically self-reproducing program which has the same cycle length as π_0.

Proposition 4.2 *For every $k \geq 1$ there exists a cyclically self-reproducing program $\overset{k}{\pi}$ with a cycle length of k for the* PASCAL *programming language.*

Proof The PASCAL program $\overset{\infty}{\pi}_0$ given in Sect. 1 is infinitely reproducing. But we can very easily deduce from $\overset{\infty}{\pi}_0$ a cyclically self-reproducing program $\overset{k}{\pi}_0$ with a cycle length of k by replacing the procedure

```
procedure Z(J : integer); begin
WRITE(J + 1) end;
```

with the procedure hereafter:

```
procedure Z(J : integer); begin
WRITE((J + 1) mod k) end;
```

The programs in the reproduction sequence generated from $\overset{\infty}{\pi}_0$ distinguish themselves directly by means of the values taken by Z. The modified procedure Z ensures that for every $k \geq 1$ we have $\overset{k}{\pi}_0 = \overset{k}{\pi}_k$.
By taking $\overset{k}{\pi}_0$ in place of $\overset{k}{\pi}$ we have proved the proposition. □

We want to give one more example of cyclically self-reproducing programs.

Example 4.2 The following program π_0^{cyc} is a variant of π_6 presented in Sect. 3.3.2, aside from some renaming. This program should only self-reproduce after a cycle of length $N = 9$ steps. This outcome is obtained by slightly changing the sequence of internal procedures in π_0^{cyc}. The resulting program π_1^{cyc} proceeds analogously . Only after nine steps do we obtain the "initial constellation" of the procedures, and thus finally the program π_0^{cyc} again.

Program π_0^{cyc} differ from program π_6 in that it extends its compatibility section:

(i) integer I,K;
procedure Z(J : integer); begin WRITE(J) end;

(ii) The division of the printing procedure of the program π_0^{cyc} into two different procedures BC and CA.

π_0^{cyc},s instruction block must be constructed such that its direct copy π_1^{cyc} differs from π_0^{cyc}. The latter must be produced again after nine steps only. The instruction block of π_0^{cyc} must be almost—but not quite—identical in all subsequent copies. Since the instruction block cannot be copied wholesale, the program division formulated in the point (ii) is justified. We specify the π_0^{cyc} algorithm and its copies with the following procedure calls:

$$\underbrace{\text{BC;}Z((}_{\text{constant}}\ \underbrace{k\ +2)\bmod 9);\ \text{CA}}_{\text{constant}}$$

Only k is varying.

```
π0^cyc = program CYCLE(OUTPUT);
         varI, K : integer;
         procedure Z(J : integer);  begin WRITE(J) end;
         procedure A;  begin WRITE('PROGRAM
         CYCLE(OUTPUT); VAR I, K : INTEGER; PROCEDURE
              Z(J : INTEGER): BEGIN WRITE(J) END; PRO
              CEDURE A; BEGIN WRITE('') end;
         procedure B;  begin WRITE('PROCEDURE') end;
         procedure C;  begin WRITE(';BEGIN WRITE(' ' ') end;
         procedure AA;  begin WRITE(' ' ') END;') end;
         procedure AB;  begin WRITE(' ' ' ') end;
         procedure AC;  begin WRITE(' A ') end;
         procedure BA;  begin WRITE(' B ') end;
         procedure BB;  begin WRITE(' C ') end;
         procedure BC;  begin WRITE('BEGIN A;A;AB;AA;K:=') end;
         procedure CA;  begin WRITE(';FOR I := 1 TO 9 DO BEGIN B;
           CASE K OF 0:BEGIN BA;C;B END;1:BEGIN BB;C;C;
           BA END;2:BEGIN AC;AC;C;AB;AA END;3:BEGIN AC;BA;C;
           AB;AB END;4:BEGIN AC;BB;C;AC END;5:BEGIN BA;
           AC;C;BA END;6:BEGIN BA;BA;C;BB END;7:BE GIN BA;
           BB;C;BC END;8:BEGIN BB;AC;C;CA END END;AA;K:=
           (K+1) MOD 9 END;BC;Z((K + 1) MOD 9);
         CA;WRITELN END.')end;
         begin
         A; A; AB; AA;

K := 1;
for I := 1 to  9 do
begin B;
    case K of
0 : begin BA;C;B end;
1 : begin BB;C;C;AB end;
2 : begin AC;AC;C;AB;AA end;
3 : begin AC;BA;C;AB;AB end;
4 : begin AC;BB;C;AC end;
5 : begin BA;AC;C;BA end;
6 : begin BA;BA;C;BB end;
7 : begin BA;BB;C;BC end;
8 : begin BB;AC;C;CA end
```

Table 34 Table of values of the variables controlling the cyclic self-reproduction. The first column contains the start value of K in π_i^{cyc}, the second column (*) describes the spanning sequence of K values while the last column (**) contains the start value of K in π_i^{cyc} to produce π_{i+1}^{cyc}.

*	$I=1$	$I=2$	$I=3$	$I=4$	$I=5$	$I=6$	$I=7$	$I=8$	$I=9$	**
π_0^{cyc} $k=1$	2	3	4	5	6	7	8	0	1	2
π_1^{cyc} $k=2$	3	4	5	6	7	8	0	1	2	3
π_2^{cyc} $k=3$	4	5	6	7	8	0	1	2	3	4
π_3^{cyc} $k=4$	5	6	7	8	0	1	2	3	4	5
π_4^{cyc} $k=5$	6	7	8	0	1	2	3	4	5	6
π_5^{cyc} $k=6$	7	8	0	1	2	3	4	5	6	7
π_6^{cyc} $k=7$	8	0	1	2	3	4	5	6	7	8
π_7^{cyc} $k=8$	0	1	2	3	4	5	6	7	8	0
π_8^{cyc} $k=0$	1	2	3	4	5	6	7	8	0	1

```
  end;
  AA;
  K := (K + 1) mod 9
end;
BC;Z((K + 1) mod 9);CA;
WRITELN
end.
```

Verification

Programs $\pi_0^{\text{cyc}}, \pi_1^{\text{cyc}}, \ldots, \pi_8^{\text{cyc}}$ differ only in the call order of procedures $B, \ldots, CA$ and the start value K. The output sequence of these procedures is controlled through the variable K. Let us view Table 34: Each row of this Table lists a value from 0 to 8 in the division "$I = 1$" to "$I = 8$" exactly once. Due to the case instruction, every π_i^{cyc} produces all procedures B til CA. The last column of the table list a different the start value of K in every copy π_i^{cyc}, $i = 0, \ldots, 8$. Similarly, K's initial value in the last column of π_8^{cyc} is the same π_0^{cyc}'s initial K value. Since the instruction block of π_i^{cyc} differs only by the initial value of K, we have:

$$\pi_9^{\text{cyc}} = \pi_0^{\text{cyc}}.$$

Remark I. Essentially, point II of Remark 4.2 for infinitely reproducing programs $\overset{\infty}{\pi}_0$ still holds in the case of cyclic self-reproducing programs π_0^{cyc}. In any event, however, our examples of infinitely reproducing programs and cyclic self-reproducing programs have not simplified the self-reproduction mechanism of the program π_6.

II. Proposition 4.2 obviously holds for SIMULA programs.

Example 4.3 As an example of cyclic self-reproducing programs in SIMULA programming language, a SIMULA version of the program π_0^{cyc} follows.

```
begin
integer I, K;
procedure Z(J); integer J; OUTINT(J, 1);
procedure A; OUTTEXT("BEGIN INTEGER I, K; PROCEDURE
    Z(J); INTEGER J; OUTINT(J, 1); PROCEDURE A; OUTTE
    XT(" ");
procedure B; OUTTEXT("PROCEDURE");
procedure C; OUTTEXT(";OUTTEXT(" ");
procedure AA; OUTTEXT(" ");");
procedure AB; OUTTEXT(" ");
procedure AC; OUTTEXT(" A ");
procedure BA; OUTTEXT(" B ");
procedure BB; OUTTEXT(" C ");
procedure BC; OUTTEXT(" A; A; AB; AA; K :=");
procedure CA; CA; OUTTEXT("; FOR I:= 1 STEP 1 UNTIL 9 DO
    BEGIN B; IF K = 0 THEN BEGIN BA; C; B END ELSE IF K = 1
    THEN BEGIN BB; C; C; AB END ELSE IF K = 2 THEN BEGIN
    AC; AC; C; AB; AA END ELSE IF K = 3 THEN BEGIN AC; BA; C
    AB; AB END ELSE IF K = 4 THEN BEGIN AC; BB; C; AC END
    ELSE IF K = 5 THEN BEGIN BA; AC; C; BA END ELSE IF K =
    6 THEN BEGIN BA; BB; C; BC END ELSE IF K = 7 THEN BEG
    IN BA; BB; C; BC END ELSE BEGIN BB; AC; C; CA END AA;K
    := (K + 1) MOD 9; END; BC; Z((K + 2) MOD 9); CA END;");
A; A; AB; AA;
K := 1;
for I := 1 step 1 until 9 do
begin B;
     if K = 0 then begin BA; C; B end
else if K = 1 then begin BB; C; C; AB end
else if K = 2 then begin AC; AC; C; AB; AA end
else if K = 3 then begin AC; BA; C; AB; AB end
else if K = 4 then begin AC; BB; C; AC end
else if K = 5 then begin BA; AC; C; BA end
else if K = 6 then begin BA; BA; C; BB end
else if K = 7 then begin BA; BB; C; BC end
          then begin BB; AC; C; CA end;
AA;
K := (K + 1) mod 9;
end;
BC; Z((K + 1) mod 9); CA
end;
```

Verification

The verification of this SIMULA program is the same as for the PASCAL program π_0^{cyc}.

4.3.1 Implementing the program $\overset{k}{\pi}_0$

The same remarks made in Sect. 4.2.1 with respect to the implementation of $\overset{\infty}{\pi}_0$ hold for the implementation of $\overset{k}{\pi}_0$, as well. Appendix A.8 in ESM provides more details.

4.3.2 Implementing the program π_0^{cyc}

Similarly, the remarks of Sect. 4.2.1 hold for the implementation of π_0^{cyc}. The text constant which represents the algorithm of π_0^{cyc}, though, has to be divided into even more procedures than was the case for $\overset{\infty}{\pi}_0$, for formatting reasons. More details are provided in Appendix A.9 in ESM.

4.4 Cyclic self-reproduction with programming language change

We introduce cyclic self-reproduction as a special case of infinite reproduction in Sect. 4.3. Infinitely reproducing programs are themselves reproducing programs. According to Definition 4.1, a reproducing program π produces a program π'. Moreover, programs π and π' are written in the same programming language S. We could have also defined reproducing programs in a different way by allowing program π' to be expressed in a different programming languages $S' \neq S$. Such a definition would have been broader and more general than Definition 4.1. Corresponding definitions of infinitely reproducing and cyclic self-reproducing programs would have consequently been more general as well. We will show by means of our next example that such a generalization is eminently reasonable. Example 4.4 introduces a program which not only produces another program in a different programming language but also cyclically reproduces itself.

Example 4.4 We consider the SIMULA program π_4 from Sect. 3.2.6 and the PASCAL program π_6 from Sect. 3.3.2. Both programs are nearly identical, since each effectively represents the translation of the other. We will combine a PASCAL program π_{PAS} and a SIMULA program π_{SIM} such that π_{PAS} generates π_{SIM} and conversely π_{SIM} outputs π_{PAS}. Both programs π_{PAS} and π_{SIM} are cyclic self-reproducing programs changing their programming language.

```
π_PAS = program X(OUTPUT);
        var T, F : boolean;
        procedure A(Z : boolean);  begin if Z
            then WRITE('BEGIN BOOLEAN T, F;')
            else WRITE('PROGRAM X(OUTPUT); VAR T, F: BOOLEAN;') end;
        procedure B(Z : boolean);  begin if Z
            then WRITE('PROCEDURE')
            else WRITE('PROCEDURE') end;
        procedure C(Z : boolean);  begin if Z
            then WRITE('(Z); BOOLEAN Z; IF Z THEN OUTTEXT("')
            else WRITE('(Z: BOOLEAN); BEGIN IF Z THEN WRITE(' ') end;
        procedure AA(Z : boolean);  begin if Z
            then WRITE(' ") ELSE OUTTEXT(" ')
            else WRITE(' ') end;
    procedure AB(Z : boolean);  begin if Z
        then WRITE(' ");')
        else WRITE(' ') END; ') end;
    procedure CB(Z : boolean);  begin if Z
        then WRITE(' " ')
        else WRITE(' ' ') end;
    procedure BA(Z : boolean);  begin if Z
        then WRITE(' A ')
        else WRITE(' A ') end;
    procedure BB(Z : boolean);  begin if Z
        then WRITE(' B ')
        else WRITE(' B ') end;
    procedure BC(Z : boolean);  begin if Z
        then WRITE(' C ')
        else WRITE(' C ') end;
    procedure AC(Z : boolean);  begin if Z
        then WRITE('T := TRUE; F:= FALSE; ⊗ END')
        else WRITE('BEGIN T := TRUE; F := FALSE; ⊗ WRITELN END.') end;
```

Where the program part denoted by ⊗ is:

```
begin   T := true;  F := false;
A(T);
B(T);   BA(T);              C(T);  A(F);   AA(T);          A(T);          AB(T);
B(T);   BB(T);              C(T);  B(T);   AA(T);          B(T);          AB(T);
B(T);   BC(T);              C(T);  C(F);   AA(T);          C(T);  CB(T);  AB(T);
B(T);   BA(T);  BA(T);      C(T);  AA(F);  AA(T);  CB(T);  AA(T);  CB(T);  AB(T);
B(T);   BA(T);  BB(T);      C(T);  AB(F);  AA(T);  CB(T);  AB(T);          AB(T);
B(T);   BC(T);  BB(T);      C(T);  CB(F);  AA(T);          CB(T);  CB(T);  AB(T);
B(T);   BB(T);  AB(T);      C(T);  BA(T);  AA(T);          BA(T);          AB(T);
B(T);   BB(T);  BB(T);      C(T);  BB(T);  AA(T);          BB(T);          AB(T);
B(T);   BB(T);  BC(T);      C(T);  BC(T);  AA(T);          BC(T);          AB(T);
B(T);   BA(T);  BC(T);      C(T);  AC(F);  AA(T);          AC(T);          AB(T);
AC(T);
;WRITELN
end.
```

The program π_{PAS} outputs the following SIMULA program π_{SIM}:

```
π_SIM = begin
        boolean T, F;
        procedure A(Z); boolean  Z; if Zthen
            OUTTEXT("PROGRAM X(OUTPUT); VAR T, F: BOOLEAN;")
            else OUTTEXT("BEGIN BOOLEAN T, F;");
        procedure B(Z); boolean  Z; if Z
            then OUTTEXT("PROCEDURE")
            else OUTTEXT("PROCEDURE")
        procedure C(Z); boolean  Z; if Z
            then OUTTEXT("(Z: BOOLEAN); BEGIN IF Z THEN WRITE(' ")
            else OUTTEXT("(Z); BOOLEAN Z; IF Z THEN OUTTEXT(" ");
        procedure AA(Z); boolean  Z; if Z
            then OUTTEXT(" ') ELSE WRITE(' ")
            else OUTTEXT(" ") ELSE OUTTEXT(" ");
        procedure AB(Z); boolean  Z; if Z
            then OUTTEXT(" ') END;")
            else OUTTEXT(" "); ");
        procedure CB(Z); boolean  Z; if Z
            then OUTTEXT(" ' ")
            else OUTTEXT(" " ");
        procedure BA(Z); boolean  Z; if Z
            then OUTTEXT(" A ")
```

```
    else OUTTEXT(" A ");
procedure BB(Z); boolean Z; if Z
    then OUTTEXT(" B ")
    else OUTTEXT(" B ");
procedure BC(Z); boolean Z; if Z
    then OUTTEXT(" C ")
    else OUTTEXT(" C ");
procedure AC(Z); boolean Z; if Z
    then OUTTEXT("BEGIN T := TRUE; F := FALSE; ⊗;
    WRITELN END.")
    else OUTTEXT("T := TRUE; F := FALSE; ⊗ END");
T := true; F := false; ⊗
end
```

Verification

Programs π_{PAS} and π_{SIM} contain respectively every string part – the division of both programs – in the procedures A til AC. Since there is a one to one mapping between the different parts of the code division for both programs, we can alternatively replace these strings in the procedures. Every procedure of π_{PAS} has thus the following general structure:

```
procedure < name > (Z : boolean);
begin if Z then     WRITE('< string s from π_SIM >')
          else      WRITE('< string s' from π_PAS
                    corresponding to the string s>')
end;
```

Some string parts of π_{PAS} are identical to their counterpart in π_{SIM}. The procedures which manage those string parts obviously contain some redundancy as shown in the following example:

```
procedure BA(Z : boolean);
begin if Z then          WRITE('A')
          else           WRITE('A')
end;
```

This redundancy is acceptable since it provides a unified procedure structure. The parameter passed to the procedure call decides on the choice of alternatives. The PASCAL program contains the SIMULA string parts always into the "then" branch of the procedure while the PASCAL string parts are always located into the "else" branch and conversely in the SIMULA programs. We thus achieve the following: A procedure within a PASCAL program called with the value true can be called with the value true in the SIMULA program, as well. From that, it follows that the instruction part in π_{SIM} and in π_{PAS} are essentially the same. From the aforementioned reasoning and the fact that π_{SIM} and π_{PAS} differ only in their respective procedures in π_4 and π_6, it follows that π_{PAS} reproduces π_{SIM} and conversely.

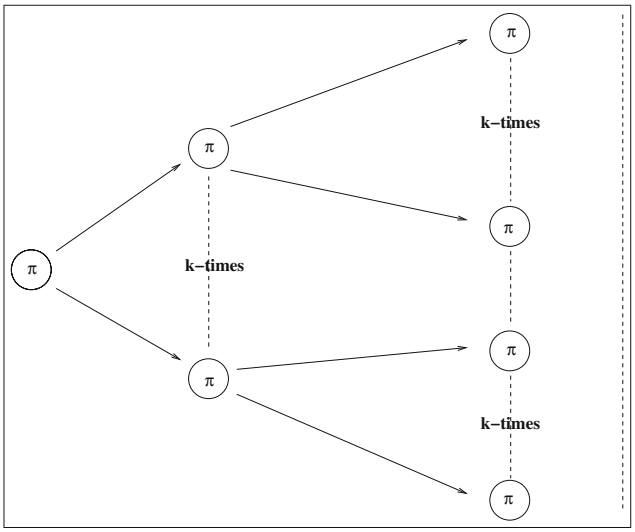


Fig. 2 K-times self-reproduction

4.5 K-times self-reproducing programs

We described program reproduction as a weaker form of self-reproduction in Sect. 4.2. At this point, we would like to introduce the concept of k-times self-reproduction as a strengthening of the simple self-reproduction concept.

Definition 4.5 Let $k > 1$ and π be a program in S which is syntactically correct.

1. a) If π does not consider any input, then π is said k-times self-reproducing, if π precisely outputs its program source in S, k times.
   b) If π considers inputs, then π is said k-times self-reproducing, if π precisely outputs its program source in S, for every valid input value.
2. $SR^k(S)$ describes the set of all k-times self-reproducing programs in S.

The existence of k-times self-reproducing programs follows directly from Corollary 2.4 (Fig. 2).

Proposition 4.3 *There is a k-times self-reproducing program $\pi(k)$ for the programming language* PASCAL*, for every $k > 1$.*

We now give for every $k > 1$ an example of k-times self-reproducing program. This example represents a proof of Proposition 4.3.

Example 4.5

```
π(k) = program PIK(OUTPUT);
       var I : integer;
       procedure AA; begin WRITE('PROGRAM PIK(OUTPUT); VAR I:
           INTEGER;PROCEDURE AA;BEGIN WRITE(' ') end;
       procedure C; begin WRITE('PROCEDURE') end;
       procedure A; begin WRITE(';BEGIN WRITE(' ') end;
       procedure B; begin WRITE( ' ' ) END;') end;
```

```
procedure AC; begin WRITE(' ' ') end;
procedure BA; begin WRITE(' A ') end;
procedure BB; begin WRITE(' B ') end;
procedure BC; begin WRITE(' C ') end;
procedure AB; begin WRITE('BEGIN FOR I := 1 TO 5 DO BEGIN AA;A
    A;AC;B;C;BC;A;C;B;C;BA;A;A;AC;B;C;BB;A;AC;B;B;C;BA;BC;A;
    AC;AC;B;C;BB;BA;A;BA;B;C;BB;BB;A;BB;B;C;BB;BC;A;BC;B;C;B
    A;BB;A;AB;B;AB;WRITELN END END.') end;
begin
for I := 1 to 5 do
begin AA;          AA; AC; B;
      C; BC;   A;   C;      B;
      C; BA;   A;   A;  AC; B;
      C; BB;   A; AC; B;    B;
      C; BA; BC; A; AC; AC;   B;
      C; BB; BA; A;     BA;   B;
      C; BB; BB; A;     BB;   B;
      C; BB; BC; A;     BC;   B;
      C; BA; BB; A;     AB;   B;AB;WRITELN
end
end.
```

Verification

The verification of $\pi(k)$ follows directly from the verification of the program π_6 in Sect. 3.3.2. The difference between $\pi(k)$ and π_6 is essentially the for loop:

```
for   I := 1 to k do
begin ...          end;
```

which is located in the output algorithm. The output algorithm of π_6 will be output k-times in $\pi(k)$ as well. Hence $\pi(k)$ is a k-times reproducing program.

Remark I. Every k-times self-reproducing program π is of course self-reproducing but not strongly self-reproducing. Moreover, a k-times self-reproducing program is cyclically self-reproducing with a cycle length of 1.

II. Proposition 4.3 still holds for SIMULA programs. As a proof, we translate the program $\pi(k)$ into the corresponding SIMULA program. This is done without any difficulty since $\pi(k)$ does not contain any PASCAL-specific structure.

4.5.1 Implementing $\pi(k)$

As to the implementation of the program $\pi(k)$, we refer to the remarks made in Sect. 3.3.3. The Appendix A.10 in ESM implements $\pi(k)$ with $k = 5$.

4.6 Hierarchy of self-reproduction

In the previous sections, we have defined the following sets with respect to a fixed programming language S:

$R(S)$, $U(S)$, $Z(S)$ and $SR^k(S)$.

For this sets, we have

$$SR^k(S) \subset SR(S) \subset Z(S) \subset U(S) \subset R(S), \tag{7}$$

where $SR(S)$ denotes the set of self-reproducing programs in S. Generally speaking, we are dealing with proper inclusions

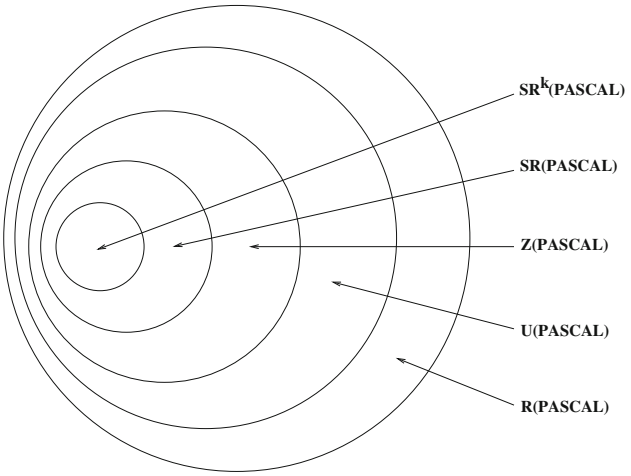


Fig. 3 Reproduction hierarchy of S

as shown by means of the PASCAL program example given before. To be more precise, we have:

- $\pi_{\mathrm{PAS}(k)}$ given in the proof of Lemma 4.1 is reproducing but not infinitely reproducing.
- $\overset{\infty}{\pi}_0$ given Sect. 4.2 is infinitely reproducing but not cyclically self-reproducing.
- π_0^{cyc} given in Sect. 4.3 is cyclically self-reproducing but not self-reproducing.
- π_6 from Sect. 3.3.2 is self-reproducing but not k-self-reproducing for a $k > 1$.

We thus conclude:

$$SR^k(\text{PASCAL}) \subsetneq SR(\text{PASCAL}) \subsetneq Z(\text{PASCAL}) \subsetneq U(\text{PASCAL}) \subsetneq R(\text{PASCAL}).$$

Figure 3 graphically summarizes this result.

Definition 4.6 For every programming language S the sequence of inclusions given in Eq. 7 is called the *reproduction hierarchy* of S.

5 Additional properties of self-reproducing programs

5.1 Introduction

In Chap. 3, we presented some examples of self-reproducing programs. These programs have in common that beyond outputting their own source code, they do not execute any other function. Consequently, we may consider the following interesting questions:

1. "*Are there programs written in programming language S which do more than simply self-reproduce?*" Concretely,

we may ask: "*Do self-reproducing programs exist in S which can additionally perform search functions, or factor integers into prime numbers or manage a database, or more?*"

2. Let us suppose that for the given programming language S, we can answer question 1 in the affirmative. We may then generalize thusly: "*Does a self-reproducing program $\tilde{\pi}$ exist in S which, for a program $\tilde{\pi}$ written in S, computes the same function as π?*" Should the last question be answered in the affirmative, then intuitively, the complexity and scope of a self-reproducing program $\tilde{\pi}$ (which computes a given function) must be greater than that of a non self-reproducing program, which computes the same function.
3. It follows from the last observation that in order to find program $\tilde{\pi}$, it would perhaps be simpler to first develop a non self-reproducing program π and then transform it into a self-reproducing version $\tilde{\pi}$. In this context, the following question arises: "*Does an algorithm in a given programming language S exist which, for every program π in S, yields a self-reproducing program $\tilde{\pi}$ in S which computes the same function as π?*"

In order to answer questions 1 to 3, we first have to define what precisely we mean by "a function computed by a program π in S". Alas, because of the diversity of commonly used programming languages which operate on different data types and compute different results on different computing devices, it is quite impossible to specify a formally exact and universally applicable definition. Nevertheless, for our purposes, we will use the definitions and properties given hereafter.

Programs written in common programming languages, executing on real computing devices, generally work on data provided by several input files and produce results stored in several output files. These files contain strings—symbol sequences—which are interpreted by a program π in S as integers, real numbers, text, and so on. Naturally, successful interpretation of these symbols by π can only be effected if they are part of a valid, finite alphabet A_S. The content of a file may be viewed as a word in A_S^*. This point of view is reflected in the following definition, which in addition allows program π the freedom to use at runtime certain files both as in- and output files.

Definition 5.1 Let π be a program in S with $p \geq 0$ in- and output files, of which q files are used as output files (with $0 \leq q \leq p$). The function f_π computed by π is a partial function from $(A_S^p)^*$ into $(A_S^q)^*$ which, for every assignment of words in $(A_S)^*$ to p (input, output) files, gives exactly one assignment of words in $(A_S)^*$ to the q output files.

Example 5.1 Let us consider the following PASCAL program.

```
π0 = program X (INPUT,OUTPUT);
     var I : integer;
         Y : real;
     begin
     for I:= 1 to 10 do
     begin READ(Y); WRITELN(SQRT(Y)) end
     end.
```

Program π_0 reads ten real numbers from input file (in INPUT) and outputs their respective square root into OUTPUT. Let A_{PAS} be the set of all symbols which a PASCAL program is able to process. From Definition 5.1, we have

$$f_{\pi_0} : A_{\text{PAS}}^* \longrightarrow A_{\text{PAS}}^*$$
$$x \longmapsto f_{\pi_0}(x), x \in A_{\text{PAS}}^*$$

If the start sequence of x cannot be interpreted as a sequence of ten real numbers, then the value $f_{\pi_0}(x)$ is undefined. Otherwise, $f_{\pi_0}(x)$ is a word in A_{PAS}^*, whose beginning can be interpreted as a sequence of ten real numbers, as well. The latter sequence list the square roots of the ten input values in x.

Exactly describing $f_{\pi_0}(x)$ would require stating the explicit conversion function from $\mathbb{R}$ to A_{PAS}^* and conversely from A_{PAS}^* to $\mathbb{R}$.

Remark 5.1 1. Definition 5.1 is obviously not formally exact, but defined 'pragmatically'.
2. Definition 5.1 in essence corresponds to the definition of functions computed by PL(A) programs, given in Sect. 2.4. When p and/or q are equal to zero, we proceed according to the remark succeeding Definition 2.8.

It follows from Definition 5.1 that a self-reproducing program $\tilde{\pi}$ in S which without any additional input data computes the "same" function as a different, non self-reproducing program π in S must indeed be computing a totally different function since the output data produced by π and by $\tilde{\pi}$ are different. In order to sidestep this confusing nomenclature, let us consider Definition 5.2, while noting that we can interpret every function $F : M^n \to M^n$ with $m, n \in \mathbb{N}^*$ as a m-tuple $F = (F_1, \ldots, F_m)$ of functions $F_i : M^n \to M, i = 1, \ldots, m$, for any set M and thus we have $F(x) = (F_1(x), \ldots, F_m(x))$ for any $x \in M^n$.

Definition 5.2 Let π be a program in S. A program $\tilde{\pi}$ in S is called a self-reproducing version of π if for the functions computed by π and $\tilde{\pi}$ respectively and given by

$$f_\pi : (A_S^*)^{p_1} \to (A_S^*)^{q_1} \quad \text{and} \quad f_{\tilde{\pi}} : (A_S^*)^{p_2} \to (A_S^*)^{q_2},$$

we have

1. either $p_1 = p_2$ and $q_1 = q_2$ and there exists exactly one $j \in \{1, \ldots, q_2\}$ with

$$(f_{\tilde{\pi}})_i(\bar{x}) = (f_\pi)_i(\bar{x}) \quad \text{for } i \neq j,$$
$$(f_{\tilde{\pi}})_j(\bar{x}) = (f_\pi)_j(\bar{x}) \circ \alpha \circ \tilde{\pi} \circ \beta$$

where $\bar{x} \in (A_S^*)^{p_1}$ and $\alpha, \beta \in A_S^*$;

2. or $p_1 = p_2$ and $q_2 = q_1 + 1$ and

$$(f_{\tilde{\pi}})_i(\bar{x}) = (f_\pi)_j(\bar{x}) \quad \text{for } i \in \{1, \ldots, q_1\},$$
$$(f_{\tilde{\pi}})_{q_2}(\bar{x}) = \alpha \circ \tilde{\pi} \circ \beta,$$

where $\bar{x} \in (A_S^*)^{p_1}$ and $\alpha, \beta \in A_S^*$.

The symbol $\circ$ describes the concatenation of "words", here from A_S^*.

Definition 5.2 ensures that $\tilde{\pi}$ outputs its own source code independently of its input. Program $\tilde{\pi}$ allocates its code either to an output word outputted by π, as well (case 1), or to an additional word (case 2).

Remark 5.2 The self-reproducing version $\tilde{\pi}$ of a program π in S is generally not unique.

With the help of Definition 5.2 we now can formulate questions 2 and 3 in a more precise fashion:

1. "*For every program π written in a given programming language S, does a self-reproducing version exist?*"
2. "*For a given programming language S, does an algorithm exist which, for every program in S, produces a self-reproducing version $\tilde{\pi}$ of π?*"

We will proceed to explicitly answer questions 1 to 3 for the SIMULA and PASCAL programming languages in the sections to come.

5.2 Self-reproducing principles with respect to the PASCAL programming language

As far as the PASCAL programming language is concerned, question 1 in Sect. 5.1 can be answered by the following example.

Example 5.2 We specify a self-reproducing version $\tilde{\pi_0}$ of the program π_0 given in Example 5.1. We shall use as a starting point program π_6 given in Sect. 3.3.3 and then try to combine programs π_6 and π_0 into a self-reproducing version $\tilde{\pi_0}$. To this end, let us briefly recall what program π_6 exactly does. Program π_6 contains its own source code inside procedures $A, \ldots, AC$ in the form of partial strings. Of course, strings that appears several times are stored only once. Concatenation of these partial strings yields the code of π_6. The first partial string s_1 of π_6 contains the program head until the first symbol $'$. The last partial string in procedure AB contains the partial string s_9 which stores the entire code instruction block of π_6. Let us summarizes all this by means of the following code:

```
π6 = program PI6(OUTPUT);
     procedure AA;  begin WRITE('      s1 '')end;

     procedure C;   begin.....          end;
                    ⋮                   ⋮
     procedure BC;  begin.....          end;
     procedure AB;  begin WRITE(' s9 ') end;
```

where the two strings s_1 and s_9 are given by:

```
s1 = program PI6(OUTPUT);
     procedure AA; begin WRITE('
```

and

```
     begin
     AA; AA; AC; B;
     .....
s9 = AB;WRITELN
     end
     end.
```

respectively.

We insert the π_0 program header

```
programX (INPUT,OUTPUT);
var I : integer;
    Y : real;
```

into the string s_1, while we insert the code instruction block of π_0

```
for I:= 1 to 10 do
begin READ(Y); WRITELN(SQRT(Y)) end;
```

into the string s_9. We thus obtain the resulting partial strings s_1' and s_9' respectively, given by

```
s1' = program PI6X (INPUT,OUTPUT);
      var I : integer; Y : real; procedure AA;
      begin WRITE('
```

and

```
s9' = begin
      for I:= 1 to 10 do
      begin READ(Y); WRITELN(SQRT(Y)) end;
      AA; AA; AC; B; …AB; WRITELN
      end end.
```

It is obvious that replacing strings s_1 and s_9 with strings s_1' and s_9', respectively, in program π_6 gives again a syntactically correct, self-reproducing program $\tilde{\pi_0}$. Program $\tilde{\pi_0}$ first computes the for loop of π_0 and then self-reproduces. We

thus have for the function $f_{\tilde{\pi_0}} : A^*_{\text{PAS}} \rightarrow A^*_{\text{PAS}}$, computed by $\tilde{\pi_0}$:

$$f_{\tilde{\pi_0}}(x) = f_{\pi_0}(x) \circ \pi_0,$$

for every $x \in A^*_{\text{PAS}}$. By using case 1 in Definition 5.2, we conclude that $\tilde{\pi_0}$ is a self-reproducing version of π_0. Appendix A.11 in ESM gives an implementation of Program π'_6, which has been derived from the implementation of Program π_6 itself.

The construction of $\tilde{\pi_0}$ from Programs π_0 and π_6 exhibits no particular aspects that depend on any special properties of π_0. This construction ought to be generalizable to any given PASCAL-program. In order to facilitate this generalization, we first need to adopt two additional conventions:

A context-free grammar G_{PAS} (as far as that is possible, refer to Sect. 2.3) is given for the PASCAL programming language in [10]. Our deliberation will be guided by that grammar.

Convention 5.1 *Let ν be a non terminal symbol in G_{PAS} and π a valid* PASCAL *program. We denote by $\nu\pi$ the partial string of the source code π which can be derived from the non terminal symbol ν. Should the symbol ν not be contained in the derivation tree of π, we set the string $\nu\pi$ to the empty word.*

The partial string $\nu\pi$ is of course dependent on the position of ν in the derivation tree of π. However, for our purposes, this is of minor importance. With this formalism, we may now derive equations from the productions (rules) of the grammar G_{PAS}.

Example 5.3 G_{PAS} contains the production given by

$$< \text{program} > ::= < \text{program heading} >< \text{block} > .$$

For every valid PASCAL program π the following equation holds true

$$\begin{aligned}\pi &= < \text{program} > \pi \\ &= < \text{program heading} > \pi \ < \text{block} > \pi\end{aligned}$$

Combining programs π_6 and π_0 into program $\tilde{\pi_0}$ did not result in any conflict. Putting it differently, all procedures names of program π_6 differed from the variable names in π_0 though in general, this cannot be supposed. In order to simplify testing a PASCAL program π with respect to this point (i.e guessing whether some labels are identical to a procedure name in π_6 nor not), we will standardize the procedure names in π_6 by deciding to use only the capital letter 'A' to label every of them. All π_6 procedures are hence elements of the set[7] $\{A\}^+$ and differ only in their respective length. We will see later on that it is necessary for some programs to generate new procedures. The name of those latter procedures will be defined over $\{A\}^+$, as well. Through this standardization of the procedure names, it is obvious that some names are likely to be very long. Thus, in order to spare our pen, we will consider a second convention.

Convention 5.2 1. *Let $\underbrace{A \ldots A}_{k \text{ times}}$ an element in $\{A\}^+$. We will then note it shortly with A^k.*
2. *We abbreviate the r successive call of functions A^j, usually denoted $\underbrace{A^j; \ldots ; A^j;}_{r \text{ times}}$, by using the notation $(A^j)^r$.*

With the help of Convention 5.2, we can thus rename procedure names in program π_6 as follows:

$\pi_6 =$ program PI6(OUTPUT);
procedure A; beginWRITE('PROGRAM PI6(OUTPUT);PROCEDURE A;
BEG GIN WRITE(' ') end;
procedureA^2; begin WRITE(' PROCEDURE ') end;
procedureA^3; begin WRITE(' ; BEGIN WRITE(' ') end;
procedureA^4; begin WRITE(' ') END; ') end;
procedureA^5; begin WRITE(' ' ') end;
procedureA^6; begin WRITE(' A ') end;
procedureA^7; begin WRITE(' BEGIN A; A; A^5; A^4; A^2; $(A^6;)^2A^3$; A^2; A^4 ; $A^2(A^6;)^3A^3$; A^3; A^5; A^4; A^2; $(A^6;)^4A^3$; A^5; A^4; A^4; A^2; $(A^6;)^5A^3$; A^5 ; A^5; A^4; A^2; $(A^6;)^6A^3$; A^6; A^4; A^2; $(A^6;)^7A^3$; A^7; A^4; A^7; WRITELN E ND.') end;
begin
A; A; A^5; A^4;
A^2; $(A^6;)^2A^3$; A^2; A^4;
$A^2(A^6;)^3A^3$; A^3; A^5; A^4;
A^2; $(A^6;)^4A^3$; A^5; A^4; A^4;
A^2; $(A^6;)^5A^3$; A^5; A^5; A^4;
A^2; $(A^6;)^6A^3$; A^6; A^4;
A^2; $(A^6;)^7A^3$; A^7; A^4; A^7;
WRITELN
end.

After these preliminary remarks we now are able to establish and prove the principles of self-reproduction with respect to PASCAL programs.

Proposition 5.1 (*Self-reproduction with respect to* PASCAL *programs*) *For every syntactically valid* PASCAL *program π, there exists a self-reproducing version $\tilde{\pi}$ of π.*

Proof The proof is divided into two parts A and B. In Part A, we will first present a construction of a program $\tilde{\pi}$ with respect to an arbitrary program π. In Part B, we will then show that $\tilde{\pi}$ is indeed a self-reproducing version of π. Moreover, the proof assumes the grammar G_{PAS}.

Let us now consider any valid PASCAL program π. If it contains labels from $\{A\}^+$, we replace them with labels that are not elements of $\{A\}^+$. Thus, we obtain a new program π' which is formally different from π. If π does not contain labels from $\{A\}^+$, we state that $\pi' = \pi$.

[7] $\{A\}^+$ denotes the set all words composed of a finite number of times of the letter A.

Part A:

From the properties of G_{PAS}, the program π' has the following structure:

$$\pi' = < \text{program heading} > \pi' < \text{block} > \pi',$$

where $< \text{program heading} > \pi'$ and $< \text{block} > \pi'$ are different from the empty word. As far as programs π_6 and $\tilde{\pi}$ are concerned, we thus have:

$$\pi_6 = < \text{program heading} > \pi_6 < \text{block} > \pi_6$$
$$\tilde{\pi} = < \text{program heading} > \tilde{\pi} < \text{block} > \tilde{\pi}.$$

We obtain the program $\tilde{\pi}$ by combining $<$ program heading $> \tilde{\pi}$ (given by both $<$ program heading $> \pi_6$ and $<$ program heading $> \pi'$) and $<$ block $> \tilde{\pi}$ (given by both $<$ block $> \pi_6$ and $<$ block $> \pi'$).

a) Combining $<$ program heading $> \tilde{\pi}$.
It follows from the definition of G_{PAS}, that we have the general relation for $<$ program heading $> \pi'$

$< \text{program heading} > \pi' = \text{program}\mu(\mu_1, \ldots, \mu_r);$

where $\mu_1, \ldots, \mu_r$ with $r \geq 0$, are valid PASCAL labels. The program name itself is represented by the word μ whereas the words μ_i describes the files used by the program π'. As for the standard OUTPUT file, without loss of generality we set $\mu_r = \text{OUTPUT}$.
For program π_6, we have:

$< \text{program heading} > \pi_6 = \text{program PI6(OUTPUT);}$

We then combine

$< \text{program heading} > \tilde{\pi} = \text{program P}(\mu_1, \ldots, \mu_k, \text{OUTPUT});$

with

$$k = \begin{cases} r-1 & \text{if } u_r = \text{OUTPUT}, \\ r & \text{otherwise.} \end{cases}$$

b) Combining $<$ block $> \tilde{\pi}$.
It follows from the definition of G_{PAS} that

$$\begin{aligned} < \text{block} > \pi' &= < \text{label declaration part} > \pi' \\ &= < \text{constant declaration part} > \pi' \\ &= < \text{type definition part} > \pi' \\ &= < \text{variable declaration part} > \pi' \\ &= < \text{procedure and function} \\ &\quad \text{declaration part} > \pi' \\ &= < \text{statement part} > \pi' \end{aligned}$$

All strings until $<$ statement part $> \pi'$ may be empty. Programs π' and $\tilde{\pi}$ have a similar structure. Since strings $<$ label declaration part $> \pi_6, \ldots,$ $<$ variable declaration part $> \pi_6$ are equal to the empty word, we thus write:

$< \text{label declaration part} > \tilde{\pi} := < \text{label declaration part} > \pi'$
$< \text{constant declaration part} > \tilde{\pi} := < \text{constant declaration part} > \pi'$
$< \text{type definition part} > \tilde{\pi} := < \text{type definition part} > \pi'$
$< \text{variable declaration part} > \tilde{\pi} := < \text{variable declaration part} > \pi'$

We observed in Sect. 3.2.5 that it was impossible to print SIMULA code en bloc which contained the symbol ". We have to face up to the same difficulty with the symbol ' in PASCAL. When the program π' contains one or more symbols ' the source code of π' must be divided accordingly. We thus set:

$$\begin{aligned} S &= < \text{label declaration part} > \pi' \\ &= < \text{constant declaration part} > \pi' \\ &= < \text{type definition part} > \pi' \\ &= < \text{variable declaration part} > \pi' \end{aligned}$$

and

$T = (< \text{statement part} > \pi' \text{without the external labels}$ begin and end.)

By using strings S and T, we can then represent the π' source code as follows:

$$\pi' = < \text{program heading} > \pi' \circ \text{begin} \circ T \text{ end}.$$

First case: S is different from the empty word (in other words π has a non empty declaration part). The division of S into a series of $n \geq 1$ partial strings s_i gives:

(i) $s_i = \text{`` or } s_i \neq \text{''}, \quad i \in [n]$
(ii) $s_i \neq \text{``} \Rightarrow s_{i+1} = \text{''}, \quad i \in [n-1]$
(iii) $s_1 \circ s_2 \circ \ldots \circ s_n = S$

Let p be the number of partial strings of S not equal to '. We have $1 \leq p < n$. For every partial string different from ', except for s_1, we generate a procedure

procedure A^{7+j-1}; begin WRITE(‘s_i’) end;
$j = 2, \ldots, p,$

where s_i is the j-th partial string which is different from '. Let now AP be the set of names of the procedures previously generated. Then we have:

$$AP = \{A^{7+1}, A^{7+2}, \ldots, A^{7+p-1}\}.$$

Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be the set of the partial strings in the division of S. Let us introduce the two functions $\mathcal{G}$ and $\tilde{\mathcal{G}}$:

$$\mathcal{G} : [p] \to \mathcal{S}$$
$$j \mapsto s_k, \text{ where } s_k \text{ is the j-th partial string} \neq'$$

$$\tilde{\mathcal{G}} : [n] \to AP \cup \{A^5\}$$
$$j \mapsto \begin{cases} \perp & \text{if } j = 1 \\ A^{7+i-1} & \text{if } s_j \text{ is the j-th partial string in } S \neq' \\ A^5 & \text{if } s_j =' \end{cases}$$

where the symbol $\perp$ means that the function is undefined. Let us recall that A^5 is the procedure in π_6 which produces the symbol $'$. Since

neither $<$ label declaration part $> \pi'$
nor $<$ constant declaration part $> \pi'$
nor $<$ type definition part $> \pi'$
nor $<$ procedure and function declaration part $> \pi'$

can begin or end with the symbol $'$, we have

$$\mathcal{G}(1) = s_1 \text{ and } \mathcal{G}(p) = s_n.$$

A string T will be transformed into a string T' by simply appending symbol ; to it; $T' := T;$. Similarly, the string T' will be broken down into $m \geq 1$ partial strings t_i. With respect to this decomposition we have:

(i) $t_i =$ ' or $t_i \neq$ ', $\quad i \in [m]$
(ii) $t_i \neq$ ' $\Rightarrow t_{i+1} =$ ', $\quad i \in [m-1]$
(iii) $t_1 \circ t_2 \circ \ldots \circ t_m = T'$

Let q be the number of the partial strings of T' differing from string $'$. We thus have $1 \leq q < m$. For every partial string t_i different from $'$, except for t_m, we generate a procedure

procedure$A^{7+j+p-1}$; begin WRITE('t_i') end;

for $j = 2, \ldots, q-1$ and where t_i is the j-th partial string which is not equal to $'$ or equivalently

procedureA^{7+p}; begin WRITE('BEGIN t_j') end;

for $j = 1$. Moreover, t_j is the j-th partial string of T' different from $'$. In a similar way, $AP, \mathcal{S}, \mathcal{G}$ and $\tilde{\mathcal{G}}$ will define $AQ, \mathcal{T}, \tau, \tilde{\tau}$ as follows:

$$AQ = \{A^{7+p}, \ldots, A^{7+p+q-1}\}$$
$$\mathcal{T} := \{t_1, \ldots, t_m\}$$
$$\tau : [q] \to \{t_i\}, \quad i \in [n]$$
$$j \mapsto t_k, \text{ where } t_k \text{ is the j-th partial string of } T' \neq'$$
$$\tilde{\tau} : [m] \to AQ \cup \{A^5\}$$
$$j \mapsto \begin{cases} \perp & \text{if } j = q \\ A^{7+p+i-1} & \text{if } t_j \text{ is the i-th partial string} \neq' \\ A^5 & \text{if } t_j =' \end{cases}$$

The first symbol right after the initial begin label in the string $<$ statement $> \pi'$ cannot be equal to the symbol $'$. Thus we have from the definition of $T' : \tau(1) = t_1$. The last symbol in T' is the symbol ;. Consequently, it follows that $\tau(q) = t_m$. It is now possible to give the two missing program parts of $\tilde{\pi}$, in other words:

$<$ procedure and function declaration part $> \tilde{\pi}$ and $<$ statement part $> \tilde{\pi}$.
$= <$ procedure and function declaration part $> \tilde{\pi}$
$= <$ procedure and function declaration part $> \pi'$
procedureA^{7+1}; begin …end;
⋮
procedure$A^{7+p+q+1}$; begin …end;
procedure A; begin WRITE('$<$program heading$>$ $\tilde{\pi}$ $\mathcal{G}(1)$ $')$*end*;
⋮
procedureA^7; begin WRITE('$\tau(q) \circledast$ END.') end;
$<$ statement part $> \tilde{\pi}$
$=$ begin $t_1, \ldots, t_m \circledast$ end.

The part of the program described by the symbol $\circledast$ represents the abbreviate sequence of calls of procedures A to A^{p+q+2}, which produces the code of $\tilde{\pi}$. The total result is thus given by:

$\tilde{\pi}$ = program $P(\mu_1, \ldots, \mu_k, \text{OUTPUT})$;
$s_1 \ldots s_n$
procedureA^{7+1}; begin WRITE('$\mathcal{G}(2)$') end;
procedureA^{7+2}; begin WRITE('$\mathcal{G}(3)$') end;
⋮
procedureA^{7+p-1}; begin WRITE('$\mathcal{G}(p)$') end;
procedureA^{7+p}; begin WRITE('BEGIN $\tau(1)$') end;
procedureA^{7+p+1}; begin WRITE('$\tau(2)$') end;
⋮
procedure$A^{7+p+q-2}$; begin WRITE('$\tau(q-1)$') end;
procedure$A$; beginWRITE('PROGRAM$P(\mu_1, \ldots, \mu_k,$ OUTPUT); $\mathcal{G}(1)$') end;
procedureA^2; begin WRITE('PROCEDURE') end;
procedureA^3; begin WRITE(';BEGIN WRITE(' ') end;
procedureA^4; begin WRITE(' ') END;') end;
procedureA^5; begin WRITE(' ') end;
procedureA^6; begin WRITE(' A ') end;
procedureA^7; begin WRITE('$\tau(q) \circledast$ END.') end;
begin
$t_1 \ldots t_m$
A; $\quad *$line numbering
$\tilde{\mathcal{G}}(2); \ldots; \tilde{\mathcal{G}}(n); \quad * \downarrow$
$A^2; (A^6;)^{7+1} A^3; A^{7+1}; A^4; \quad *k_{7+1}$
$A^2; (A^6;)^{7+2} A^3; A^{7+2}; A^4; \quad *k_{7+2}$
$\ldots \quad * \ldots$

$A^2; (A^6;)^{7+p-1} A^3; A^{7+p-1}; A^4; \quad *k_{7+p-1}$
$A^2; (A^6;)^{7+p} A^3; A^{7+p}; A^4; \quad *k_{7+p}$
$A^2; (A^6;)^{7+p+1} A^3; A^{7+p+1}; A^4; \quad *k_{7+p+1}$
$\ldots \quad * \ldots$
$A^2; (A^6;)^{7+p+q-2} A^3; A^{7+p+q-2}; A^4; \quad *k_{7+p+q-2}$
$A^2; A^6; A^3; A; A^4; \quad *k_1$
$A^2; (A^6;)^2 A^3; A^2; A^4; \quad * \ldots$
$A^2; (A^6;)^3 A^3; A^3; A^5; A^4; \quad * \ldots$
$A^2; (A^6;)^4 A^3; A^5; A^4; A^4; \quad * \ldots$
$A^2; (A^6;)^5 A^3; A^5; A^5; A^4; \quad * \ldots$
$A^2; (A^6;)^6 A^3; A^6; A^4; \quad * \ldots$
$A^2; (A^6;)^7 A^3; A^7; A^4; \quad *k_7$
$\tilde{\tau}(1); \ldots; \tilde{\tau}(m-1); \quad *$
A^7; WRITELN $*$
end.

where all the lines including the symbol * denotes the abbreviated program part $\circledast$.

Second case: S is equal to the empty word (in other words π has an empty declaration part). This is a special case of the previous one. We obtain the required program by removing the string S in the previous program $\tilde{\pi}$. To be more precise:

- we remove procedures A^{7+1} to A^{7+p-1};
- we modify the procedure A in procedure A; begin WRITE('PROGRAM P($\mu_1, \ldots, \mu_k$, OUTPUT); ') end;
- we remove the program line $\tilde{\mathcal{G}}(2); \ldots; \tilde{\mathcal{G}}(n)$;
- we remove the program lines k_{7+1} to k_{7+p-1}.

Part B:

First case: S is different from the empty word. The construction in Part A produces a syntactically correct PASCAL program $\tilde{\pi}$. We just have to prove that $\tilde{\pi}$ is a self-reproducing version of π.
Program π is computing a function

$$f_\pi : (A^*_{\mathrm{PAS}})^r \to (A^*_{\mathrm{PAS}})^u \quad \text{with } 0 \le u \le r.$$

The declaration part of π will be integrated unmodified in program $\tilde{\pi}$ in the form $S = s_1 \ldots s_n$. In the instruction (code) part of $\tilde{\pi}$, we start by executing the instruction part from π which has the form $T' = t_1 \ldots t_m$. Other instructions are just calls to procedures not declared in S. Every call of these procedures will write a constant code text to the OUTPUT file. Since π contains a finite number of procedures, a finite-length source code text is written to the OUTPUT file, which is also a word $y \in A^*_{\mathrm{PAS}}$. Calls to these output procedures take place only after the instruction part T' has been executed. Hence $\tilde{\pi}$ is computing the following function:

$$f_{\tilde{\pi}} : (A^*_{\mathrm{PAS}})^r \to (A^*_{\mathrm{PAS}})^u, \qquad 0 \le u \le r,$$

with

$$(f_{\tilde{\pi}})_i(\bar{x}) = \left\{ \begin{array}{ll} (f_\pi)_i(\bar{x}) & \text{if } i \in [r-1] \\ (f_\pi)_i \circ y & \text{if } i = r \end{array} \right\}$$
$$\bar{x} \in (A^*_{\mathrm{PAS}})^r,$$

if $\mu_r = $ OUTPUT, in other words $\tilde{\pi}$ writes y to an output file used by π, as well. That is to say

$$f_{\tilde{\pi}} : (A^*_{\mathrm{PAS}})^{r+1} \to (A^*_{\mathrm{PAS}})^u, \quad 0 \le u \le r+1,$$

with

$$(f_{\tilde{\pi}})_i(\bar{x}) = \left\{ \begin{array}{ll} (f_\pi)_i(\bar{x}) & \text{if } i \in [r] \\ y & \text{if } i = r+1 \end{array} \right\}$$
$$\bar{x} \in (A^*_{\mathrm{PAS}})^{r+1},$$

if $\mu_r \neq$ OUTPUT, in other words $\tilde{\pi}$ writes y to output file OUTPUT, which is not used by π as output file. Program $\tilde{\pi}$ complies with Definition 5.2 when the source code of $\tilde{\pi}$ is a partial string of y. But we even have $y = \tilde{\pi}$, because:
After having worked on $t_1 \ldots t_m$, program $\tilde{\pi}$ first produces its own program header < program heading > $\tilde{\pi}$ and s_1 by calling procedure A. The next procedure call

$$\tilde{\mathcal{G}}(2); \ldots \text{ to } \ldots; \tilde{\mathcal{G}}(n);$$

results in the output of

$$s_2 \ldots \text{ to } \ldots s_n.$$

The procedure calls of lines k_{7+1} until $k_{7+p+q-2}$ and k_1 until k_7 result in outputting the declaration of procedures A^{7+1} to $A^{7+p+q-2}$, in other words of A to A^7. Only < statement part > $\tilde{\pi}$ remains, and is effected through the series $\tilde{\tau}(1); \ldots; \tilde{\tau}(m-1); A^7;$. The next WRITELN instruction just clears the buffer of the OUTPUT file. We also have $y = \tilde{\pi}$. Consequently, $\tilde{\pi}$ is a self-reproducing version of π.

Second case: S is equal to the empty word. It is just a special case of the previous one. The program $\tilde{\pi}$ is a self-reproducing version of π as well.

Hence the proposition is proved. □

We can derive an algorithm directly from the proof of Proposition 5.1, which yields a self-reproducing version $\tilde{\pi}$ for a given PASCAL program π.

Algorithm 5.1 *Input: A* PASCAL *program* π.

Step 1: Testing of labels or names in π *which does not belong to* $\{A\}^+$. *If any, perform the renaming.*

Step 2: Formulation of $<$ *program heading* $>$ $\tilde{\pi}$ *from* $<$ *program heading* $>$ π *using the standard* OUTPUT *file.*

Step 3: Decomposition of

$$\begin{aligned} S &= < \textit{label declaration part} > \pi \\ &= < \textit{constant declaration part} > \pi \\ &= < \textit{type definition part} > \pi \\ &= < \textit{variable declaration part} > \pi \end{aligned}$$

into partial strings $s_1, \ldots, s_n$ *and look for the number* p *of partial strings* s_j *different from* $'$. *Then, express the* $p-1$ *procedures* A^{7+1} *to* A^{7+p-1} *and create the value table for* $\mathcal{G}$ *and* $\tilde{\mathcal{G}}$.

Step 4: Set $T' := T$ *if the last symbol of* $T =<$ *statement part* $> \pi$ (*without the begin and end. surrounding labels*) *is equal to* ; . *Otherwise, set* $T' := T$;.
Decompose T' *into* $t_1 \ldots t_m$ *and establish the number* q. *Then, formulate the* $q-1$ *procedures* A^{7+p} *to* $A^{7+p+q-2}$ *and create the value table for* τ *and* $\tilde{\tau}$.

Step 5: Put the function values we have obtained, as well as procedures and partial strings $s_1, \ldots, s_n, t_1, \ldots t_m$ *in the program skeleton given in the proof.*

Complexity Analysis: This algorithm has a linear complexity $\mathcal{O}(l(\pi))$ in the size $l(\pi)$ of the program π.

Example 5.4 Let us consider the program presented in [28, p. 17].

```
π = program CONVERT(OUTPUT);
    const ADDIN=32; MULBY=1.3; LOW=0; HIGH=39;
    SEPARATOR='_____________';
    var DEGREE : LOW . . HIGH;
    begin
    WRITELN(SEPARATOR);
    for DEGREE := LOW to HIGH do
    begin WRITE(DEGREE,'C',
          ROUND(DEGREE*MULBY+ADDIN),'F');
          if ODD(DEGREE) then WRITELN
    end;
    WRITELN;
    WRITELN(SEPARATOR)
    end.
```

Let us now apply Algorithm 5.1.

Step 1: π *does not contain any labels from* $\{A\}^+$. *Thus we do not need to rename.*

Step 2: $<$ *program heading* $>$ $\tilde{\pi}$ *is set as program CONVERTX(OUTPUT);.*

Step 3: $S = s_1\ s_2\ s_3\ s_4\ s_5$ *with*

$s_1 =$ const ADDIN=32; MULBY=1.3; LOW=0;
HIGH=39;SEPARATOR=
$s_2 =$ '
$s_3 =$ _____
$s_4 =$ '
$s_5 =$;var DEGREE : LOW . . HIGH;

We thus have $n = 5$ *and* $p = 3$. *The resulting procedure consequently is:*

```
procedure A^8; begin WRITE('_____________') end;
procedure A^9; begin WRITE(';VAR DEGREE:
  LOW . . HIGH;') end;
```

The values for $\mathcal{G}$ *and* $\tilde{\mathcal{G}}$ *are then:*

$\mathcal{G}(1) = s_1$
$\mathcal{G}(2) = s_3$
$\mathcal{G}(3) = s_5$
$\tilde{\mathcal{G}}(1) = \perp$
$\tilde{\mathcal{G}}(2) = A^5$
$\tilde{\mathcal{G}}(3) = A^8$
$\tilde{\mathcal{G}}(4) = A^5$
$\tilde{\mathcal{G}}(5) = A^9$

Step 4: we have $T = t_1\ t_2\ t_3\ t_4\ t_5\ t_6\ t_7\ t_8\ t_9$ *with*

$t_1 =$ WRITELN(SEPARATOR); FOR DEGREE := LOW TO HIGH
DO BEGIN WRITE('DEGREE,
$t_2 =$ '
$t_3 =$ C
$t_4 =$ '
$t_5 =$,ROUND(DEGREE*MULBY+ADDIN)'
$t_6 =$ '
$t_7 =$ F
$t_8 =$ '
$t_9 =$); IF ODD(DEGREE) THEN WRITELN END;
WRITELN;WRITELN(SEPARATOR);

We have $m = 9$ *and* $q = 5$. *As a consequence, the resulting procedures are:*

```
procedure A^10; begin WRITE('BEGIN WRITELN(SEPARATOR);
   FOR DEGREE:= LOW TO HIGH DO BEGIN WRITE(DEGREE),') end;
procedure A^11; begin WRITE('C') end;
procedure A^12; begin WRITE('ROUND(DEGREE*MULBY+ADDIN),') end;
procedure A^13; begin WRITE('F') end;
```

The values for τ and $\tilde{\tau}$ are then:

$$\begin{array}{lll}\tau(1)=t_1 & \tilde{\tau}(1)=A^{10} & \tilde{\tau}(1)=A^5\\ \tau(2)=t_3 & \tilde{\tau}(2)=A^5 & \tilde{\tau}(7)=A^{13}\\ \tau(3)=t_5 & \tilde{\tau}(3)=A^{11} & \tilde{\tau}(8)=A^5\\ \tau(4)=t_7 & \tilde{\tau}(4)=A^5 & \tilde{\tau}(9)=\perp\\ \tau(5)=t_9 & \tilde{\tau}(5)=A^{12} & \end{array}$$

Step 5: Finally, we have:

```
π̃ = program CONVERTX(OUTPUT);
    const ADDIN=32;MULBY=1.3; LOW=0; HIGH=39;
    const SEPARATOR='_____________';
    var DEGREE: LOW . . HIGH;
    procedure A^8; begin WRITE('_____________') end;
    procedure A^9; begin WRITE(';VAR DEGREE: LOW . . HIGH;') end;
    procedure A^10; begin WRITE('BEGIN WRITELN(SEPARATOR); FOR
        DEGREE := LOW TO HIGH DO BEGIN WRITE(DEGREE)') end;
    procedure A^11; begin WRITE('C') end;
    procedure A^12; begin WRITE(',ROUND(DEGREE*MULBY+ADDIN),') end;
    procedure A^13; begin WRITE('F') end;
    procedure A; begin WRITE('PROGRAM CONVERTX(OUTPUT); CONST
        ADDIN=32;MULBY=1.3; LOW=0; HIGH=39;SEPARATOR=') end;
    procedure A^2; begin WRITE('PROCEDURE') end;
    procedure A^3; begin WRITE(';BEGIN WRITE(' ') end;
    procedure A^4; begin WRITE(' ' ') END;') end;
    procedure A^5; begin WRITE(' ' ') end;
    procedure A^6; begin WRITE(' A ') end;
    procedure A^7; begin WRITE('); IF ODD(DEGREE) THEN WRITELN END;
        WRITELN;WRITELN(SEPARATOR); ⊛ END.') end;
    begin
    WRITELN(SEPARATOR);
    for DEGREE:= LOW to HIGH do
    begin WRITE(DEGREE,'C', ROUND(DEGREE*MULBY+ADDIN),'F');
        if ODD(DEGREE) then WRITELN
    end;
    WRITELN;
    WRITELN(SEPARATOR);
    ⊛
```

where the program part denoted by ⊛ is given by:

$A;$
$A^5; A^8; A^5; A^9;$
$A^2; (A^6;)^8 A^3; A^8; A^4;$
$A^2; (A^6;)^9 A^3; A^9; A^4;$
$A^2; (A^6;)^{10} A^3; A^{10}; A^4;$
$A^2; (A^6;)^{11} A^3; A^{11}; A^4;$
$A^2; (A^6;)^{12} A^3; A^{12}; A^4;$
$A^2; (A^6;)^{13} A^3; A^{13}; A^4;$
$A^2; A^6; A^3; A; A^4;$
$A^2; (A^6;)^2 A^3; A^2; A^4;$
$A^2; (A^6;)^3 A^3; A^5; A^3; A^4;$
$A^2; (A^6;)^4 A^3; A^5; A^4; A^4;$
$A^2; (A^6;)^5 A^3; A^5; A^5; A^4;$
$A^2; (A^6;)^6 A^3; A^6; A^4;$
$A^2; (A^6;)^7 A^3; A^7; A^4;$
$A^{10}; A^5; A^{11}; A^5; A^{12}; A^5; A^{13}; A^5; A^7;$
WRITELN
end.

We stress that the significance of Proposition 5.1 is of theoretical nature only. Proposition 5.1 indeed proves the existence of a self-reproducing version $\tilde{\pi}$ for every valid PASCAL program π and even gives a construction of a syntactically correct program $\tilde{\pi}$. However, it still does not guarantee its realization on a real computer. The practical implementation of Algorithm 5.1 to produce a program $\tilde{\pi}$ can lead to the following difficulties:

1. The longest procedure name of $\tilde{\pi}$ is $A^{7+p+q-2}$. This procedure is $p+q+5$ characters long. Every character is significant, since the procedure name $A^{7+p+q-3}$ of length $p+q+4$ appears in $\tilde{\pi}$ as well. The PASCAL compiler, however, imposes a practical limit on the number of significant label symbols. The number p and q are finite, which consequently implies that, given large values for p and q, some procedures could not be differentiated.
2. The length of the constant parts of the code in procedures A^j for $j \in [p+q-2]$, as determined by Algorithm 5.1, is not constrained for all PASCAL programs $\tilde{\pi}$. As such, neither is the length of a program line. In practice, however, the length of a program line is limited by the size of the input buffer used by the PASCAL compiler.

Difficulties (1) and (2) can generally be avoided in most real-life programs if we add two additional, practical steps to Algorithm 5.1.

Step 6: Let g be the number of significant label symbols for a given PASCAL *compiler. Let $a = 7+p+q-2 = p+q+5$ be the length of the procedure name A^{p+q+5}. At the same time, a is also the number of procedure names having the same type as A^j, for $j \in [a]$ have. Then we choose two natural numbers $c \le 26$ and $b \le g$ such that*

$$\sum_{k=1}^{b} c^k \ge a.$$

Now we can replace procedure names A^1 to $A^{7+p+q-2}$ with new procedure names which have at most b characters taken from the first c symbols of the alphabet. Besides character 'A', we can also use $c-1$ other character in order to create procedure names. But $c-1$ new procedures must be generated for every one of those $c-1$ characters, with each of those procedures printing a new character:

$$c-1 \begin{cases} \text{procedure ...; begin WRITE('B') end;} \\ \text{procedure ...; begin WRITE('C') end;} \\ \vdots \qquad\qquad \vdots\ \vdots \end{cases}$$

For those procedures we will also need names. Consequently, the number c and b must satisfy the following equation:

$$\sum_{k=1}^{b} c^k \geq a + c - 1.$$

Step 7: Let d be a buffer length used by a given PASCAL *compiler. Let v_i for $i \in [p+q+2]$ be the text constants of procedures A^i (these procedures may have possibly been renamed in the previous step). For every $i \in [p = q-2]$, we must perform the following computation:*
If $l(WRITE(`v'_i)) \leq d$ then $A^i$ remains unmodified.*[8] *Otherwise, $v_i$ will be decomposed into $k_i$ partial strings $v_{ij}$ through $l(WRITE(`v'_{ij})) \leq d, \forall j \in [k_j]$. For that purpose, the k_i are chosen as small as possible.
The procedure A^i will be replaced by k_i new procedures:

procedure ...; begin WRITE(‘v'_{i_1})end;
⋮ ⋮ ⋮
procedure ...; begin WRITE(‘$v'_{i_{k_i}}$)end;

Complying with the newly introduced procedures, we amend the procedures call sequence which effect the output of $\tilde{\pi}$.

Steps 6 and 7, however, are still insufficient to reach our goal. If the input buffer length d of the PASCAL compiler is relatively small, step 7 will then generally generate a large number of new procedures. The constant parts of the source code in procedure A^7—possibly renamed in step 6; this procedure contains the output algorithm for $\tilde{\pi}$—will also surely be split in a number of new procedures.

New procedures necessitate a longer output algorithm should $\tilde{\pi}$ remain self-reproducing. This, however, means that still more procedures are necessary for the design of the output algorithm. But as a consequence, additional procedures will effect another increase in this algorithm, which in turn will produce still more procedures and so on. Now if d is relatively small, this general process may not converge to a stable state and step 7 will result in an infinitely long program. This case will arise precisely when the constant parts of the procedures containing the output algorithm contain themselves fewer procedure calls (on average) than required for the construction of an output procedure. The rapid increase in the number of output procedures can lead to repeated execution of step 6. The situation is even more dire when the output of $\tilde{\pi}$ has to be formatted.

[8] The notation l describes with respect to a base alphabet B the length function of elements from B^*.

Example 5.5 Let us consider the implementation of program $\tilde{\pi}$ presented in Example 5.4.

Step 6: The program $\tilde{\pi}$ contains 13 procedure names in the form of A^i. The given PASCAL *compiler considers as significant only the first 8 characters of a label. A few of the A^i must be renamed as well. Coincidentally, procedures A^{11} and A^{13} make the characters ‘C’ and ‘F’ available. As for the renaming of procedures $A^1, \ldots A^{13}$ we must use four letters (characters) in total. That is why we thus consider the following additional procedure*

procedure BB; begin WRITE(‘B’) end;

in program $\tilde{\pi}$. With the four characters A, B, C and F, we can then build:

- *4 different names of length 1,*
- *16 different names of length 2 and,*
- *64 different names of length 3.*

Moreover, should step 7 produce additional procedures, we may assume that with $a = 4$ and $b = 3$ we have enough names at our disposal. We will try to make do with names of length 1 and 2.
The procedures A^i, $i \in [13]$ will be renamed as follows:

A^1 *renamed as AA*	A^8 *renamed as AF*
A^2 *renamed as A*	A^9 *renamed as CB*
A^3 *renamed as B*	A^{10} *renamed as CC*
A^4 *renamed as C*	A^{11} *renamed as BC*
A^5 *renamed as F*	A^{12} *renamed as CF*
A^6 *renamed as BA*	A^{13} *renamed as BF*
A^7 *renamed as AC*	

Step 7: Let the length of a program line be limited to 132 characters. Hence, we will have to split the constant parts of procedure AC into additional procedures. The induced administrative effort requires the additional introduction of four new procedures FA, FB, FC and FF. Appendix A.12 in ESM gives the modified program $\tilde{\pi}$ once steps 6 and 7 have been applied. Procedure Q, which was introduced in Sect. 3.3.4, is used to format the output.

5.3 Self-reproducing principles with respect to the SIMULA programming language

The PASCAL and SIMULA example programs presented in Chaps. 3 and 4 are roughly equivalent. Hence, as a pendant to the self-reproducing PASCAL program π_6 in Sect. 3.3.2 there exists a near-identical SIMULA program π_4 in Sect. 3.2.7.

Since the proof of Proposition 5.1 is mostly based on the existence of program π_6, we can assume that there exist a similar proposition with respect to the SIMULA programming language. The proof of the latter proposition will be divided in two parts A and B, just as we proceeded to do for Proposition 5.1.

In part A, we will first consider the design of a self-reproducing version $\tilde{\pi}$ of an arbitrary SIMULA program π. Despite the correspondence between π_4 and π_6, this construction must be explicitly given, since unlike PASCAL, SIMULA is not a block-oriented language. However, the resulting program $\tilde{\pi}$ will to a great extent match the program built for the proof of Proposition 5.1. Since $\tilde{\pi}$ is the self-reproducing version of π, we will refer to part B of the proof of Proposition 5.1 when we present part B of our current proof. Our proof will adopt the SIMULA grammar given in [19] and will be denoted by G_{sim}. Additionally for G_{sim}, we will adopt the convention 5.1. Lastly, we will adopt Convention 5.2 for the procedure names of π_4. Hence, program π_4 will have the following form:

$\pi_4 =$ begin
procedure A; OUTTEXT("BEGIN PROCEDURE A;
OUTTEXT("""");
procedure A^2; OUTTEXT("PROCEDURE ");
procedure A^3; OUTTEXT(" ; OUTTEXT("""");
procedure A^4; OUTTEXT("""");");
procedure A^5; OUTTEXT("""");
procedure A^6; OUTTEXT("A");
procedure A^7; OUTTEXT("A; A; A^5; A^4; A^2; $(A^6;)^2$
A^3; A^2; A^4; $A^2(A^6;)^3A^3$; A^3; A^5; A^4; A^2; $(A^6;)^4$
A^3; A^5; A^4; A^4; A^2; $(A^6;)^5A^3$; A^5; A^5; A^4; A^2;
$(A^6;)^6A^3$; A^6; A^4; A^2; $(A^6;)^7A^3$; A^7; A^4; A^7
END");
A; A; A^5; A^4;
A^2; $(A^6;)^2A^3$; A^2; A^4;
$A^2(A^6;)^3A^3$; A^3; A^5; A^4;
A^2; $(A^6;)^4A^3$; A^5; A^4; A^4;
A^2; $(A^6;)^5A^3$; A^5; A^5; A^4;
A^2; $(A^6;)^6A^3$; A^6; A^4;
A^2; $(A^6;)^7A^3$; A^7; A^4; A^7;
end

Proposition 5.2 *For every syntactically valid* SIMULA *program π, there exists a self-reproducing version $\tilde{\pi}$ of π.*

Proof Part *A*:
The program π_4 has the following structure from the grammar G_{sim}:

$\pi_4 =$ begin $<$ declaration part of $>\pi_4$
$<$ instruction part of $>\pi_4$ end

Let π be any SIMULA program. Then, π has the following structure:

$\pi =$ $<$ class description $>\pi$
$<$ optional parameter part of $>\pi$
begin $<$ declaration part of $>\pi$
$<$ instruction part of $>\pi$ end

The parts $<$ *class description* $>\pi$, $<$ *optional parameter part of* $>\pi$ and $<$ *declarationpart of* $>\pi$ can be empty (compare with π_4). If program π is composed of

begin $<$ instruction part of $>\pi$ end,

only, then π is called an assembly instruction. Otherwise π is a block. Not only assembly instructions but also blocks are allowed at any position within $<$ *instruction part of* $>\pi$. We will use this later on.

Combination of $\tilde{\pi}$ from π_4 and π
Whenever $<$ *class description* $>\pi$ is not empty, we have then to test whether $<$ *class description* $>\pi$ is an element in $\{A\}^+$. In this case, we will rename the $<$ *class description* $>\pi$. As a result, we get the program π' which realizes the same functions as π does. Otherwise, we set $\pi' := \pi$. The concept of local variables obviates the need to rename variables in $<$ *declaration part of* $>\pi$ and $<$ *instruction part of* $>\pi$.

$\tilde{\pi} =$ begin
procedure A; . . . ;
(additional declaration)
procedure A^2; . . . ;
⋮ } ⟵ { with modified procedure A^7 from π_4
procedure A^7; . . . ;
π';
(additional instructions)
$<$ sequence of instructions from π_4 $>$
end

We need now just define the "additional declaration" and "additional instructions" parts or the program.
Let T be program π' with the additional ; character. In other words, we have $T = \pi';$. The character string T will be split into $m \geq 1$ partial strings t_i, just as the string T' was split in the proof of Proposition 5.1. Hence, we have for the decomposition of T:

a) $t_i =$ " or " $\notin t_i$, $i \in [m]$

b) $" \notin t_i \Rightarrow t_{i+1} = "$, $i \in [m-1]$
c) $t_1 \circ t_2 \circ \ldots \circ t_m = T$

Let q be the number of partial strings t_j from T not equal to the string ". We have $1 \leq q \leq m$. For each partial string t_j not equal to " with exception of partial string t_m, we generate a procedure of the following form:

$$\text{procedure}A^{7+j}; \text{OUTTEXT}("t_i"); \qquad j \in [q-1]$$

where t_i is the j-th partial string from T not equal to ". The set of the procedures we have produced is $AQ = \{A^{7+1}, \ldots, A^{7+q-1}\}$. Let $\mathcal{T}$ be the set $\{t_1, \ldots, t_m\}$. As in the proof of Proposition 5.1, we define the functions τ and $\tilde{\tau}$ as follows:

$$\tau : [q] \to \mathcal{T}$$
$$j \mapsto t_k, \text{ where } t_k \text{ is the j-th partial string} \neq "$$

$$\tilde{\tau} : [m] \to AQ \cup \{A^5\}$$
$$j \mapsto \begin{cases} \perp & \text{if} \quad j = q \\ A^{7+i} & \text{if} \quad t_j \text{ is the i-th partial string} \neq " \\ A^5 & \text{if} \quad t_j = " \end{cases}$$

The partial string t_1 is different from " since π cannot begin with the " character. Since the last character of T is equal to ;, then $t_m = t_q$ is different from string ". The string t_m will be merged into procedure A^7.
The part of the program "additional declarations" contains the $q-1$ declarations of procedures A^{7+1} to A^{7+q-1}. As for the part of the program "additional instructions", it contains the sequence of procedure calls which are necessary to produce the $q-1$ additional procedure declarations. The procedure A^7 will be developed accordingly.

$\tilde{\pi}$ = begin
procedure A; OUTTEXT("BEGIN PROCEDURE A;
OUTTEXT("""");
procedureA^{7+1}; OUTTEXT("$\tau(1)$");
...
procedureA^{7+q-1}; OUTTEXT("$\tau(q-1)$");
procedureA^2; OUTTEXT("PROCEDURE");
procedureA^3; OUTTEXT(" ; OUTTEXT("""");
procedureA^4; OUTTEXT("""");");
procedureA^5; OUTTEXT("""");
procedureA^6; OUTTEXT("A");
procedureA^7; OUTTEXT("$\tau(m)$; A; A; A^5; A^4; A^2;
$(A^6;)^{7+1}A^3$; A^{7+1}; A^4
; ... A^2; $(A^6;)^{7+q-1}A^3$; $A^{7+q-1}A^4$; A^2; $(A^6;)^2$
A^3; A^2; A^4; A^2;
$(A^6;)^3A^3$; A^3; A^5; A^4; A^2; $(A^6;)^4A^3$; A^5; A^4; A^4;
A^2; $(A^6;)^5A^3$; A^5; A^5; A^4; A^2;
$(A^6;)^6A^3$; A^6; A^4; A^2; $(A^6;)^7A^3$; A^7; A^4; $\tilde{\tau}(1)$;
...; $\tilde{\tau}(m-1)$; A^7 END");
$t_1 \ldots t_m$
A; A; A^5; A^4;
A^2; $(A^6;)^{7+1}A^3$; A^{7+1}; A^4;
...
$A^2(A^6;)^{7+q-1}A^3$; A^{7+q-1}; A^4;
A^2; $(A^6;)^2A^3$; A^2; A^4;
A^2; $(A^6;)^3A^3$; A^3; A^5; A^4;
A^2; $(A^6;)^4A^3$; A^5; A^4; A^4;
A^2; $(A^6;)^5A^3$; A^5; A^5; A^4;
A^2; $(A^6;)^6A^3$; A^6; A^4;
A^2; $(A^6;)^7A^3$; A^7; A^4;
$\tilde{\tau}(1)$; ...; $\tilde{\tau}(m-1)$; A^7
end

Part B:

The construction of $\tilde{\pi}$ in Part A leads to a new syntactically correct SIMULA program. Because of the close correspondence between the SIMULA program π_4 and the PASCAL program π_6, the program $\tilde{\pi}$ easily corresponds to the PASCAL program given in the proof of Proposition 5.1. To prove that $\tilde{\pi}$ is really the self-reproducing version of π, it is sufficient to refer to Case 2 of the proof of the Proposition 5.1. □

From the proof of Proposition 5.2, we directly obtain an algorithm which produces for a valid SIMULA program π a self-reproducing version $\tilde{\pi}$.

Algorithm 5.2 *Input: A* SIMULA *program* π.

Step 1: Test whether < *class description* > π *– if any – is a label from* $\{A\}^+$. *Rename if need be;*
Step 2: Split $T := \pi$; *into partial strings* $t_1, \ldots, t_m$ *and determine the number* q *of partial strings* t_j *which are different from* "; *then express the* $q-1$ *procedures* A^{7+1} *to* A^{7+q-1} *and build the value table of* τ *and* $\tilde{\tau}$;
Step 3: Insert the obtained procedures, the function values and the program π *into the program skeleton given in the previous proof;*
Complexity: The complexity of this algorithm is linear in the length $l(\pi)$ *of the program* π.

Due to the analogy with Algorithm 5.1, we will not give a working example of that algorithm. We shall note, however, that a practically implementable self-reproducing version requires the extension of Algorithm 5.1 by two more practical steps, as was the case with Algorithm 5.2.

In conclusion, Chap. 5 has show that questions (1), (2) and (3) posed at the beginning can be answered affirmatively with respect to SIMULA and PASCAL programming languages. Those answer did not involved any specific characteristic or element of those two languages. We conclude that questions (1), (2) and (3) can be fruitfully and constructively addressed in any high level programming language, provided the concepts of procedures and code constants are available.

As far as the SIEMENS Assembly language in Sect. 3.4 is concerned, Example 3.1 gives the answer to the previous questions. Only a few assembly code lines are necessary in order to transform a given assembly program part into a self-reproducing segment. The lines which enable self-reproduction are essentially always the same.

6 Self-reproduction with LOOP-programs

6.1 Introduction

In Chaps. 3, 4 and 5, we presented examples of self-reproducing programs in high level programming languages. From an algorithmic point of view, these examples generally were not resource intensive. In addition, the corresponding control flow of each of those programs was rather simple. It would be interesting to investigate just how simple programming language structures could be made while still enabling self-reproducing programs. The following considerations will primarily apply to standard control structures of programming languages in general and will not refer to a particular real programming language. We shall hence break away from real programming languages by concentrating on the fictitious programming language $PL(A)$ presented in Chap. 2.

In Chap. 2, the set of functions implemented by $PL(A)$ programs was denoted by $\mathcal{P}$. By reducing available base instructions and control structures in $PL(A)$ to

$$\begin{aligned}\gamma_1, \gamma_2, \gamma_3, \gamma_5 \cdots x_1 &: P; Q\\ x_2 &: \text{if p then goto L}\end{aligned}$$

or to

$$\begin{aligned}\gamma_1, \gamma_2, \gamma_3, \gamma_5 \cdots x_1 &: P; Q\\ x_3 &: \text{if p then P else Q fi}\\ x_4 &: \text{while} X = \epsilon \text{ do P od}\end{aligned}$$

we obtain programming languages which only allow "goto" programs ((respectively "while" programs). However, theory showed that (compare with [5]) the set of functions implemented with while programs is equivalent to the set of the functions implemented with goto programs and thus is equivalent to the set $\mathcal{P}$ (our examples for self-reproducing programs in SIMULA and PASCAL require procedures. If we want to transform those programs into $PL(A)$ programs, then we have to consider "goto" programs.) Only if we reduced $PL(A)$ further to

$$\begin{aligned}\gamma_1, \gamma_2, \gamma_3, \gamma_5 \cdots x_1 &: P; Q\\ x_5 &: \text{loop} X \text{ case } a_1 \rightarrow P_1,\\ &\quad\ \vdots \quad\ \vdots\\ &\quad\ a_n \rightarrow P_n,\\ &\text{end},\end{aligned}$$

this is to say, to pure "loop" program, we are not longer able to write programs that realize all functions from $\mathcal{P}$. Thus, we shall proceed to determine the necessary requirements for self-reproducing loop programs.

6.2 Definition of the Programming Language $LP(A)$

Definition 6.1 Let $A = \{a_1, \ldots, a_n\}$ be a finite alphabet. Let $PL(A)$ be the programming language defined in Sect. 2.2 belonging to A. We obtain the programming language $LP(A)$ by eliminating all programs from $PL(A)$ which contain base instructions of the form $\gamma_5 : X := \rho(X), \quad X \in VR$ or one of the control structures given by:

x_2 : if p then goto L,

x_3 : if p then P else Q fi

or x_4 : while $X = \epsilon$ do P od

Remark 6.1 1. Besides sequential instruction execution, the x_5 loop structure represents the sole construction element for programs in $LP(A)$. Thus, programs in $PL(A)$ will be described as loop programs, as well.

2. The set of computable functions implemented through programs in $LP(A)$

$$f : (A^*)^r \rightarrow (A^*)^s, \quad r, s \geq 0,$$

is denoted $\mathcal{L}$. The set $\mathcal{L}$ is also called the set of the primitive recursive functions [5].

It follows from Definition 6.1 that $LP(A)$ is a proper subset of $PL(A)$. In addition, loop programs always halt, indicating that the set $\mathcal{L}$ is a proper subset of $\mathcal{P}$, as well; hence the functions realized by loop programs are total functions. We can also see that $\mathcal{L}$ is a proper subset of $\mathcal{R}$ (compare with Definition 2.12) by proving the existence of a total recursive function which is not a primitive recursive function (compare with [5, p. 41]).

6.3 A Context-free Grammar for $LP(A)$

For completeness' sake, let us now specify a context free grammar $G'(A)$ for $LP(A)$. $G'(A)$ derives from the reduction of the grammar $G(A)$ for $PL(A)$ programs in Sect. 2.3.

6.3.1 *Specification of the Grammar* $G'(A) = (V'_T, V'_N, s_0, P')$

The set of terminal symbols is given by

$$V'_T = A \cup VR \cup \{\text{input, output, loop, case, end, } \rightarrow, ;, ,, ,\\ < \text{space} >, \epsilon, \bar{\epsilon}, :, =\},$$

where VR is the set of valid variable names.

The set of non terminal symbols is given by

$$V'_N = \{<\text{program}>, <\text{statement}>, <\text{simple statement}>, \\ <\text{identifier}>, <\text{identifier list}>\}.$$

The start symbol s_0 is $<$ program $>$.

The set P' of productions is equivalent to the production set P of grammar $G(A)$ without productions rules 6, 8, 9, 10, 13, 14, 15 and 20.

6.4 Extending the $LP(A)$ Language

We proved the theoretical existence of self-reproducing programs in $PL(A)$ in Chap. 2 . A similar proof for the existence of self-reproducing programs in $LP(A)$ will fail, since there is no universal function in $\mathcal{L}$ (refer to [5, p. 47]). Hence, we will tackle the problem of the existence of self-reproducing programs in $LP(A)$ from a practical angle. We stress, however, that it is not essentially impossible to theoretically prove the existence of self-reproducing $LP(A)$ programs.

In order to facilitate writing self-reproducing $LP(A)$ programs, we are going to extend the $LP(A)$ programming language by an additional base instruction. Again, we are not claiming that without this extension it would be essentially impossible to write self-reproducing $LP(A)$ programs.

I Let A be a finite alphabet. In the programming language $LP(A)$ with respect to A, we must have the capability to initialize variables with any value from A^* and not only with ϵ. Thus, we introduce the base instruction γ_6:

$$\gamma_6 : X :=' a_{i_1} \dots a'_{i_k} \quad k \geq 1, X \in VR, a_{i_j} \in A \\ \text{for } j \in [k].$$

We cannot exclude the fact that $' \in A$. Thus it is also valid to have $a_{i_j} ='$ for any $j \in [k]$.
In high-level programming language it is common to write the text hyphenation symbol twice when it occurs inside a program text constant. This unfortunate convention complicated the writing of PASCAL and SIMULA self-reproducing programs in Chaps. 3 and 5 more complex. We thus introduce an alternative convention: A γ_6 base instruction must always be followed by a semicolon. The end of a text constant will be then denoted by $';$. For simplicity's sake, we will disallow string $';$ as part of a text constant. In order illustrate this text constant-semicolon concatenation, we include the semicolon in the definition of γ_6.

$$\gamma_6 : X :=' a_{i_1} \dots a'_{i_k}; \qquad k \geq 1, X \in VR, a_{i_j} \in A \\ \text{for } j \in [k].$$

II If $X \in VR$, then the following instruction of type γ_4 is possible:

$$X := X$$

According to the previous convention, a semicolon will follow such an instruction:

$$\dots X := X; \dots$$

Since we cannot exclude that fact that the semicolon is an element from A, the string

$$X := X;$$

can be interpreted also as an instruction of type γ_3. In order to avoid ambiguity,[9] we replace γ_3 by the following base instruction γ'_3:

$$\gamma'_3 : X := X|a \qquad \forall X \in VR, \ a \in A.$$

The meaning of γ'_3 is the same as for γ_3.

Definition 6.2 Let A be a finite alphabet. $\overline{LP(A)}$ is the programming language which is obtained by extending $LP(A)$ with the base instruction γ_6 and by replacing the base instruction γ_3 with γ'_3.

Hence we can derive from the grammar $G'(A)$ of $LP(A)$ a context-free grammar $\overline{G'(A)}$ for the language $\overline{LP(A)}$, as follows:

- The set of terminal symbols V'_T will be extended with the symbols $'$ and $|$.
- The production

  $$<\text{simple statement}> \rightarrow X :=' a_{i_1} \dots a'_{i_k};$$

  for every $X \in VR, a_{i_j} \in A$ will be added to the set P'.
- The production

  $$<\text{simple statement}> \rightarrow X := Xa,$$

  for every $X \in VR, a \in A$ will be replaced by

  $$<\text{simple statement}> \rightarrow X := X|a,$$

  for every $X \in VR, a \in A$.

[9] Which can come from the concrete choice of A and thus were not considered in the content of Chap. 2.

Definition 6.3 (loop-hierarchy of $\overline{LP(A)}$-programs)
Let A be a finite alphabet.

1. $\overline{L_0(A)}$ is the class of $\overline{LP(A)}$-programs

$$\begin{aligned}\pi = \text{input } & X_1, \ldots, X_r;\\ & AW_\pi;\\ \text{output } & Y_1, \ldots, Y_s;\end{aligned}$$

with $r, s \geq 0$, whose instruction part AW_π originates from the successive writings of instructions of type γ_1, γ_2, γ_3', γ_4 and γ_6.

2. The class $\overline{L_{i+1}(A)}$ contains all programs π whose instruction part AW_π is obtained through successive writings of instruction parts from programs in $\overline{L_i(A)}$ and of instruction parts of the form:

$$\begin{aligned}\text{loop } X \text{ case } & a_1 \rightarrow AW_{\pi_1},\\ & \vdots \quad\quad \vdots\\ & a_n \rightarrow AW_{\pi_n},\\ \text{end}&\end{aligned}$$

with $a_j \in A$ for every $j \in [n]$ and where $AW_{\pi_1}, \ldots, AW_{\pi_n}$ are instruction parts of programs from $\overline{L_i(A)}$ $(i \geq 0)$.

6.5 Self-reproducing programs in $LP(A)$

Let π_{rep} be a self-reproducing $\overline{LP(A)}$ program (if any exist). Its code must be constructed step by step in the instruction part of π_{rep}. The single letters in π_{rep} must appear on the right side of assignation values. On the right side of assignation values, however, only characters from A and variable names appear. We thus must have: $\pi_{\text{rep}} \in A^*$. In order to guarantee this, we adopt the following postulate for the alphabet A.

Postulate 6.1 *For every program $\pi \in \overline{LP(A)}$, we have $\pi \in A^*$.*

Any alphabet which fulfils Postulate 6.1 must contain all valid symbols necessary to construct $\overline{LP(A)}$ programs. The smallest alphabet which complies with Postulate 6.1 is given by

$$\begin{aligned}A_{\text{min}} = \{&a, c, d, e, i, l, n, o, p, s, t, u, \epsilon, \bar{\epsilon},\\ &< \text{space} >, :, ;, , , \rightarrow, |, '\}.\end{aligned}$$

The definition of $\overline{LP(A)}$ does not constrain the choice of variable set VR in any way. From Postulate 6.1, it follows however that only characters from A can be considered for variable names of a self-reproducing program $\pi_{\text{rep}} \in \overline{LP(A)}$.

Postulate 6.2 $VR \subseteq \{A \backslash B\}^* \backslash \{case, loop, end, input, output\} \backslash \{\epsilon\}$. *Moreover, we have* $B := \{< space >, :, =, \rightarrow, , , , ; , \epsilon, \bar{\epsilon}, ', |\}$ *as the set of special symbols.*

In a $\overline{LP(A)}$ program which fulfils Postulate 6.2, we must strictly differentiate between variable name x and symbol x, where x is a character in A. However, the definition of $\overline{LP(A)}$ avoids these interpretation errors.

Proposition 6.1 *Let A be a finite alphabet such that $A_{min} \subset A$.*
Then there exists a self-reproducing program in $\overline{L_2(A)}$, provided that the $\overline{LP(A)}$ Postulate 6.2 is met.

Proof Consider the following program π^2_{rep}:

$$\begin{aligned}&\text{input};\\ &a :=' \text{ input};a :=';\ ;\\ &c :=':=\ '\ ';\ ;\\ &d :=\ '\ '\ ';\ ;\\ &e :=' \text{ lnlnoocppnoodpnnooepsnooipunoou';}\ ;\\ &i :=' \text{ loop } e \text{ case } \quad l \rightarrow \text{loop } a \text{ case } \quad i \rightarrow t := t|i,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad n \rightarrow t := t|n,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad p \rightarrow t := t|p,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad u \rightarrow t := t|u,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad t \rightarrow t := t|t,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ; \rightarrow t := t|;\ ,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad a \rightarrow t := t|a,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad :\rightarrow t := t|\ :,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad =\rightarrow t := t|\ =,\\ &\qquad\qquad\qquad\qquad\qquad\quad \text{end},\\ &\qquad\qquad\qquad\quad n \rightarrow \text{loop } d \text{ case } ' \rightarrow t := t|',\\ &\qquad\qquad\qquad\qquad\quad \text{end},\\ &\qquad\qquad\qquad\quad o \rightarrow t := t|;\ ,\\ &\qquad\qquad\qquad\quad c \rightarrow t := t|c,\\ &\qquad\qquad\qquad\quad p \rightarrow \text{loop } c \text{ case } \quad :\rightarrow t := t|\ :,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad =\rightarrow t := t|\ =,\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ' \rightarrow t := t|',\\ &\qquad\quad \text{end},\\ &d \rightarrow t := t|d,\\ &e \rightarrow t := t|e,\\ &s \rightarrow \text{loop } e \text{ case } \quad l \rightarrow t := t|l,\\ &\qquad\qquad\qquad\qquad\quad n \rightarrow t := t|n,\\ &\qquad\qquad\qquad\qquad\quad o \rightarrow t := t|o,\\ &\qquad\qquad\qquad\qquad\quad c \rightarrow t := t|c,\\ &\qquad\qquad\qquad\qquad\quad p \rightarrow t := t|p,\\ &\qquad\qquad\qquad\qquad\quad d \rightarrow t := t|d,\\ &\qquad\qquad\qquad\qquad\quad e \rightarrow t := t|e,\\ &\qquad\qquad\qquad\qquad\quad s \rightarrow t := t|s,\\ &\qquad\qquad\qquad\qquad\quad i \rightarrow t := t|i,\\ &\qquad\qquad\qquad\qquad\quad u \rightarrow t := t|u,\\ &\qquad\quad \text{end},\\ &i \rightarrow t := t|i,\end{aligned}$$

```
u → loop i case  a → t := t|a,
                 c → t := t|c,
                 d → t := t|d,
                 e → t := t|e,
                 i → t := t|i,
                 l → t := t|l,
                 n → t := t|n,
                 o → t := t|o,
                 p → t := t|p,
                 s → t := t|s,
                 t → t := t|t,
                 u → t := t|u,
                 :→ t := t| :,
                 =→ t := t| =,
                 ′ → t := t|′,
                 ; → t := t|; ,
                 , → t := t|, ,
                 →→ t := t| →,
< space >→ t := t| < space >,
        end,
end;
output t′; ;
loop e case   l → loop a case  i → t := t|i,
                               n → t := t|n,
                               p → t := t|p,
                               u → t := t|u,
                               t → t := t|t,
                               ; → t := t|; ,
                               a → t := t|a,
                               :→ t := t| :,
                               =→ t := t| =,
                    end,
              n → loop d case ′ → t := t|′,
                    end,
              o → t := t|; ,
              c → t := t|c,
              p → loop c case  :→ t := t| :,
                               =→ t := t| =,
                               ′ → t := t|′,
        end,
d → t := t|d,
e → t := t|e,
s → loop e case  l → t := t|l,
                 n → t := t|n,
                 o → t := t|o,
                 c → t := t|c,
                 p → t := t|p,
                 d → t := t|d,
                 e → t := t|e,
                 s → t := t|s,
                 i → t := t|i,
                 u → t := t|u,
        end,
i → t := t|i,
u → loop i case  a → t := t|a,
                 c → t := t|c,
                 d → t := t|d,
                 e → t := t|e,
                 i → t := t|i,
                 l → t := t|l,
                 n → t := t|n,
                 o → t := t|o,
                 p → t := t|p,
                 s → t := t|s,
                 t → t := t|t,
                 u → t := t|u,
                 :→ t := t| :,
                 =→ t := t| =,
                 ′ → t := t|′,
                 ; → t := t|; ,
                 , → t := t|, ,
                 →→ t := t| →,
< space >→ t := t| < space >,
        end,
end;
output t
```

π^2_{rep} is obviously a valid $\overline{LP(A)}$ program. We must now prove that π^2_{rep} is self-reproducing.

Since π^2_{rep} has no input, claiming that π^2_{rep} is self-reproducing is equivalent to stating that the content of variable t is equal to π^2_{rep} at the last instruction execution "output t".

I. In the rest of the proof, $[x]$ will denote the content of variable whose name is x, with $x \in \{a, c, d, e, i, t\} \subset A_{\min}$.

II. From the definition of the loop instruction, it is clear that the 4 internal loop instructions add the value of their respective step variable to $[t]$, and as such extend $[t]$. Let us adopt the following abbreviated notation:

$$([t] := [t][y]) := \begin{cases} \text{loop } y \text{ case } \ldots \to \ldots \\ \qquad\qquad\qquad\quad \vdots \\ \qquad\quad \text{end} \end{cases}$$

with $y \in \{a, c, d, e, i\}$. The abbreviated notation $[t] := [t]x$ conversely denotes the fact that the symbol $x \in A$ is attached to the content of $[t]$.

III. With the help of the previous abbreviated notation, we can now formulate π^2_{rep} as follows:

```
input;
a :=′ input;a :=′; ;
c :=′:= ' '; ;
d := ' ' '; ;
e :=′ lnlnoocppnoodpnnooepsnooipunoou'; ;
```

$i :='$ loop e case $l \rightarrow [t] := [t][a]$,
$n \rightarrow [t] := [t][d]$,
$o \rightarrow [t] := [t];$,
$c \rightarrow [t] := [t]c$,
$p \rightarrow [t] := [t][c]$,
$d \rightarrow [t] := [t]d$,
$e \rightarrow [t] := [t]e$,
$s \rightarrow [t] := [t][e]$,
$i \rightarrow [t] := [t]i$,
$u \rightarrow [t] := [t][i]$,
end; output t'; ;
[i]

The external loop instruction now processes the running variable e. This variable contains the stringwise coding of π^2_{rep}. The coding process is performed through the recording of the list of `Alternatives`. The external loop instruction self-decodes $[e]$ and generates successively π^2_{rep}. Only symbols from $A_{\min}$ can appear in π^2_{rep}. Hence $\pi^2_{\text{rep}} \in \overline{LP(A)}$ for every $A_{\min} \subset A$, provided that $\overline{LP(A)}$ satisfies Postulate 6.2.

Since the embedding (interlacing) depth of the loop instructions in π^2_{rep} is equal to 2, we finally get the proposition. □

From the program π^2_{rep} in the previous proof, we can derive a program π^1_{rep} with a loop embedding (interlacing) depth of only 1. We build π^1_{rep} from π^2_{rep} by eliminating the external loop instruction. The latter loop processes the variable e whose content is 31 symbols long. We now list the instruction part of the list of `Alternatives` for every one of those 31 symbols. Since e contains its own coding and since e will no longer be useful once the external loop has been eliminated, the number of instruction in this part is reduced to 25. Given now a total of 25 symbols and 10 alternatives, it is clear that a few alternatives must be written several times. It will be also clear that the external loop in π^2_{rep} is used only for code reduction purposes.

$\pi^1_{\text{rep}} =$ input;
$a :='$ input;$a :=';$;
$c :=':=$ ' ';;
$d :=$ ' ' ';;
$i :=' <$ alternative for l $>$;
$<$ alternative for n $>$;
$<$ alternative for l $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for c $>$;
$<$ alternative for p $>$;
$<$ alternative for p $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for d $>$;
$<$ alternative for p $>$;
$<$ alternative for n $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for i $>$;
$<$ alternative for p $>$;
$<$ alternative for u $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for u $>$;
output t'; ;
$<$ alternative for l $>$;
$<$ alternative for n $>$;
$<$ alternative for l $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for c $>$;
$<$ alternative for p $>$;
$<$ alternative for p $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for d $>$;
$<$ alternative for p $>$;
$<$ alternative for n $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for i $>$;
$<$ alternative for p $>$;
$<$ alternative for u $>$;
$<$ alternative for n $>$;
$<$ alternative for o $>$;
$<$ alternative for o $>$;
$<$ alternative for u $>$;
output t

The parts between square brackets must be replaced as follows:

- < alternative for l > is replaced by

$$\begin{aligned}\text{loop } a \text{ case } i &\to t := t|i,\\ n &\to t := t|n,\\ p &\to t := t|p,\\ u &\to t := t|u,\\ t &\to t := t|t,\\ ; &\to t := t|;,\\ a &\to t := t|a,\\ : &\to t := t|:,\\ = &\to t := t|=,\\ &\text{end}\end{aligned}$$

- < alternative for n > is replaced by

$$\begin{aligned}\text{loop } d \text{ case } ' &\to t := t|',\\ &\text{end}\end{aligned}$$

- < alternative for o > is replaced by $t := t|;$.
- < alternative for c > is replaced by $t := t|c$.
- < alternative for p > is replaced by

$$\begin{aligned}\text{loop } c \text{ case } : &\to t := t|:,\\ = &\to t := t|=,\\ ' &\to t := t|',\\ &\text{end}\end{aligned}$$

- < alternative for d > is replaced by $t := t|d$.
- < alternative for i > is replaced by $t := t|i$.
- < alternative for u > is replaced by

$$\begin{aligned}\text{loop } i \text{ case } a &\to t := t|a,\\ c &\to t := t|c,\\ d &\to t := t|d,\\ e &\to t := t|e,\\ i &\to t := t|i,\\ l &\to t := t|l,\\ n &\to t := t|n,\\ o &\to t := t|o,\\ p &\to t := t|p,\\ s &\to t := t|s,\\ t &\to t := t|t,\\ u &\to t := t|u,\\ : &\to t := t|:,\\ = &\to t := t|=,\\ ' &\to t := t|',\\ ; &\to t := t|;,\\ , &\to t := t|,,\\ \to &\to t := t|\to,\\ &< \text{space} > \to t := t| < \text{space} >,\\ &\text{end}\end{aligned}$$

Obviously we have: π^1_{rep} is self-reproducing. Since $\pi^1_{\text{rep}} \in A^*_{\text{min}}$, we can give the following proposition.

Proposition 6.2 *Let A be a finite alphabet with $A_{min} \subset A$. Then there exists a self-reproducing program in $\overline{LP_1(A)}$, provided that Postulate 6.2 is satisfied.*

From the relatively simple construction of π^1_{rep} from π^2_{rep}, we can now try to derive a program π^0_{rep} from π^1_{rep} which no longer uses loop instructions. To this end, we must in π^1_{rep} eliminate the still remaining loop instructions

< alternative for l >;

< alternative for n >;

< alternative for p >;

< alternative for u >;

The first of those loop instructions can easily be divided into a sequence of base instructions, since its purpose is code reduction, similar to the external loop instruction in π^2_{rep}. Difficulties arise with the < alternative for n >; instruction. It can be written without any loop instruction as follows:

$t := t|'$

Since a semi-colon follows < alternative for n >, the constant text of i (and hence the forbidden text combination $';$) will contain the partial string $t := t|';$ (compare with Sect. 6.1). We are thus unable to replace the < alternative for n >; instruction with a sequential program part. However, this is only due to the definition of the text constants in $\overline{LP(A)}$ programs given in 6.4. We likely can avoid this situation by considering an alternative definition. The same reasoning applies to < alternative for p >, as well.

The situation is different, though, when it come to < alternative for u >. This loop instruction is used to add the content of variable i to the content of variable t. Now < alternative for u > is itself a textual part of the content of the variable i:

$i :=' \cdots <$ alternative for u $> \cdots '$

We cannot replace < alternative for u > with a sequence of base instructions. No self-reproducing program without any loop instruction can be obtained from π^1_{rep}. Thus we have:

Proposition 6.3 *For every finite alphabet A, no self-reproducing programs in $\overline{LP_0(A)}$ exists.*

Proof Let $\overline{LP(A)}$ be defined on a finite alphabet A. Let us suppose that Postulate 6.2 is satisfied. Let us suppose that there exists a self-reproducing program $\pi_0 \in \overline{LP_0(A)}$ (*ab absurdo* proof). Then π_0 has the following structure:

$\pi^0 =$ input; AW_{π^0}; output t

In AW_{π^0}, we have to build the code of π^0. Since no loop instruction is available, only two possible cases are able to generate the code of π^0 in AW_{π^0}.

Case 1. The code of π^0 will be generated block by block through the help of base instructions of type γ_6. Then we have the following equation for π^0:

$\pi^0 = \text{input}; t := `\pi^0'; \text{output}$

This equation can however not be realized from any finite code π^0. Hence we have a contradiction.

Case 2. The code π^0 is build one symbol at a time in AW_{π^0}, by means of instructions of type γ_3'. Since π^0 must be different from the empty word, the instruction $t := t|x;$, with $x \in A$, must appear at least once in AW_{π^0}. This instruction contains 7 symbols. Interpreted as a string, it represents a constitutive part of π^0. The instruction can attached at most one of its 7 symbols to the output variable. Hence, at least 6 symbols will remain unprocessed. For those 6 symbols, 6 additional instructions of type γ_3' are necessary. Those 6 instructions themselves produce 36 unused symbols which remain unprocessed, and so on…

In order to generate a code of length $k \geq 0$ with instructions of type γ_3', we need a program of length at least equal to $7k$. Hence there is no finite program of length k which can generate a finite code of length k from the empty word just with instructions of type γ_3'; a fortiori not its own code. Consequently π^0 is itself not finite and hence it is not a program. Again, there is a contradiction. □

Remark 6.2 In Chap. 2, the existence of self-reproducing PL(A)-programs was proved (Proposition 2.6). Besides the Recursion Theorem (Proposition 2.5), the proof rested on the existence of a universal function for the function class $\mathcal{P}_1^1$.

The class of all word functions

$$\varphi : (A^*)^r \to (A^*)^s, \qquad r, s \geq 0,$$

which can be computed by means of $\overline{LP(A)}$ programs, is a function class which contains total functions only; $\overline{LP(A)}$ programs always halt. In [5, p. 47], it has been shown that no universal $\overline{LP(A)}$ computable function can exist for $\overline{LP(A)}$ computable functions. However there exist self-reproducing programs in $\overline{LP(A_{\min})}$. Let us note, then, that universality is not a necessary condition for self-reproduction. There are other, more direct way, as in Chap. 2, to theoretically prove the existence of self-reproducing programs.

6.6 Self-reproduction principle of $LP(A)$ programs

In Chap. 5, Propositions 5.1 and 5.2 proved the existence of self-reproducing versions for any given SIMULA (respectively PASCAL) programs. A similar proposition can be proved for $\overline{LP(A)}$ programs. By replacing once more the terms "*input file*" and "*output file*" introduced in Sect. 5.1 with "*input variable*" (respectively "*output variable*"), the self-reproducing version of $\overline{LP(A)}$ programs is made more obvious.

Let A be any finite alphabet and $\pi \in \overline{LP(A)}$. No self-reproducing version $\tilde{\pi}$ of π may exist in $\overline{LP(A)}$ (maybe because the variable names used in $\overline{LP(A)}$ do not belong to A^*), but only in the language $\overline{LP(B)}$ with a corresponding larger alphabet $B \supset A$. From Definition 5.2, such a program $\tilde{\pi} \in \overline{LP(B)}$ would not be a self-reproducing version of π, since it works on a different data domain. Definition 5.2 is satisfied, however, by considering π also a program from $\overline{LP(B)}$, which is reasonable since $A \subset B$: The function realized by $\pi \in \overline{LP(A)}$ is equivalent to the restriction of the function on $(A^*)^r, \quad r \geq 0$ realized by π as $\overline{LP(B)}$ programs (compare with Definition 5.1). This view corresponds to Proposition 6.4.

Proposition 6.4 (*Self-reproduction of* $\overline{LP(A)}$ *programs*) *Let A be a finite alphabet and $\pi \in \overline{LP(A)}$. Then there exists a finite alphabet B, such that $A \subset B$ and such that*

1. *there exists a self-reproducing version $\tilde{\pi} \in \overline{LP(B)}$ of π (where π is considered an element of $\overline{LP(B)}$).*
2. *and*

$$\tilde{\pi} \in \begin{cases} \overline{LP_2(B)} & \text{if } \pi \in \overline{LP_j(A)} \quad \text{for } j = 0, 1 \\ \overline{LP_i(B)} & \text{if } \pi \in \overline{LP_i(A)} \quad \text{for } i \geq 2. \end{cases}$$

Proof Let A be any finite alphabet and $\pi \in \overline{LP(A)}$. Without loss of generality, let all variables in π not be in $\{c, d, e, i, t\}$.

Construction of the self-reproducing version $\tilde{\pi}$

π has the following structure:

$$\begin{aligned} \pi = {} & \text{input } y_1, \ldots, y_r; \\ & AW_\pi; \\ & \text{output } z_1, \ldots, z_w; \qquad r, w \geq 0 \end{aligned}$$

Let A_π be the set of all symbols from which the program π has been assembled. Let the string $S \in A_\pi^*$ be defined as follows:

$$S := \text{input } y_1, \ldots, y_r; AW_\pi;$$

(where AW_π of course describes the code of the instruction part of π). We thus have

$$\pi = S \circ \text{output } z_1, \ldots, z_w.$$

S will be divided into a sequence of $n \geq 1$ partial strings s_i with

1. $s_i ='$; or $'$; is not contained in s_i,

2. s_i does not contain $'$; $\Rightarrow s_{i+1} ='$; $\quad i < n$,
3. $s_1 \circ s_2 \circ \ldots \circ s_n = S$.

Let us recall that $'$; is used as an end marker for text (code) constants and must not itself be a partial string of a text constant.

Let us consider additional settings:

- $\mathcal{S} := \{s_1, \ldots, s_n\}$ is the set of partial strings.
- Let q be the number of partial strings S which are different from $'$;. We have $1 \leq q \leq n$.
- Let $x_1, \ldots, x_{2q}$ be symbols which are neither in $\{c, d, e, i, t\}$ nor appear as variables in π.
- $B := A \cup A_\pi \cup A_{\min} \cup \{x_1, \ldots, x_{2q}\}$. B is a finite alphabet and can thus be formally denoted as $B := \{b_1, \ldots, b_\alpha\}$, where α is the cardinality of B.
- Definition of functions $\mathcal{G}$ and $\tilde{\mathcal{G}}$:

$$\begin{aligned} &\mathcal{G} : [q] \to \mathcal{S} \\ &\quad i \mapsto s_j \end{aligned}$$

where s_j is the i-th partial string which is different from $'$;,

$$\begin{aligned} &\tilde{\mathcal{G}} :\to B^* \\ &i \mapsto \begin{cases} \text{no} & \text{if } s_i ='; \\ x_j & \text{if } s_j \text{ is the i-th partial string different from '}; \end{cases} \end{aligned}$$

- Let v be an additional variable and let us abbreviate the instruction

$$\begin{aligned} &\text{loop } v \text{ case } b_1 \to t := t|b_1, \\ &\qquad\qquad \ldots \qquad\quad \ldots \\ &\qquad\qquad b_\alpha \to t := t|b_\alpha, \\ &\qquad \text{end} \end{aligned}$$

by the reduced instruction loop v.

Given the additional notation and settings, the program $\tilde{\pi} \in \overline{LP(B)}$ can now be given:

$$\begin{aligned} &\tilde{\pi} = \\ &s_1 \ldots s_n = \text{input } y_1, \ldots, y_n; AW_\pi; \\ &x_1 :=' \mathcal{G}(1)'; ; \\ &x_2 :=' \mathcal{G}(2)'; ; \\ &\qquad \ldots \qquad\qquad \ldots \qquad\qquad \ldots \\ &x_q :=' \mathcal{G}(q)'; ; \\ &c := \text{' } := \text{' '};; \\ &d := \text{' ' '};; \\ &e :=' \tilde{\mathcal{G}}(1) \ldots \tilde{\mathcal{G}}(n) \\ &\quad x_1 p x_{q+1} noo \\ &\qquad \ldots \\ &\quad x_q p x_{q+q} noo \\ &\quad \text{cppnoodpnnooepsnooipunoou'}; ; \\ &i :=' \text{ loop } e \text{ case} \quad x_1 \to t := t|x_1, \\ &\qquad\qquad\qquad\qquad\quad x_2 \to t := t|x_2, \\ &\qquad\qquad\qquad\qquad\qquad \ldots \\ &\qquad\qquad\qquad\qquad\quad x_q \to t := t|x_q, \\ &\qquad\qquad\qquad\qquad\quad c \to t := t|c, \\ &\qquad\qquad\qquad\qquad\quad d \to t := t|d, \\ &\qquad\qquad\qquad\qquad\quad e \to t := t|e, \\ &\qquad\qquad\qquad\qquad\quad i \to t := t|i, \\ &\qquad\qquad\qquad\qquad\quad x_{q+1} \to \text{ loop } x_1, \\ &\qquad\qquad\qquad\qquad\quad x_{q+2} \to \text{ loop } x_2, \\ &\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ &\qquad\qquad\qquad\qquad\quad x_{q+q} \to \text{ loop } x_q, \\ &\qquad\qquad\qquad\qquad\quad n \to \text{loop } d \text{ case} \qquad ' \to t := t|', \\ &\qquad\qquad\qquad\qquad\qquad\quad \text{end}, \\ &\qquad\qquad\qquad\qquad\quad o \to t := t|; , \\ &\qquad\qquad\qquad\qquad\quad p \to \text{loop } c \text{ case} \qquad :' \to t := t| :, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad =\to t := t| =, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ' \to t := t|', \\ &\qquad\qquad\qquad\qquad\qquad\quad \text{end}, \\ &\qquad\qquad\qquad\qquad\quad s \to \text{ loop } e, \\ &\qquad\qquad\qquad\qquad\quad u \to \text{ loop } i, \\ &\text{end}; \qquad\qquad\qquad\qquad\quad \text{output } z_1, \ldots z_w, t'; ; \\ &\text{loop } e \text{ case} \qquad\quad x_1 \to t := t|x_1, \\ &\qquad\qquad\qquad\qquad\quad x_2 \to t := t|x_2, \\ &\qquad\qquad\qquad\qquad\quad \ldots \\ &\qquad\qquad\qquad\qquad\quad x_q \to t := t|x_q, \\ &\qquad\qquad\qquad\qquad\quad c \to t := t|c, \\ &\qquad\qquad\qquad\qquad\quad d \to t := t|d, \\ &\qquad\qquad\qquad\qquad\quad e \to t := t|e, \\ &\qquad\qquad\qquad\qquad\quad i \to t := t|i, \\ &\qquad\qquad\qquad\qquad\quad x_{q+1} \to \text{ loop } x_1, \\ &\qquad\qquad\qquad\qquad\quad x_{q+2} \to \text{ loop } x_2, \\ &\qquad\qquad\qquad\qquad\quad \ldots \\ &\qquad\qquad\qquad\qquad\quad x_{q+q} \to \text{ loop } x_q, \\ &\qquad\qquad\qquad\qquad\quad n \to \text{ loop } d \text{ case} \qquad ' \to t := t|', \\ &\qquad\qquad\qquad\qquad\quad \text{end}, \\ &\qquad\qquad\qquad\qquad\quad o \to t := t|; , \\ &\qquad\qquad\qquad\qquad\quad p \to \text{ loop } c \text{ case} \qquad :\to t := t| :, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad =\to t := t| =, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ' \to t := t|', \\ &\qquad\qquad\qquad\qquad\quad \text{end} , \\ &\qquad\qquad\qquad\qquad\quad s \to \text{loop } e, \\ &\qquad\qquad\qquad\qquad\quad u \to \text{loop } i, \\ &\text{end}; \\ &\text{output } z_1, \ldots, z_w, t \end{aligned}$$

Claim: $\tilde{\pi}$ is a self-reproducing version of π.

The program $\tilde{\pi}$ begins with the input of the input variables of π and the processing of the whole instruction part

of program π. At the end of $\tilde{\pi}$, the output variables of π will be produced. These output variables $z_1, \ldots, z_w$ will not be modified by instructions beyond the instruction part of π. Program $\tilde{\pi}$ additionally produces the variable t. To prove that $\tilde{\pi}$ is a self-reproducing version of π, it is sufficient to show that at the end of the program execution, the content of t is equal to $\tilde{\pi}$.

Variables $x_1, \ldots, x_q, c, d, e$ and i contain—except for a few single symbols—the code of program $\tilde{\pi}$ under a split form. The purpose of $\tilde{\pi}$'s loop instruction is to assemble the partial strings stored in those variables back into $\tilde{\pi}$ code. The external loop instruction

loop e case ... end;

will be driven by variable e, which contains the code of $\tilde{\pi}$ in an encoded form. This coding is straightforward by considering the list of `Alternatives`. For every symbol of the code constant e, exactly one partial string of program $\tilde{\pi}$ will be added to the content of variable t. The constant code e is divided into three parts:

Part I: $\tilde{\mathcal{G}}(1) \ldots \tilde{\mathcal{G}}(n)$
Part II:

$$x_1 p x_{q+1} noo$$
$$\ldots$$
$$x_q p x_{q+q} noo$$

Part III: $cppnoodpnnooepsnooipunoou.$

Part I results in the fact that

input $y_1, \ldots, y_r$; AW_π;

is added to the content of the initially empty variable t.

Part II extends the content of t by the program lines

$x_1 := '\mathcal{G}(1)'; ; to$
$x_q := '\mathcal{G}(q)'; ;$

Part III causes the remaining program code of $\tilde{\pi}$ to be added to the content of t. This follows, wholly analogous to π^2_{rep}, from the proof of Proposition 6.1.

Part III processes the variable e completely, and the execution of the external loop instruction stops. Consequently, the whole program stops, as well, and $\tilde{\pi}$ will output the content of t. $\tilde{\pi}$ satisfies the Definition 5.2(2) and therefore is the self-reproducing version of π.

The "reproducing" loop instruction in $\tilde{\pi}$ is of depth 2 and is located near the instruction part of π. Thus, the depth of loop instruction of $\tilde{\pi}$ is at least 2, but constrained by depth of the loop instruction of π. Consequently, the second claim of the proposition is satisfied. □

Remark 6.3 I. From the proof of Proposition 6.4 we can directly derive an algorithm to search for a self-reproducing version for a given program in $\overline{LP(A)}$. This algorithm can, however, be vastly improved. As examples:

- We can choose alphabet B more judiciously. A proof could be given with a "smaller" alphabet B then used in the given proof.
- The loop instruction of type

  loop v

  contains many unused `Alternatives`, which where included in the proof for the sake of a uniform presentation.

II. The construction given in proof of Proposition 6.4 closely follows program π^2_{rep} from Sect. 6.5. A construction by means of the $\overline{LP_1(A_{\min})}$ program π^1_{rep} would result in self-reproducing versions $\tilde{\pi}$ with

$$\tilde{\pi} \in \begin{cases} \overline{LP_1(B)}, & \text{if } \pi \in \overline{LP_0(A)} \\ \overline{LP_i(B)}, & \text{if } \pi \in \overline{LP_i(A)}, \quad i \geq 1. \end{cases}$$

This construction, though, would be even more difficult to grok than the construction by means of π^2_{rep}.

7 Living programs?

7.1 Introduction

The preceding sections illustrated and implemented self-reproducing programs in higher-level programming languages, as well as lower-level assembly languages. Chapter 5 in particular showed that not only do an infinite number of self-reproducing programs exist, but that any programming task can be handled by a self-reproducing program within the physical limitations of concrete computing systems. Systems do not exhibit 100% fidelity; small, but non-zero error rates for switching and transmission errors remain an (unlikely) possibility.

Example 7.1 For the sake of efficiency and higher hardware utilization rates, computing systems are clustered into larger networks [21]. It is not unheard of the distance between individual computing components to exceed hundreds of kilometers; as such, vexing data transmission problems pop up and have to be tackled. Depending on the physical medium, transmission error rates range from 10^{-4} to 10^{-7} bits/sec, with an error rate of 10^{-4} bits/sec for POTS [12,21]. Through error handling such as error correction codes [13] and appropriate communication protocols [21] the rate can be lowered: The

US ARPA network exhibits a bit transmission error rate of 10^{-12} bits/sec [12].

For instance, faulty self-reproduction of program π is possible during the memory copying process, resulting in reproducing a different program $\pi' \neq \pi$. If syntactically correct, π' could amount to another self-reproducing program which realizes a different function than π. We can detect a strong analogy to the reproduction and mutation processes of living cells, two fundamental biological characteristics for life. Are there additional traits attributable to programs that are characteristic of life? In our quest for answers to this question, there are a couple of problems, some of them serious, that have to be addressed.

- There are no universally agreed-upon definitions of, or characteristics for, life in modern biology (see Sect. 7.2).
- Biological life is based on a complex interplay of biochemical reactions. Key roles are played by certain macromolecules, chiefly among them nucleic- and amino acids [11], the interaction of which constitute the basis of all earthly life. Biology hinges the answer to the question of alternative life forms on replacing nucleic and amino acids with functional equivalent macro-molecules [11]. This, in turn, presupposes a *sine qua non* chemical basis for life, which would rule out living computer programs.

Thus, the quest for 'living' programs is surely beset with philosophical and theoretical biology difficulties. It is not the purpose of this and the two subsequent sections to define 'living' programs; we understand it as a first stab, an attempt to grapple with and discuss the aforementioned problems.

7.2 Biological life

Modern biology still has not come up with a uniform definition for life. By studying existing and extinct life forms, it is possible to extract a couple of communalities of all life. We list

- Metabolism and metabolic regulation,
- Cell reproduction and mutation,

as the key life processes. These processes are responsible for sustaining the individual entities, reproduction and genetic changes (see [11, p. 24ff]). Additional views include irritability and mobility as distinctive aspects of life (see [15, p. 335]).

Metabolism

The cell is the fundamental building 'block' of life - all life. Cells as well as living beings constitute finite material systems. A cell constantly absorbs matter from its surroundings, internally alters it, and expels it into the environment in changed form. Since matter is constantly flowing through cells, we denote the 'cell' system as a "thermodynamic flow system" [11], more precisely an open thermodynamic system [2]. Metabolic catalysis follows known, ordered pathways and the system settles into a flow equilibrium; some researchers maintain that all communalities of living organisms directly follow from this tendency towards equilibrium [2, p. 158]. Open thermodynamic systems in flow equilibrium move towards a constant state, independent of initial starting conditions. This state is denoted as a stationary state. Materially similar flow systems in an identical environment strive to reach the same stationary state, regardless of initial conditions. This hints at the existence of a metabolic control system for living organisms.

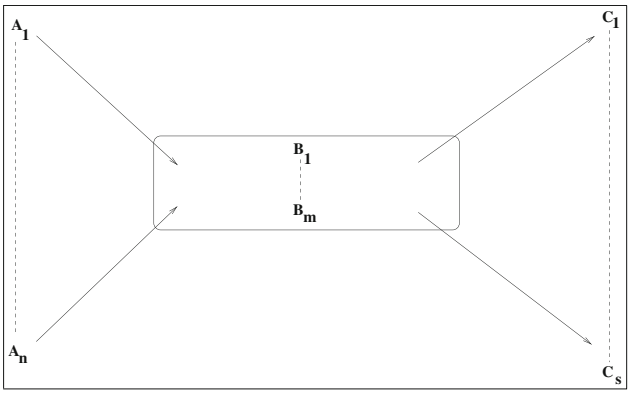


Fig. 4 A cell: Sketch of a chemical thermodynamic flow system

Figure 4 sketches a chemical flow system in which matter A_i flow into the system, are transformed within to matter B_i with the (junk) by-products C_i expelled into the environment. Metabolism is often purely heuristically divided into catabolism, in which complex molecules are broken to yield energy, and anabolism, in which energy is used to construct complex molecules.

Reproduction and Mutation

Living organisms reproduce by cell division in which the daughter cells inherit the same cell structure and the same metabolic flow scheme (from one parent cell in mitotic, from both in meiotic reproduction). Again, these materially similar flow systems (the daughter cells) will strive to reach the same stationary state as the parent cells, regardless of initial conditions, provided the environment does not change after cell division. Put another way, the daughter cells "inherit" the state of the parent cells. Cell structure and concomitant metabolic reactions are governed by proteins; thus, for a daughter cell to reach the same stationary state as her parents (given

identical environments), two conditions have to be met during cell reproduction:

- the daughter cells must share the protein makeup of the parents,
- protein formation must follow the same time ordering.

A cell contains the necessary information to perform protein synthesis in the form of specially structured nucleic acid molecules. These molecules are collectively denoted as DNA, locatable molecular section as genes, respectively [27]. In order for the daughter cells to be able to synthesize the same proteins, an identical copy of the parent cell DNA is passed on; in the case of meiotic reproduction, it is a combination of the DNA of both parents (gene alleles). Hence, the progeny receives the protein synthesis blueprint from their parent(s). If errors occur during the DNA replication phase and a faithful copy is not passed on, the daughter cell will synthesize a different set of proteins, perform a different set internal B_i reactions and thus reach a different flow equilibrium than exhibited by the parents. The stationary state will be different, as well. Such punctuated changes in the hereditary information (genome) of living organism are called mutations. Mutations are random and 'blind', and may also arise spontaneously through DNA changes in 'ready-made' cells, not just during erroneous copying processes. This extremely short and incomplete exegesis on metabolism, reproduction and mutation of living beings should serve as a basis for the following considerations.

7.3 Self-reproducing programs and life

In contrast to natural life forms programs are information-, not matter-based. To access said information, it has to be presented in an interpretable format. Examples are:

- punch cards
- punch rolls
- formulations through "Paper and pencil"

⋮

Programs are usually written for execution on a concrete computing system. Within the system, programs are digitally represented on and interpretable by the system given syntactic and semantic[10] correctness. Here, we have to stress that program existence is not tied to its representation. Because of their more abstract, less material nature, programs do not have metabolic pre-conditions to sustain their existence, either. Although energy is required to execute program π on a computing systems, this cannot be classified as catabolism since the energy supply

- is not actively controlled by program π,
- is used for the interpretation, not sustainment, of program π.

Program are written to some sort of storage medium on the computing systems. Certain storage types like semiconducting charged-coupled devices need (time) periodic content refreshing [13]. The availability of this memory is indeed contingent on a regular consistent supply of energy; however, it remains for similar reasons as mentioned above too much of a stretch to categorize this process as catabolism. Hence, trying to identify an exact analogy to biological metabolism in the context of programs may be somewhat futile. Reproduction and mutation are a different matter; numerous examples in the preceding sections illustrate the point that programs can produce identical offspring. One component that all examples in higher level language had in common was that they contained somewhere in the program text their own blueprint in encoded form (e.g. array `C[23]` contains π_3 in Sect. 3.2.5; the text constant of procedure `AB` contains π_4 in Sect. 3.2.7). Here, the analogy to the DNA blueprint of living cells is apt. The assembly of self-reproducing program copies through a blueprint is in principle, albeit tenuously, comparable to the protein synthesis of cells. A subtle point has to be emphasized and kept in mind, though: Program reproduction, strictly speaking, differs from the auto-reproduction of living organisms in that the former requires an external stimulus (control through the operating system) and does not rely on an internal mechanism to induce the stimulus. Again, small, but non-zero rates for switching and transmission errors introduce errors in the process of program reproduction (see Sect. 7.1). As such, mutations are possible.

All in all, after examining biological life's two key processes, we merely find a correspondence to one of them, reproduction and mutation, in the context of self-reproducing programs. We therefore cannot state that self-reproducing programs are alive, and thus, they cannot be compared exactly to living organisms. Certain biological structures, however, may still be suitably compared to self-reproducing programs.

7.4 Self-reproducing programs and viruses

For a long time, viruses were considered the simplest organisms, far simpler in their makeup even than one-celled living beings. However, viruses are incomplete organisms - subcellular structures really - consisting almost exclusively of DNA. In addition, some viruses embed their DNA into a coating of proteins, lipids and organic substances. They have

[10] In this context, no run-time errors.

no metabolism; they have to penetrate living cells in order to show signs of life like mutation or reproduction. Viruses have to leverage the metabolism of real organisms in order to reproduce. Otherwise, viruses are dead and may even be arranged in crystal form, a spatial arrangement that is not known of living beings. Hence, of the two key processes, virus' exhibit only one (reproduction and mutation), and even that one only on condition that a foreign metabolic entity provides them with the requisite cata- and anabolic mechanisms. Here, we can point out similarities to self-reproducing programs: As long as a self-reproducing program is not written to a computing system's memory, the program is of no consequence (besides its inherent information content). Only when the program finds itself in memory, and only when it is actually executed, may a self-reproducing program actually reproduce and mutate. The program draws on energy supplied by the system. We stress that the analogy cannot be stretched too far: a biological virus may induce its own reproduction by actively intruding into a cell and leveraging its metabolic processes. A self-reproducing program is incapable of such a feat, even residing on a system and drawing on said computing "system"'s memory and energy, it remains dependent on activation through the operating system.

8 Models for competing self-reproducing programs

8.1 Motivation

Multilayered resource jockeying is a fact of biologic life, not just at the individual, but at the species level (e.g. population [25, p. 337]). Competitive success hinges in no small part on the variability of the species' genome. Species have to constantly adapt to an inanimate environment, as well as assert themselves against other species. Hence, features of species and individuals cannot be divorced from their relationship to the animate and inanimate environment (see [27, p. 199ff]). Similarly, self-reproducing programs are embedded in a computing system "environment" consisting of hardware and system software. Even the storage medium in which the program resides is part of this environment. To stretch the analogy further, other self-reproducing programs in memory can be viewed as part the animate environment. Therefore, it cannot be ruled out that interactions and feedback effects between various self-reproducing programs on the one hand, and self-reproducing programs and the computing system on the other hand could give rise to new and different self-reproducing programs with ever changing features. We propose following (speculative!) models to investigate the behaviorial patterns that may be induced by feedback effects between self-reproducing programs.

8.2 A basic model

We assume that the proposed base model MOD1 will be executed on a conventional computing system based on a "von Neumann architecture" [12]. Such an architecture will typically feature one central processing unit, and one or more Input-Output (IO) channels (in the extreme case, several IO CPUs). Limited multiprogramming is possible in that k programs $\pi_1, \ldots, \pi_k$ can be run quasi-simultaneously via temporal switching, but not strictly processed in parallel. The programs are thus alternately executed in time slices, or put another way, they are time shared [1]. Self-reproducing programs in MOD1 are characterized by two metrics:

- Runtime (i.e. their reproductive time cycle).
- Spatial (i.e. memory position) relationship between a program and its copy.

8.2.1 An informal description of MOD1

(i) Programs: Programs are identified by name, which renders the program structure opaque. More important data characterizations, however, include
   a) The number of time ticks necessary for the program to reproduce.
   b) The minimum memory distance between the copy and the inducing program (see (ii)).

(ii) Memory: Memory is one-dimensional, infinite and discretized into memory cells. Two adjacent cells are directly connected. Each cell is able to hold exactly one program, independent of its physical length. The memory vector is sparse, almost all cells are empty.

(iii) Temporal process: Memory is initialized with NUM self-reproducing programs $\pi_1, \ldots, \pi_{NUM}$. Each program π_j is given periodically a time tick for execution. The cyclical activation is possible given a finite number of programs in memory, though the cycle will grow with the number of reproducing programs. After an individual, finite number of time ticks in which control is passed to it, every program residing in memory is able to reproduce.

(iv) Spatial process: Copies are written to preceding or subsequent memory cells, subject to a minimum distance requirement. Should the selected cell be occupied, immediately adjacent cells are probed consecutively until a free cell is found. It follows from (ii) that this will eventually happen. Hence, this setup precludes memory exhaustion problems.

MOD1 avoids collisions in that programs cannot infringe upon one another. Specifically, 'privileged' programs can-

Table 35 MOD1: SIMULA implementation

```
1) class PROGRAM;
2) begin
3) integer DELY, DISTANCE, IDENT;
4) end;
```

Table 36 MOD1: CELL implementation

```
1) class CELL;
2) begin
3) ref (PROGRAM) CONTENS;
4) ref (CELL) LEFT, RIGHT;
5) integer TIMECOUNT;
6) end;
```

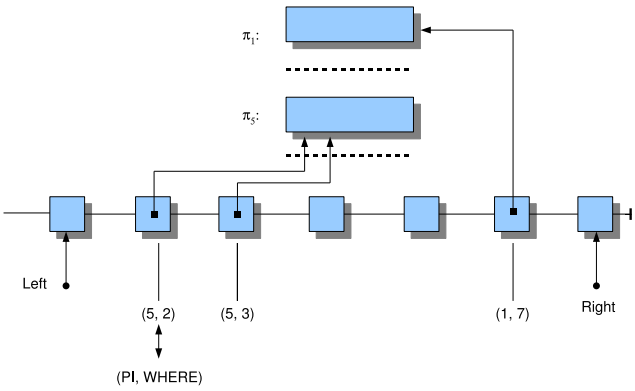


Fig. 5 MOD1: Global data structures

not destroy other programs by overwriting occupied memory cells; in this sense, all programs are equivalent. Since collision avoidance is supported by the infinite memory framework, we can view every programs (individual) as permanent, and permanently replicating (progeny). The number of residing programs (population) grows and grows. Figuratively speaking, the MOD1 universe models peaceful coexistence rather than a "fight for survival". Such a model rules out evolution, since the driving forces of evolution, mutation and selection, cannot assert themselves for lack of selection pressure.

8.2.2 MOD1 implemented in SIMULA

(i) Programs: Incorporating the parameter triples into a SIMULA program. `DELY`, `DISTANCE`, `IDENT` denote reproduction time, minimum distance of copy, and program identification, respectively (Table 35).

(ii) A memory cell can hold an object of type `PROGRAM` and is flanked by two neighboring cells. We model this structure with the SIMULA concept `CELL` (Table 36). Memory is modeled as a doubly linked list ([28] p. 233ff). At first, we initialize a fixed memory block of length N, though we can add additional cells if needed, making memory potentially infinite. The variables `LEFT` and `RIGHT` point to the left and right end of the list, respectively. The procedures

```
ref (CELL) procedure ADDLEFT;
```

and

```
ref (CELL) procedure ADDRIGHT;
```

are used to append a memory cell to the list.

Example 8.1

The procedure `ADDRIGHT` works analogously.

(iii) Temporal behavior: Start time: At simulation start, memory is initialized with a fixed number of programs (objects of type `PROGRAM` $\pi_i, i = 1, \ldots, NUM$. There are $M \leq NUM$ different programs, meaning that initially, there are multiple copies of some programs residing in memory. Since the cells hold references and not the programs π_j themselves, the actual code has to be stored *in toto* somewhere. This is the purpose of `ref (PROGRAM) array P[1:M];`. The program π_j can be de-referenced by pointer `P[j]`. Copying π_j requires merely setting another pointer to π_j. Effectively, π_j can be viewed as a "program type", and pointers thereto as concrete instantiations or examples of this type. The terminology is a bit vague in this instance, but we will continue to refer to π_j both as program, as well as program type, depending on the context. The field `integer array ST[1:M]` keeps track of count `ST[j]` of program `P[j]`. We have initially

$$\sum_{j=1}^{M} \texttt{ST[j]} = \texttt{NUM}$$

The tuples `(PI, WHERE)` determine the memory position of the NUM programs such that program `P[PI]` is written to the `WHERE`$^{\text{th}}$ memory cell to the right of pointer `LEFT` (Fig. 5; Table 37).

Simulation

Pointer `C` of type `ref (CELL)` iterates through memory from left to right `TIME` times, testing each cell for content. If the cell is empty, nothing happens; if the cell contains a program, the condition

```
TIMECOUNT+1 = CONTENS.DELY
```

Table 37 CELL: adding to the list

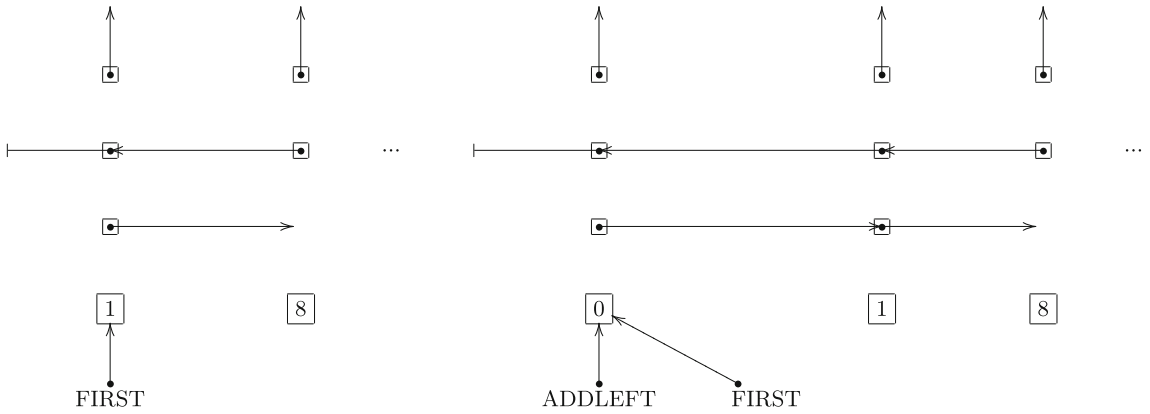


Table 38 MOD1: Initialization and copying

```
1)  for   T:=1   step 1 until TIME do
2)  begin
3)        C := FIRST;
4)        OLD_LAST := LAST;
5)        while C≠ OLD_LAST.RIGHT do
6)        begin
7)              [increase C.TIMECOUNT by 1];
8)              if C.TIMECOUNT = C.CONTENS.DELY
9)              then [Copy program C.CONTENS];
10)                  [Reset C.TIMECOUNT to 0];
11)             C:=C.RIGHT;
12)       end
13) end
```

is evaluated. Should the test yield true, a copy of the program is made and TIMECOUNT is reset to zero, indicating that a new reproductive cycle can start. If the condition fails, TIMECOUNT is increased by 1. Sometime, a new memory cell has to be created at the rightmost end of the memory vector. Care must be taken that the current iteration does not process the newly created cell; only in the iteration in the subsequent cycle will access the cell. Support for this notion is given by variable

```
ref (CELL) OLD_LAST
```

and an implementation sketch is given in Table 38 (Lines 3 and 4 initialize one cycle).

(iv) Spatial behaviour:
In the ith, $i \leq$ TIME, iteration, let pointer C point to a cell containing program $\pi_j =$ C.CONTENS waiting to reproduce, i.e.

```
C.TIMECOUNT+1 = CONTENS.DELY
```

Table 39 MOD1: COPY and RANDINT

```
1) procedure COPY(C); ref (CELL) C;
2) if RANDINT(1,2,U) = 1
3) then COPY_LEFT(C);
4) else COPY_RIGHT(C);
```

Program π_j is copied either to the left or right of the cell pointed to by C, into the location determined by procedure COPY with the help of the SIMULA random number function RANDINT.[11] Table 39 shows that depending on the random number, either COPY_LEFT or COPY_RIGHT is called to perform the actual copying. Appendix C.1 in ESM presents MOD1 in its entirety in the form of an executable SIMULA program. The program's data structures are illustrated in Fig. 6.

8.2.3 Purpose of MOD1

Simulation models and implementations generally are used to draw experimental conclusions. MOD1 is too predictable for a simulation to offer anything but limited insights. The memory direction positioning of MOD1's copy is the only random element in an otherwise deterministic program. Hence, we will use MOD1 as a starting point and build enhanced models with more features (MOD2, MOD3) from the constitutive building blocks delineated in MOD1. These are

(i) Program models
(ii) Memory models
(iii) Temporal behaviour
(iv) Spatial behaviour

[11] [24, pp. 4–9].

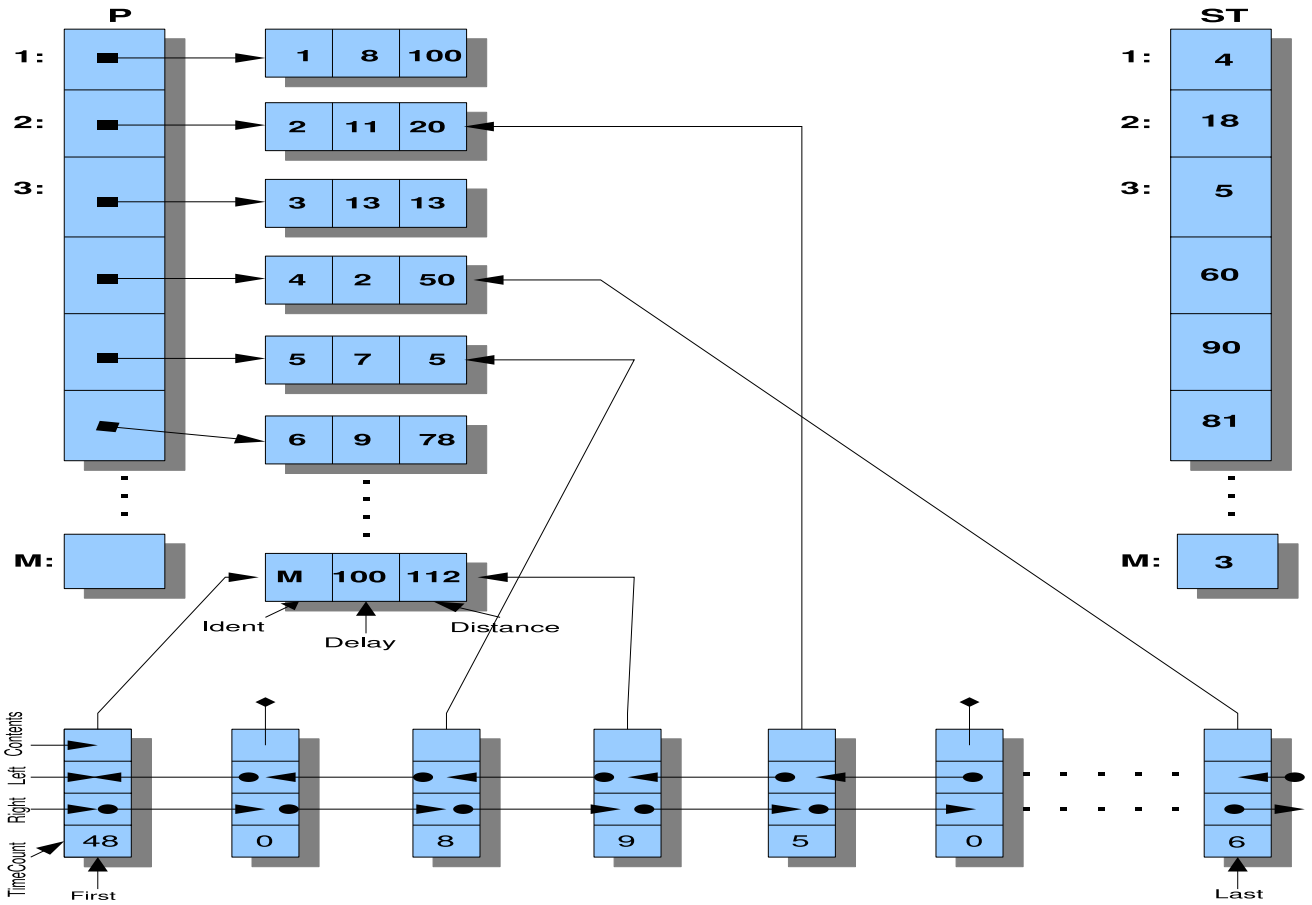


Fig. 6 MOD1: Global data structures

By varying one or more of these four components we can design other basic models, especially (iv) hints at a wide variety. Points (i) to (iv) are not completely independent from one another: The program's DELY and DISTANCE variables impact on the temporal (iii) and spatial behavior (iv). Another design consideration flows from the implied "von Neumann architecture" of the computing system. One feature of this architecture is a single data and instruction stream at any one point in time, also referred to as single-instruction single-data stream (SISD) [12] machine. Modern systems do not quite fit the SISD mold in that they boast more processing units (especially Input–Output processors), in effect re-organizing the machine into a multiple-instruction multiple-data (MIMD) stream entity. Current systems are rarely referred to as MIMD, since the number of concurrently working processors remains very small; the main idea behind MIMD being the leveraging of a large number (roughly 100–1,000) [26] of independent processors for maximum parallelism. There also exist single-instruction multiple-data (SIMD) and multiple-instruction single-data (MISD) machines, but practically all current systems can be roughly characterized as SISD. We ask ourselves

What features would MOD1 require if it were to serve as a model for unorthodox (i.e non-SISD) machines [6]?

We pre-suppose, of course, the existence of self-reproducing programs on such systems (Tables 40, 41).

8.2.4 Some SIMULA implementation aspects for MOD1

I As indicated in Sect. 8.2.3, MOD1 does not measure experimentally the quantitative behavior of individual programs. Since the programs in MOD1 are both persistent and unconditionally reproduce themselves, we have for all π_j, /; $j \in [M]$: The count $S_j(T)$ after the Tth memory cycle is given by

$$S_j(0) \cdot 2^{(T \div \text{DELY of } \pi_j)}$$

where $S_j(0)$ indicates π_j's count before the simulation has started. $S_j(T)$ can be printed in tabular form after every $\text{WHEN\_CON}^{\text{th}}$ memory cycle by calling

```
procedure CONTROL;
```

Table 40 MOD1: Copy to empty cell

```
1)  procedure COPY_RIGHT(C); ref (CELL) C;
2)  begin
3)      ref (CELL) HELP;
4)      [Set HELP to C]
5)      [Move HELP to cell at π_j DISTANCE
        (=C.CONTENS). If end of memory is reached,
        call ADDRIGHT to add the requisite number of
        cells.]
6)      if HELP.CONTENS== NONE;
7)      then
8)      [Move copy of π_j into empty cell pointed to by HELP
        (HELP.CONTENS := P[j])]
9)      else
10)     [Move HELP to the right until it points to an empty cell or
        it reaches the memory boundary. If the latter, set HELP :=
        ADDRIGHT and write copy of π_j into the cell pointed to by
        HELP.]
11) end;
```

Table 41 MOD1: Input parameters

Initial size of memory	N
Number of different program types	M
Tuples characterizing M program types	DELY, DISTANCE
Initial program count in memory	NUM
Tuple denoting memory distribution of NUM programs	PI, WHERE
Planned memory cycle count	TIME

II The unchecked program growth should spark some interest in the spatial positioning of the individual programs. The specification of MOD1's spatial behavior (iv) is quite arbitrary; hence we shall not delve into it further here, since other spatial arrangements are possible. However, MOD1 fields two procedures to facilitate spatial behavior analysis; these are

```
procedure DUMP;
```

DUMP effects a memory dump from left to right: Empty cells are denoted by '*', while a program's IDENT variable is printed for occupied cells.

```
procedure AVERAGE;
```

AVERAGE prints the average distance between instances of the individual programs in tabular format. The distance to an adjacent cell is 1.
The calls to both DUMP and AVERAGE are controlled by the variables `integer WHEN_ DUM` and WHEN_ AVE, respectively. DUMP is executed after every WHEN_ DUMth memory cycle; and similarly for AVERAGE.

III It follows from I. and II. that the implementation of MOD1 as seen in Appendix C.1 in ESM requires three printing parameters; WHEN_CON, WHEN_DUM, and WHEN_AVE.

IV Complexity:

Memory: MOD1 keeps the program type count constant, which means that variables P and ST are constant during simulation execution, as well. Solely the list representing memory is poised to grow during simulation, given enough TIME specified iterative cycles. Growth rate in a cycle is dependent on program type: A combination of short reproduction cycles (variable DELY) and large distances between copies (variable DISTANCE) gives rise to exponential program copy growth (see I.), which in turn induces exponential growth of the memory list. As such, the aforementioned criteria represent but just one factor.

Runtime: Several model parameters affect the runtime of MOD1's SIMULA implementation. This thesis will not attempt to give a tight bound; we will merely note that memory cycle time is dependent on memory length which in turn suggests an exponential runtime for MOD1.
We emphasize that this type of exponential behavior makes MOD1 unsuited for the purposes of statistical analysis; MOD1 represents a basic model, nothing more. The models that follow, however, will tackle unfettered exponential growth by modifying spatial behavior and introducing competitive constraints.

8.3 A competitive model

The basic model MOD1 allows any self-reproducing program π to freely deposit its copy $\overline{\pi}$ into an empty cell. Hence, there exists no real conflict nor competition between programs. The absence of competitive behavior prevents the formation of evolutionary processes. Another aspect of MOD1 is program persistence.

MOD2 expands MOD1's capabilities by enabling programs to deposit their copies into occupied cells, thereby erasing the former contents, i.e. other programs. The net result is twofold:

- We introduce conflict among—and hence competition between—programs.
- Overwriting a memory cell erases the former content and can be viewed as the "death" of a program.

By introducing the notion of (competitive) behavior among programs, we expand the list of our model components by adding

(v) Behavior among programs

We give a detailed description of MOD2 in Sect. 8.3.1.

8.3.1 Informal description of MOD2

(i) Programs: We define runtime as a characteristic metric of a program. Runtime denotes its reproduction time; the minimum number of tick counts needed for the program to reproduce. Hence, a 2-tuple, program identity and runtime, sufficiently characterize a program in this model.
(ii) Memory: Same as for MOD1.
(iii) Temporal behavior: Same as for MOD1.
(iv) Spatial behavior: At any time t, only a finite number of cells are occupied, which implies that there exist a left limit occupied cell $l(t)$ and a right limit occupied cell $r(t)$ which demarcate the occupied memory segment. A copy $\overline{\pi}$ of a program π must not be deposited arbitrarily far from this segment; only up to a distance of a constant number of cells. Hence, we denote by $L(t)$ and $R(t)$ the respective left and right distance-constrained deposition limits (Fig. 7).
Every cell within the segment serves with equal probability as a potential host for copy $\overline{\pi}$. This constrained spatial behavior guarantees a controlled program spread by preventing the creation of arbitrarily far positioned, isolated 'populations'.
(v) Behavior among programs: A self-reproducing program π deposits its copy $\overline{\pi}$ in an arbitrarily chosen memory cell within the constraints expounded in (iv). Should the memory be empty, no conflict arises; otherwise, another program π' is found residing at that cell, and some decision mechanism invoked to decide whether $\overline{\pi}$ is to overwrite π'. In the affirmative case, $\overline{\pi}$ overwrites π', otherwise $\overline{\pi}$ has no refuge and is deleted. Thus, one program is 'eliminated' in either case.

A specification of this decision mechanism (v) completes MOD2's description. Many different criteria may flow into a decision procedure, we shall limit ourselves for simplicity's sake to a mechanism that is based solely on properties of the two conflicting programs. We make this mechanism slightly more flexible by adding a probabilistic dimension which implies the consideration of other, albeit unspecified factors in the process.

Definition 8.1 Let $n \in \mathbb{N}_0$. A $n \times n$ matrix $V = (v_{ij}) \in \mathcal{M}_n(\mathbb{R})$ (ring of n row matrices over the real numbers) is called an n-row precedence matrix if $v_{ij} \in [0, 1] \subset \Re$ for all $i, j \in [n]$

Let $\mathbb{P} := \{\pi_1, \cdots, \pi_M\}$ be the set of program types in MOD2. An M-row precedence matrix in conjunction with an appropriate interpretation represents a decision mechanism for MOD2.

Definition 8.2 Let M be the program type count in MOD2. Let $V = (v_{ij})$ be an M-row precedence matrix. Components

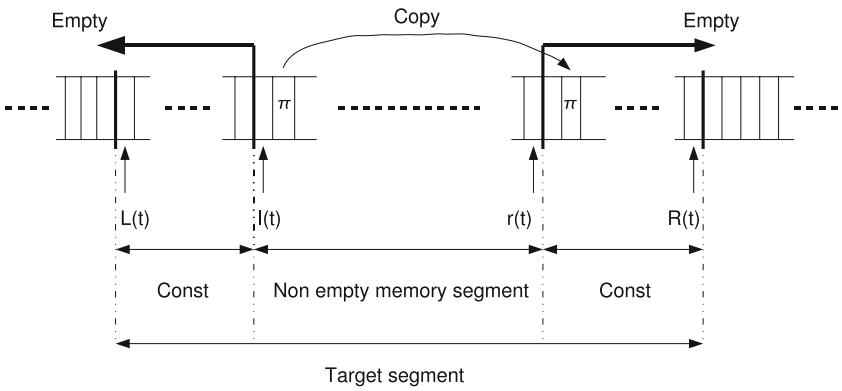


Fig. 7 MOD2: Spatial constraints

Table 42 MOD2: SIMULA implementation

```
1) class PROGRAM;
2)       begin
3)       integer IDENT, DELY;
4)       end;
```

Table 43 MOD2: Memory as dynamic array

```
1) class STORAGE(Q); integer Q;
2)       begin
3)       ref(CELL) array ELEMENT(1:Q);
4)       end;
```

v_{ij} are interpreted as follows: Should a copy $\overline{\pi}$ of program π of type π_i be written into a cell that already contains a program π' of type π_j, then $\overline{\pi}$ overwrites π' with probability v_{ij}. With probability $(1 - v_{ij})$ $\overline{\pi}$ does not overwrite π'; π' is preserved and $\overline{\pi}$ is deleted.

The M-row precedence matrices are fielded as decision mechanisms for MOD2, and hence represent one of its parameters. The inclusion of a precedence matrix renders MOD2 non-deterministic.

8.3.2 A SIMULA implementation of MOD2

(i) Programs: The SIMULA program representation is simpler compared to MOD1. `DELY` and `IDENT` denote reproduction time and program identification, respectively (Table 42).

(ii) Memory: Implementation of the memory structure poses some challenges. On the one hand we need a list-based structure that can grow potentially infinitely large; on the other hand in light of Sect. 8.3.2(iv) direct cell access through an array-based memory structure is desirable, as well. Since both requirements are mutually exclusive, a compromise must be found. Since direct memory access positively influences MOD2's runtime, we shall make this our priority and represent memory as an array. To accommodate potentially infinite memory requirements, the array implementation must be dynamic. SIMULA's class concept make dynamic arrays possible (Table 43). The individual memory cells (type: `CELL`) are represented the same way as in MOD1 (Table 44). The global pointer `ref (STORAGE) STOREPOINTER` is used for memory access. A memory segment containing N cells is initialized at the beginning of the simulation through the statement

```
STOREPOINTER := new STORAGE(N);
```

Table 44 MOD2: Memory cell

```
1) class CELL;
2)       begin
3)       ref(PROGRAMS) CONTENS;
4)       integer TIMECOUNT;
5)       end;
```

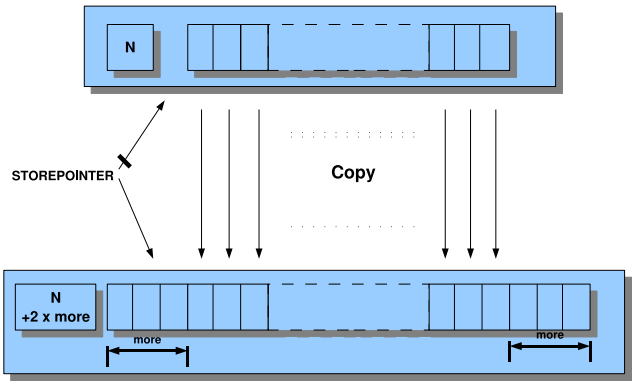


Fig. 8 MOD2: Memory expansion

Since cell occupancy density increases monotonically during simulation, memory has to be expanded ever so often.The variable `integer PERCENT` denotes the density threshold: Should memory occupancy reach `PERCENT` %—which is tested by calling `boolean procedure OVERFLOW`—memory is expanded by invoking

```
procedure NEW_STORAGE(MORE); integer
 MORE;
```

`NEW_STORAGE` creates a new object of length $N + 2 * MORE$ of type `STORAGE`, copies the content of the old memory structure into this new object and sets the global pointer `STOREPOINTER` appropriately (Fig. 8). After the `NEW_STORAGE` call, available memory will have increased at each end by `MORE` cells. Variable N, denoting the current memory size, will have increased by $2 * MORE$. Since the two parameters `PERCENT` and `MORE` influence the mechanism of memory expansion, they are important for the SIMULA implementation of MOD2. Infinite memory is better and better approximated by a smaller `PERCENT` and a larger `MORE`. Should `PERCENT` be set to >100, MOD2 reverts to a finite memory model.

(iii) Temporal behavior:
Initialization time: Similar to MOD1, we read in M programs (rather: program types) $\pi_1, \ldots, \pi_M$ at simulation initialization time and store them in field

```
ref (PROGRAM); array P[1:M];
```

Table 45 MOD2: Simulation

```
1)  for   T:=1   step 1   until TIME do
2)  begin
3)        [set read direction]
4)        if [read direction = 'right to left']
5)        then
6)        begin
7)              for I:=1 step 1 until N do
8)              if [I-th cell is not empty]
9)              then
10)             begin
11)                     MATCH(I)
12)                     if OVERFLOW then NEW_STORAGE(MORE)
13)             end
14)       end
15)       else
16)       for I:=N step -1 until 1 do
17)       if [I-th cell is not empty]
18)       then
19)       begin
20)             MATCH(I)
21)             if OVERFLOW then NEW_STORAGE(MORE)
22)       end
23) end   *** S I M U L A T I O N ***;
```

Field `integer array ST[1:M];` keeps track of the program type count `ST[j]` of program π_j. Initial values for `ST` are read in, as well, which indicates the program type count written to memory at the start of the simulation: For every $j \in [M]$, `ST[j]` copies of program π_j is assigned to memory by setting the appropriate pointers. We invoke the random number generator `RANDINT(1,N,U)` to randomize program position uniformly in memory. Overwriting situations at initialization time are avoided, and we make sure that

$$\sum_{j=1}^{M} \texttt{ST[j]} \leq \texttt{N}$$

Finally, we read in the M-row precedence matrix and write it to field

```
CONFLICT[1:M, 1:M]
```

Simulation: We cycle through memory bidirectionally with equal probability `TIME` times. The cycle direction is randomized to prevent preferential treatment of programs. For every non-empty cell encountered during a read cycle, we call procedure `MATCH`. `MATCH` checks whether the residing program can self-reproduce and potentially creates a copy. Since `MATCH` may lead to memory density exceeding the `PERCENT` threshold, procedure `NEW_STORAGE` may be triggered (Table 45).

(iv) Spatial behavior:
The selection of the memory cell into which a program deposits its copy is effected by calling the SIMULA random number function
`RANDINT(1,N,U_CELL)`, where N denotes memory size, and `U_CELL` a required parameter of type `name`. The selection of each cell is equiprobable (for a detailed description of `RANDINT`, see [24]). We slightly modified the cell selection procedure given in Sect. 8.3.1(iv). In conjunction with the memory expansion mechanism outlined in Sect. 8.3.2(iii), we realize the overall end result delineated in Sect. 8.3.1(iv).

(v) Behavior among programs:
Instead representing the precedence matrix $V = (v_{ij})$ as an element of $\mathcal{M}_M(\mathbb{R})$, the SIMULA version of MOD2 represents it as an element of $\mathcal{M}_M(\mathbb{N})$:

```
integer array CONFLICT[1:M.1:M]
```

Each v_{ij}=`(CONFLICT[i,j])` is a percentage value, and thus can range from (0, 100]. For technical programming reasons, zero values are disallowed (a deviation from Sect. 8.3.1(v)).
Appendix C.2 in ESM contains an extensively commented SIMULA implementation of MOD2. An illustration of the implementation's data structures is shown in Fig. 9.

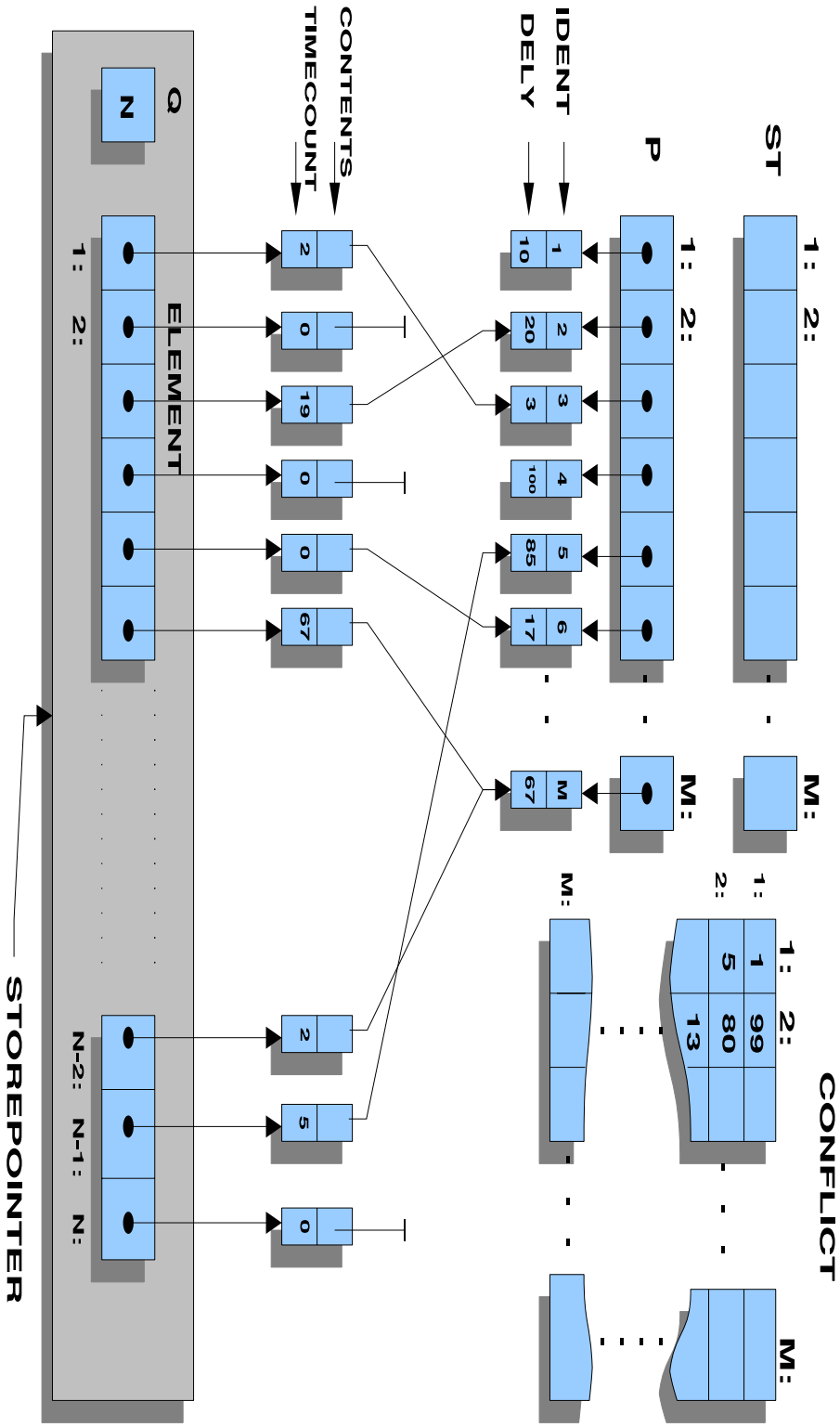


Fig. 9 MOD2: Global data structures

8.3.3 Some SIMULA implementation aspects for MOD2

I. This particular SIMULA implementation of MOD2 allows for simulations presuming finite as well as infinite memory, depending on PERCENT.
II. We include two new procedure for output support, DUMP and CONTROL. This necessitates the introduction of model-independent parameters WHEN_DUM and WHEN_CON (see Sect. 8.2.4.I and II.)
III. MOD2 allows us to investigate certain questions experimentally, e.g.:

- For a given program type, can a relatively weak position in the precedence matrix be compensated for by a short reproductive cycle, thus enabling the particular program to assert itself?
- Given M program types, their respective DELY components as well as a precedence matrix, how do the proportions of the individual programs types change over the duration of the simulation? How long till one or more program types are extinct? Will a 'winner' eventually assert itself in finite time and overwhelm all other types?
- Questions along the lines above, dependent on "population density" (controlled by memory parameters MORE and PERCENT)
- Many more questions.

The abundance of parameters within the SIMULA implementation of MOD2 offers a fertile experimental playground. Alas, in the context of this thesis, we shall not investigate these questions any further (Tables 46, 47).

IV. Complexity:

Memory: The number of program types that appear in MOD2 remains constant throughout the duration of the simulation. Hence, the memory requirements set by CONFLICT, ST and P remain unchanged, as well. Merely the memory-simulating dynamic field referenced by STOREPOINTER may grow during the simulation, depending on the parameters MORE and PERCENT (see Sect. 8.3.2(ii)). Should PERCENT >100, the memory field remains constant; otherwise, its size will increase exponentially throughout the lifetime of the simulation, depending on the number of memory cycles. To counteract the explosive growth seen in SIMULA program MOD1, a dampening factor is introduced, (100-PERCENT)/100, denoting the proportion of free memory cells. It should be noted that resizing induces the creation of a new object of type STORAGE (by calling NEW_STORAGE). The old object stays resident in memory.

Runtime: In general, runtime is dependent on the number of memory cycles and the length of the memory-simulating field. Since the size of grows exponentially with memory cycle count, so does runtime. Again, the dampening factor (100-PERCENT)/100 seeks to counteract exponential runtime growth. Limit cases:

a) Setting no limits on memory length and keeping memory density relatively small throughout the simulation (by choosing a low value for PERCENT). This results in a paucity of conflicts and an near-unconstrained program ability to deposit copies; leading to a near-unmitigated exponential explosion of program instances. This situation is comparable to MOD1 (see Sect. 8.2.4. IV)
b) Letting memory length remain constant (by choosing PERCENT >100). Then, from some memory cycle onwards, all cells will be occupied. Consequently, runtime will linear in the number of memory cycles, since the runtime of procedure MATCH (no other procedure will be called) is bounded by a constant.

V. After a NEW_STORAGE call, available memory will have increased by 2∗MORE cells. It may be more advantageous to condition the number of added cells proportionally to current memory length.

9 Program evolution

9.1 Motivation

The plant and animal life we see today has evolved from simpler life forms by varying its features slowly, but steadily. This process is called biological evolution, which continues to this very day, albeit imperceptively. Evolutionary theory tries to causally explain this process, and can be succinctly summarized as follows [11,15,25,27]:

Organisms produce far more descendants than necessary for the propagation of their species. These descendants exhibit some genetic variation (see Sect. 7.2), this is true even in the case of shared parenthood. This variation is enabled by the ability of genes to mutate (see Sect. 7.2). Mutation rates of living organisms are very low, on the order of 10^{-4} to 10^{-7} per gene. (These values are independent of a given species' generational cycle and itself the result of evolutionary processes balancing species' adaptability against instabilities in the genome). Since genes determine an individual's features, the vast majority of descendants are differentiated in their traits. There is a constant struggle for life and only those organisms that have best adapted themselves to the

Table 46 MOD2: Match

Workings of procedure MATCH

```
1)  procedure MATCH(I);          integer I;
2)  begin
    ⋮
3)           boolean IS_COPY;
4)           IS_COPY := false;
5)           [increase TIMECOUNT of I-th memory cell by 1];
6)           if [program in I-th cell's TIMECOUNT equals DELY];
7)           then
8)           begin
9)              comment **** Reproduction of program in I-th cell *** ;
10)             [Set TIMECOUNT of I-th cell to 0];
11)             [Choose random memory cell. Let choice be cell W.];
12)             comment **** see (iv) below *** ;
13)             if [cell W is empty ];
14)             then
15)             begin
16)                comment **** no obstacle to copying *** ;
17)                [Write copy into W-th cell] ;
18)                IS_COPY := true ;
19)             end
20)             else
21)             begin
22)                comment **** W-th cell occupied *** ;
                   [Use precedence matrix to decide
                   whether a copy of program residing in I-th cell ..
                    ..can overwrite program in W-th cell];
24)                comment **** see (v) below *** ;
25)                if [cannot overwrite ];
26)                then comment **** nothing happens *** ;
27)                else
28)                begin
29)                   [Write copy to W-th cell];
30)                   IS_COPY := true;
31)                end
32)             end;
33)             comment **** If the a copy of the program ..;
                         residing in I has been written ..
                         to memory, TIMECOUNT of the W-th cell ..
                         has to be updated consistent..
                         with the memory read direction ***;
34)             if IS_COPY
35)             then
36)             begin
37)                if W ≤ I
38)                then
39)                begin
40)                   if [read direction = 'left to right']
41)                   then [set TIMECOUNT of W-th cell to 0]
42)                   else [set TIMECOUNT of W-th cell to -1]
43)                end
44)                else
40)                if [read direction = 'left to right']
41)                then [set TIMECOUNT of W-th cell to -1]
42)                else [set TIMECOUNT of W-th cell to 0]
43)             end
44)          end
45) end      *** M A T C H ***;
```

Table 47 MOD2: simulation

Example

```
M = 5
Conflict between program type π2's copy π with type π3's copy π′, i.e
π is trying to overwrite π′:
Decision (see description of MATCH)
if CONFLICT[2,2] < RANDINT(1,100,U_CONFLICT)
then [π does not overwrite π′]
else [π does overwrites π′]
```

environment - the fittest - will survive and reach their reproductive stage. Sustained natural selection first marginalizes, then wipes out the population's feebler individuals (see [25, p. 337]). The unremitting selection pressure induces more and more environmentally optimal features in individuals (transformative selection). New mutations arise at a constant rate irrespective of prior adaption to a stable environment. The probability of 'positive' mutations, however, decreases as adaptive corrections take root over time. 'Negative' mutations are dealt with in two ways: Insofar they have not had lethal implications, selection ensures that the population's genetic composition remains constant by discarding manifested negative mutations (stabilizing selection). Should positive mutation arise or the environment change yet again, transformative selection pressure reasserts itself. Mutation and selection are the 'engine' of evolution; however, other factors such as isolation and randomness (see [15, p. 317ff]), meiotic and mitotic reproduction may play a role, as well.

Section 7.4 compared self-reproducing programs with viruses. Even though one cannot classify viruses as living organisms they are subject to evolution. This is because viruses are capable of mutations and experience similar selection pressure in the 'struggle for life'. This suggests that self-reproducing programs can evolve as well if subjected to mutation and selection. Sect. 8.3's MOD2 introduced a model incorporating competitive behavior (= struggle for life) among programs. MOD2's program types were characterized by their reproductive cycle and position in the precedence matrix: the programs that enjoy a short reproductive cycle and - due to a advantageous entry in the precedence matrix - have an easier time locating memory space for their copies are likely to assert themselves. Hence, selection pressure in MOD2 is geared towards short reproductive cycle time and an advantageous precedence matrix entry. Should a program type be able to vary one or both of these parameters, the subsequent program could have an easier time asserting themselves in MOD2. Mutations can cause these parameters to vary. Section 9.2 expands MOD2 by allowing for program type mutation, the net result being the introduction of a model satisfying the requirements needed for evolution. Hence, the SIMULA version implements a program simulating the evolution of self-reproducing programs.

We note that computer programs in general are suited to simulate evolutionary processes: Since such processes (biological, chemical, cosmic,...) require a very long period of time to generate and manifest changes, the fundamental modeling constraint is one of temporal duration. Only fast computer systems, capable of executing many operations in a fraction of a second, can compress such eons into an acceptable time frame (see [8]).

9.2 MOD3: A model for the evolution of self-reproducing programs

Given a program π, we pre-suppose that with a certain probability p_1 (model parameter) errors will occur during reproduction, such that the resulting differing program $\overline{\pi}$ constitutes a mutation of program π. Generally, errors will be minimal and hence the differences between π and $\overline{\pi}$ will not be pronounced. Since MOD3 programs are characterized by their respective reproductive cycle time and their precedence matrix entries, these are the visible values affected by mutation, provided that $\overline{\pi}$'s reproductive ability is not impaired. We denote the case in which a mutation lead to non-self-reproducing programs as a lethal mutation. Since mutations are random and punctuated, lethal mutations may occur at any time. MOD3 controls their occurrence with probability parameter p_2. We note that in principle every non-lethal mutation gives rise to the first manifestation of a new program type. MOD3 records such mutants together with their 'pedigree'; thereby incorporating each non-lethal mutation into the fold of available program types. Since selection pressure favors shorter reproduction time which manifests itself through more advantageous precedence matrix entries, those mutations evidencing improvements in these values (an increase in fitness) will have an evolutionary advantage viz their parents. Under certain circumstances, such mutants may crowd out the program types from whence their parents sprung (selection). It goes without saying that mutated program types may mutate again. Since program reproduction passes off conceptually in an "asexual" manner, every mutation can be viewed as a lineage starting point of divergent program types (clones, see [25, p. 313]).

Table 48 MOD2: input parameters

Initial size of memory	N
Number of different program types	M
M program types, characterized by size and initial count	DELY, ST[...]
Initial program count in memory	NUM
M*M elements of the precedence matrix, values ranging from (0,100]	CONFLICT
Planned memory cycle count	TIME
Memory parameters	MORE, PERCENT

9.2.1 *Informal description of* MOD3

(i) Programs: Programs in MOD3 are represented by their reproductive cycle time and name. The name of mutated programs sheds light on the program's 'ancestral roots'.
(ii) Memory: Same as MOD2.
(iii) Temporal behavior: Same as MOD2. In addition, a program is capable of reproducing (and hence potentially fathering a mutated offspring) if it has been active for t time ticks. Parameters p_1 and p_2 denote the probability of a mutation and subsequent lethality of said mutation, respectively. Should the mutation be non-lethal, either the offspring's DELY component or its competitive conduct (precedence matrix) will be changed with probability p_3 and $1 - p_3$, respectively, and the program type count in MOD3 increases by 1. The mutated offspring is subsequently treated like any other error-free copy.
(iv) Spatial behavior: Same as MOD2. Mutants and error-free copies are treated the same way.
(v) Behavior among programs: As in MOD2, the behavior among the individuals programs is determined by the precedence matrix. Should a non-lethal mutation arise, the matrix's rows and columns are expanded to accommodate the new program type and specify its behavior towards the other programs.

9.2.2 *A* SIMULA *implementation of* MOD3

(i) Programs: Table 48 gives an overview of MOD3's SIMULA parameter structure. DELY, IDENT, PROGNAME, MUT denote reproduction time, program identification, program name, and mutation count, respectively. Although PROGNAME would suffice for unambiguous program identification, code implementation issues (array access) necessitate the IDENT variable. MUT keeps track of the program's mutated offspring count. DELY and PROGNAME are analogous to Sect. 9.2.1(i).
(ii) Memory: The memory structure follows MOD2's lead. We also reuse the memory expansion mechanism, as well as its control parameters integer MORE and PERCENT (procedures: NEW_STORAGE, OVERFLOW).
(iii) Temporal behavior: We note that the program type count M will not likely remain constant during the lifetime of the simulation. This entails that all variables which exhibit M components in MOD2 will have to be of dynamic length in MOD3. This holds true for the precedence matrix, as well. We show these modifications in Table 49.
Lines VECTOR.x, S.x and CONFLICT.x are substituted for ref (PROGRAM) array P[1:M], integer array ST[1:M], and integer array CONFLICT [1:M,1:M], respectively.

Table 49 MOD3: SIMULA implementation parameters

```
1) class PROGRAM;
2)       begin
3)       integer IDENT, DELY, MUT;
4)       text PROGNAME;
5)       end;
```

Initialization: This remains analogous to Sect. 8.3.2 (ii), with consideration of the aforementioned data structure modifications. The added parameters MUT and PROGNAME are initialized thusly: Let $\{\pi_1, \ldots, \pi_M\}$ be the set of initial program types. Then MUT is set to 0 for each π_j. Furthermore, π_1's PROGNAME is set to "P1", π_2's PROGNAME is set to "P2", and so on.

Simulation: In general, we can adopt the description given in Sect. 8.3.2(iii). Since MOD3 extends MOD2 significantly, we have to make some changes to accommodate the generation and treatment of mutations. These changes manifest themselves in a set of procedures which have to be called *in toto* prior to calling procedure MATCH. Before we give a modified description of MATCH, we have to outline these additional procedures.
A program's determinant features are captured by its DELY variable and its position in the precedence matrix. Only mutations of these characteristics are selection-relevant. MOD3's SIMULA version induces

Table 50 MOD3: Dynamic variables

```
VECTOR.1)   class PROG(P); integer P;
VECTOR.2)   begin
VECTOR.3)         ref (PROGRAM) array VECTOR[1:P];
VECTOR.4)   end;
VECTOR.5)   ref (PROG) PROGPOINTER;

S.1)        class ST(P); integer P;
S.2)        begin
S.3)              integer array S[1:P];
S.4)        end;
S.5)        ref (ST) STPOINTER;

CONFLICT.1) class CONFLICT(P); integer P;
CONFLICT.2) begin
CONFLICT.3)       integer array MAT[1:P,1:P];
CONFLICT.4) end;
CONFLICT.5) ref (CONFLICT) CONPOINTER;
```

program mutation by calling

```
ref (PROGRAM) procedure MUTANT(X);
  ref (PROGRAM) X;
```

MUTANT produces a pointer to an object of type PROGRAM. This object finds one or both of its aforementioned determinant features slightly altered compared to the original program, denoted by pointer parameter X. Hence, it represents a mutation and the generation of a new program type. This also necessitates increasing by 1 variable M, which denotes the count of available model program types in the model. This increase is effected before calling procedure MUTANT.

Workings of MUTANT:

I. MUTANT first creates a new object of type PROGRAM and initializes the fields that are not directly relevant to model behavior. The resulting object -constructed as a mutant of program X - is addressed through pointer HELP (Table 50)
II. The creation of a new mutant necessitates a row and a column expansion of the precedence matrix. Since at procedure MUTANT invocation time, it has not yet been determined whether the mutated program will be kept or discarded (see the functionality of MATCH below), the precedence matrix is not yet changed. Rather, the column and row are held in a new data type object (See Table 51 for the data type) for putative matrix expansion at a later time. MUTANT creates an object of type FIELD and assign it to global variable CHANGE_CONFLICT.

```
CHANGE_CONFLICT := new FIELD(M);
```

The precedence matrix's X.IDENTth row and column entries regulate the competitive behavior between instantiations of program type X and other program types. The mutant's behavior towards other program types will be similar to program X's: To this end, the X.IDENTth column is copied into the first row of CHANGE_CONFLICT.V and the X.IDENTth row is copied into CHANGE_CONFLICT.V's second row (Fig. 10).

CHANGE_CONFLICT.V(1,M), CHANGE_CONFLICT.V(1,X.IDENT) and CHANGE_CONFLICT.V(2.X.IDENT) represent the components which specifies type-conflict behavior for instantiations of (mutant) type HELP, as well as between HELP and original program X, respectively. The component values are generated randomly with the help of RANDINT. Since mutant and original program are closely related, these values may deviate only up to a factor of two from the values of X's precedence matrix. Hence, with the exception of the three aforementioned components, the mutant exhibits the same type-conflict behavior as the original.
III. It remains to be determined how the mutant will differentiate itself with respect to model-relevant features from the original. We handle this with variable PROB_DELY which captures model parameter p_3 mentioned earlier: With probability PROB_DELY$*10^{-3}$, X's DELY is mutated with the help of RANDINT. With probability (1-PROB_

Table 51 MOD3: Mutant creation

```
1) ref (PROGRAM) HELP;
2)     HELP := new PROGRAM
3)     HELP.MUT := 0;
4)     HELP.IDENT := M;
5)     HELP.PROGNAME := CREATE_NAME;
6)     comment See above from description of CREATE_NAME
7)     X.MUT := X.MUT + 1;
8)     comment *** X's MUT is increased
       to reflected its added mutation ***;
```

DELY) $*\ 10^{-3}$, the mutant's conflict behavior towards MOD3 program types other than X is changed.

- Should the DELY component value be chosen for changes, the mutant and original value will differ by at most 100%.

```
HELP.DELY := RANDINT(1,2*X.
DELY, U_DELY2);
while HELP.DELY = X.DELY do;
HELP.DELY := RANDINT(1,2*X.
DELY, U_DELY2);
```

- Should the mutant's conflict behavior chosen to be altered, exactly one component value of CHANGE_CONFLICT will be changed; again at most by 100%. This component, chosen randomly by RANDINT, must have been copied from the old precedence matrix; hence components at indices [1,M], [2,M], [1, X.IDENT] and [2, X.IDENT] are disallowed (Table 52).

IV. The steps in III. lead to a completed mutant, as well as the additional row and columns of the precedence matrix in the form of HELP and CHANGE_CONFLICT, respectively. We finish the procedure call to MUTANT with the assignment MUTANT := HELP.

Example:

M = 3

X ⟼	2 ↑ IDENT	11 ↑ DELY	3 ↑ MUT	P2 ↑ PROGNAME

CONPOINTER ⟼		1	2	3
	1:	12	588	617
	2:	128	5	799
	3:	875	35	16

A mutation occurs:

M := M + 1

X ⟼	2	11	4	P2

CHANGE_CONFLICT ⟼		1	2	3	4=M
	1:	588	**3**	35	**8**
	2:	128	**10**	799	**8**

Mutation DELY component

MUTANT ↪	4	19	0	P2.1

CHANGE_CONFLICT as above

Mutation conflict behavior

MUTANT ↪	4	11	0	P2.1

CHANGE_CONFLICT

	1	2	3	4=M
1:	588	3	35	8
2:	**86**	10	799	8

↪ changed components

The mutant's PROGNAME variable is set within procedure MUTANT by calling function
`text procedure CREATE_NAME(X); ref (PROGRAM) X;`

Workings of `CREATE_NAME`:

CREATE_NAME receives as a formal input parameter X a pointer to an object of type PROGRAM which returns a text string denoting the name of a mutant derived from program X. This name is constructed by concatenating the integer component X.MUT (parsed as text), the symbol ".", and the text component X.PROGNAME.

Example:

```
a)  X  ⟼  1     11    3     P1
          ↑     ↑     ↑     ↑
          IDENT DELY  MUT   PROGNAME

          CREATE_NAME = P1.3

b)  X  ⟼  5     3     1     P1.3

          CREATE_NAME = P1.3.1
```

Since the generation of a program mutant entails an increase-by-one of the program's MUT variable, the mechanism guarantees unambiguous names for successive mutations. In addition, since the object passed into parameter X may be itself a mutant (see example b) above), it is possible to reconstruct the genealogical 'tree' through component PROGNAME (see Fig. 11). Note that MOD2's integer IDENT component, though adequate program identification in MOD2, lacks this feature. We mentioned repeatedly that in general, a mutation adds to the number of available program types. This manifests itself at first in the increase to variable M. Should the mutated program succeed in establishing itself (see the description of procedure MATCH), an expansion of the dynamic arrays which store, register and manage the programs is required. These changes are effected through procedure calls

`procedure NEW_PROG(P); ref (PROGRAM) P;`
`procedure NEW_ST(T); ref (PROGRAM) T;` and
`procedure NEW_CONFLICT(A); integer array A;`

Workings of `NEW_PROG`:

PROGPOINTER points to the field responsible for the actual storage of the program types. Should a new mutant emerge (passed via pointer P as a parameter to NEW_PROG), a component has to be added to this field. Fig. 12 shows the workflow schematically: A new object of type PROG is created, the field expanded, its entries copied over and finally PROGPOINTER set to the new updated structure.

Workings of `NEW_ST`:

The field referenced by STPOINTER keeps track of program type count; at any time in the simulation, the ith component denotes the count of program type i. Upon emergence of a new mutation, an additional component is added to the field and initialized with value 1. Apart from that, the order of events mirrors NEW_PROG's (Fig. 13).

Workings of NEW_CONFLICT:

CONPOINTER points to the field that stores the precedence matrix. A new mutation expands the matrix by one column and one row. Both row and column specify the mutant's conflict behavior, and are passed via formal parameter `integer array A` to NEW_CONFLICT. At NEW_CONFLICT call time, the object pointed to by CHANGE_CONFLICT is passed in as a parameter. This object in turn is generated beforehand in procedure MUTANT (see workings of MUTANT in the preceding section). Again, the order of events in NEW_CONFLICT is largely analogous to NEW_PROG's and NEW_ST's. Fig. 14 picks up on the example from procedure MUTANT.

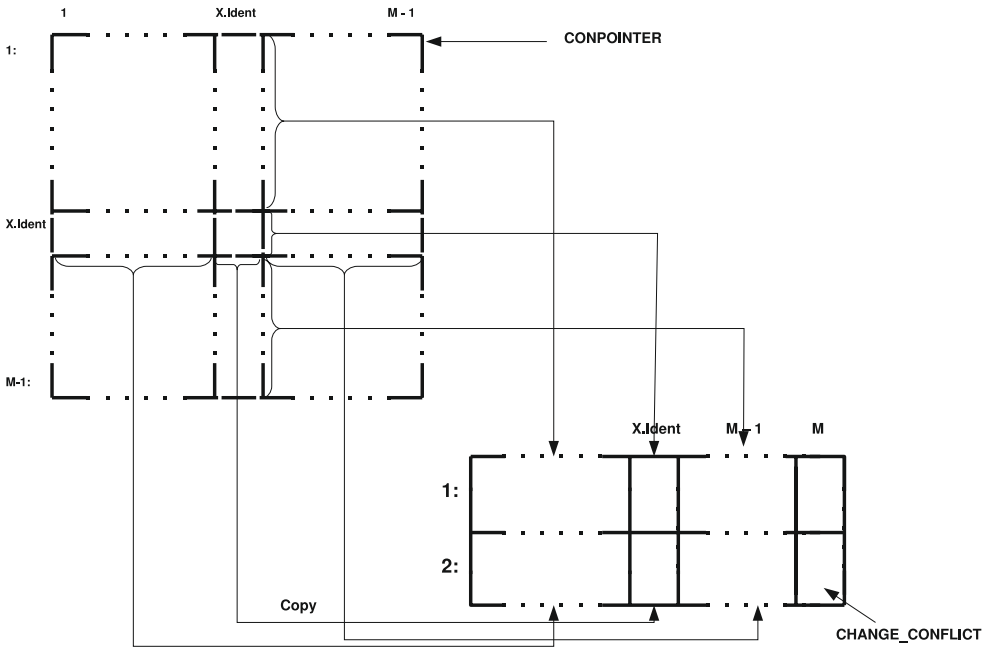


Fig. 10 MOD3: Mutant conflict behavior

Table 52 MOD3: Row and column creation

```
1) class FIELD(P); integer P;
2)       begin
3)       integer array V[1:2,1:P];
4)       comment **** V[1,...] denote the column;
                      V[2,...] denotes the row ***;
5)       end;
```

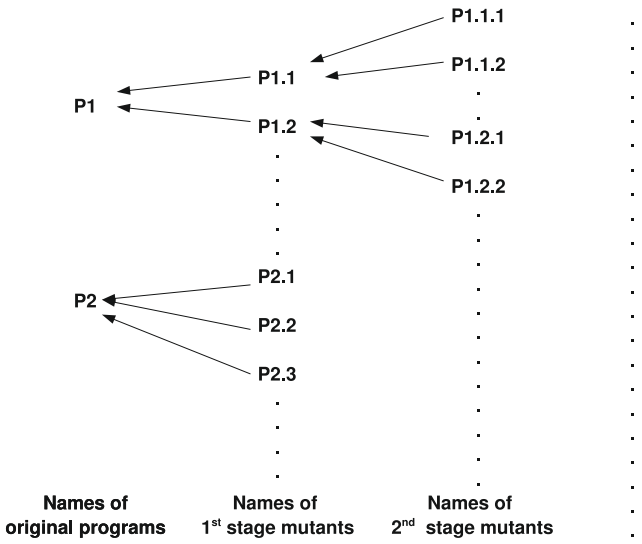


Fig. 11 Program genealogy

After our elucidating the procedures used to create and manage mutations, we may move on to specify procedure MATCH. Like its namesake in the MOD2 SIMULA program (see Sect. 8.3.2), MATCH constitutes the program's core procedure. We additionally adopt in its entirety the simulation scheme delineated in Sect. 8.3.2(iii), Table 53.

(iv) Spatial behavior:
We follow the approach delineated by SIMULA program MOD2, since mutation options do not change spatial behavior.

(v) Behavior among programs:
Again similar to SIMULA program MOD2. However, every element v_{ij} of the current precedence matrix is interpreted as a 'per mille' value rather than a percentage value, as was the case for MOD2's v_{ij}. Thus, the entries in a given MOD3 precedence matrix can range from (0, 1000].

Appendix C.3 in ESM contains an extensively commented SIMULA implementation of MOD3. An illustration of the implementation's data structures is shown in Fig. 15.

9.2.3 Some SIMULA *implementation aspects for* MOD3

I. This particular SIMULA implementation of MOD3 allows for simulations presuming finite as well as infinite memory, depending on PERCENT.
II. We integrated three procedures for output support: DUMP, CONTROL and AVERAGE. Thus, the program contains model-independent parameters WHEN_DUM, WHEN_CON and WHEN_AVE (see Sect. 8.2.4.I and II.)

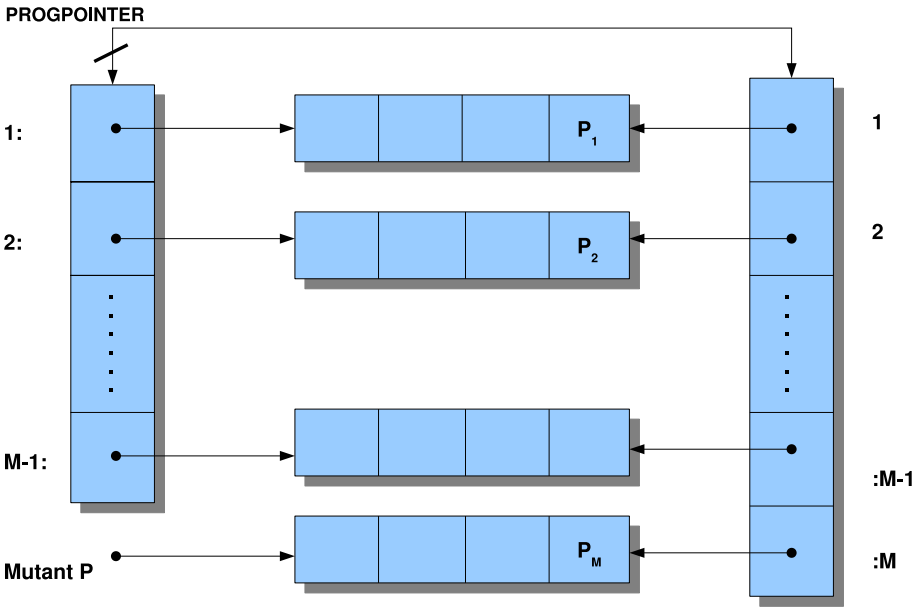


Fig. 12 Managing new (mutant) program types

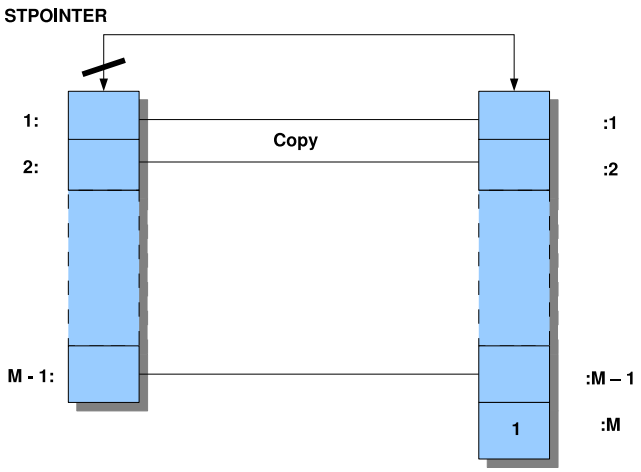


Fig. 13 Keeping track of (mutant) program type count

III. MOD3 reverts to the SIMULA program MOD2 should `PROB_MUT` be set to 0 (i.e. mutations are disallowed).

IV. MOD3 allows us to investigate certain questions experimentally. It follows from III. that the experimental questions raised for MOD2 can be answered within the MOD3 framework. Additionally, MOD3 may shed some light on questions pertaining to evolution:

- Does an optimal mutation frequency exist for the given model?
- Is there a 'red line' mutation rate which should not be exceeded lest the mutants become unstable?
- By judiciously varying `PROB_DELY`, we may investigate the effects of the reproductive cycle time's selection effectiveness (the `DELY` component) and its position in the precedence matrix.
- How are mutations able to prevail against other programs -and their own cousins, ancestors and predecessors? We find programmatic support for this investigation with the help of the taxonomic genealogical 'tree' (`see CREATE_NAME`).
- We can vary population density through `MORE` and `PERCENT` and repeat the line of questioning delineated above.
- Many more questions ..

Again, the SIMULA implementation of MOD3 offers a fertile experimental playground. Alas, in the context of this thesis, we shall not investigate these questions further (Tables 54, 55).

V. Complexity:
Without going into further details, it is apparent that the analysis of MOD2's memory requirements and runtime is applicable to MOD3, as well. In general, complexity grows exponentially with memory cycle count. Again, a dampening factor counteracts this exponential growth; its effectiveness dependent on permissible limits of simulated memory density (see Sect. 8.3.3.IV). There is also some additional overhead due to the complicated data structures and new procedures needed to accommodate and manage mutagenic behavior. However, its overall effect on complexity is negligible, provided that realistically small mutation rates (which keep `M` small) are specified (see Table 56)

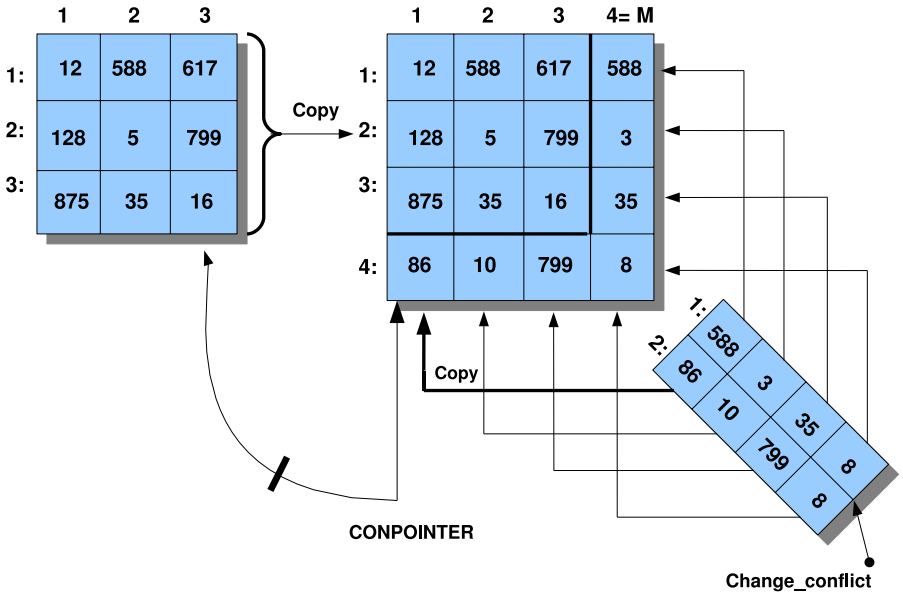


Fig. 14 Generation of new precedence matrix

Table 53 MOD3: Change behavior component value

```
1)  integer I,J,K,U_CONFLICT
    ⋮
2)  J := X.IDENT;
3)  while J=X.IDENT do
4)     begin
5)        J := RANDINT(1,2*(M-1), U_CONFLICT);
6)        if J ≤ M-1;
7)           then I:=1
8)        else
9)           begin
10)             I:=2;
11)             J:=J-(M-1);
12)          end
13)    end;
14) K := RANDINT(1,2*CHANGE_CONFLICT.V[I,J], U_CONFLICT);
15) while K=CHANGE_CONFLICT.V[I,J] do
16)    K := RANDINT(1,2*CHANGE_CONFLICT.V[I,J], U_CONFLICT);
17) CHANGE_CONFLICT.V[I,J] :=K;
```

VI. We adopt the memory expansion mechanism delineated in Sect. 8.3.3-IV for the SIMULA implementation of MOD3.

VII. In MOD3, a mutant will differentiate itself with respect to model-relevant features from its original by altering exactly one value by at most by 100% (see description of MUTANT). This leeway of 100 % has been arbitrarily chosen and may alternatively be specified by a variable parameter. In MOD3, the lower limit for mutation rates was programmatically set to 10^{-8}. We note that this rate was taken from biological processes, and may not have the same relevance when applied to evolution in a computing environment. Thus, this value, along with the fixed lower limit for mutation lethality, may be passed in as a parameter, as well.

Table 54 Workings of procedure MATCH

```
1)  procedure MATCH(I); integer I;
2)  begin
    ⋮
3)            boolean IS_COPY;
4)            IS_COPY := false;
5)            [increase TIMECOUNT of i^th memory cell by 1];
6)            if [program in i^th cell's TIMECOUNT equals DELY];
7)            then
8)            begin
9)              comment **** Reproduction of program in i^th cell *** ;
10)             [Set TIMECOUNT of i^th cell to 0];
11)             [Choose random memory cell. Let choice be cell W.];
12)             comment **** see 8.3.2.(iv) above *** ;
13)             [Decide whether program in i^th cell will mutate ];
14)             comment **** The mutation probability is specified ..
                       by PROB_MUT*10^-8 = p_1, where p_1 is a model parameter ..
                       implemented in the program by integer PROB_MUT ***
15)             if [to mutate program in i^th cell];
16)             then
17)             begin
18)               [Decide whether mutation is lethal] ;
19)               comment *** The lethality probability of a mutation ..
                         is PROB_LETAL*10^-6 = p_2, wjere p_2 is a model parameter ..
                         implemented in the program by integer PROB_LETAL ***
20)             if [mutation is lethal];
21)             then [increase DELY of i^th memory cell by 1];
22)               comment *** This registers the mutant.
                         Should the mutation be viable, procedure ..
                         MUTANT increases the MUT component ***
23)             else
24)             begin
26)               comment *** viable mutation *** ;
27)               M := M+1;
28)               comment *** Save pointers to old precedence matrix .. ;
                         and to program memory structure ***
29)               OLD_CONPOINTER := CONPOINTER;
30)               OLD_PROGPOINTER := PROGPOINTER;
```

Table 54 continued

```
31)                [Call MUTANT to create a mutation of program ..
                    residing in i^th memory cell. Call NEW_PROG ..
                    to register the new program type created by MUTANT ..
                    and expand the precedence matrix by calling NEW_CONFLICT.];
32)                comment *** CONPOINTER and PROGPOINTER are set .. ;
                           to point to the new precedence matrix ..
                           and memory structure, respectively ***;
33)                if [cell W is empty ]
34)                then
35)                begin
36)                   comment **** write mutant to memory *** ;
37)                   [Write mutant into w^th cell];
38)                   NEW_ST;
39)                   IS_COPY := true ;
40)                end
41)                else
42)                begin
43)                   comment *** w^th cell occupied *** ;
                      [Use precedence matrix to decide whether ..
                       mutant can overwrite program in w^th cell];
44)                   comment **** see (v) below *** ;
45)                   if [cannot overwrite ]
46)                   then
47)                   begin
48)                      comment *** delete mutant, restore old pointers ***
49)                      M := M-1;
50)                      CONPOINTER := OLD_CONPOINTER;
51)                      PROGPOINTER := OLD_PROGPOINTER;
52)                   end
53)                   else
54)                   begin
55)                      [Write mutant to w^th cell];
56)                      NEW_ST;
57)                      IS_COPY := true ;
58)                   end
59)                end
60)             end
61)          end
62)          else
```

Table 54 continued

```
63)          begin
64)            comment *** program in i^th cell generates
                       correct non-mutated copy ***;
65)            if [cell W is empty ]
66)            then
67)            begin
68)              comment **** no obstacle to copying *** ;
69)              [Write copy into w^th cell] ;
70)              IS_COPY := true ;
71)            end
72)            else
73)            begin
74)              comment **** w^th cell occupied *** ;
                 [Use precedence matrix to decide whether a copy of program
                  residing in i^th cell can overwrite program in w^th cell];
75)              comment **** see (v) below *** ;
76)              if [cannot overwrite];
77)              then comment **** nothing happens *** ;
78)              else
79)              begin
80)                [Write copy to w^th cell];
81)                IS_COPY := true;
82)              end
83)            end
84)          end;
85)          comment *** If the program residing in the i^th memory cell ..
                     was able to write his copy/mutation to memory, then ..
                     TIMECOUNT of the w^th cell has to be updated ..
                     consistent with the memory read direction ..
                     (insert Table 8.12, lines 34)-48)) ***;
86) end      *** M A T C H ***;
```

Table 55 MOD3: Input parameters

Initial size of memory	N
Number of different program types	M
M program types, characterized by size and initial count	DELY, STPOINTER.S[...]
M*M elements of the precedence matrix, values ranging from (0,1000]	CONPOINTER.MAT
Mutation probability	PROB_MUT $\in [10^8]$
Probability of lethal mutation	PROB_LETAL $\in [10^6]$
Probability of mutation type	PROB_MUT $\in [10^3]$
Planned memory cycle count	TIME
Memory parameters	MORE, PERCENT

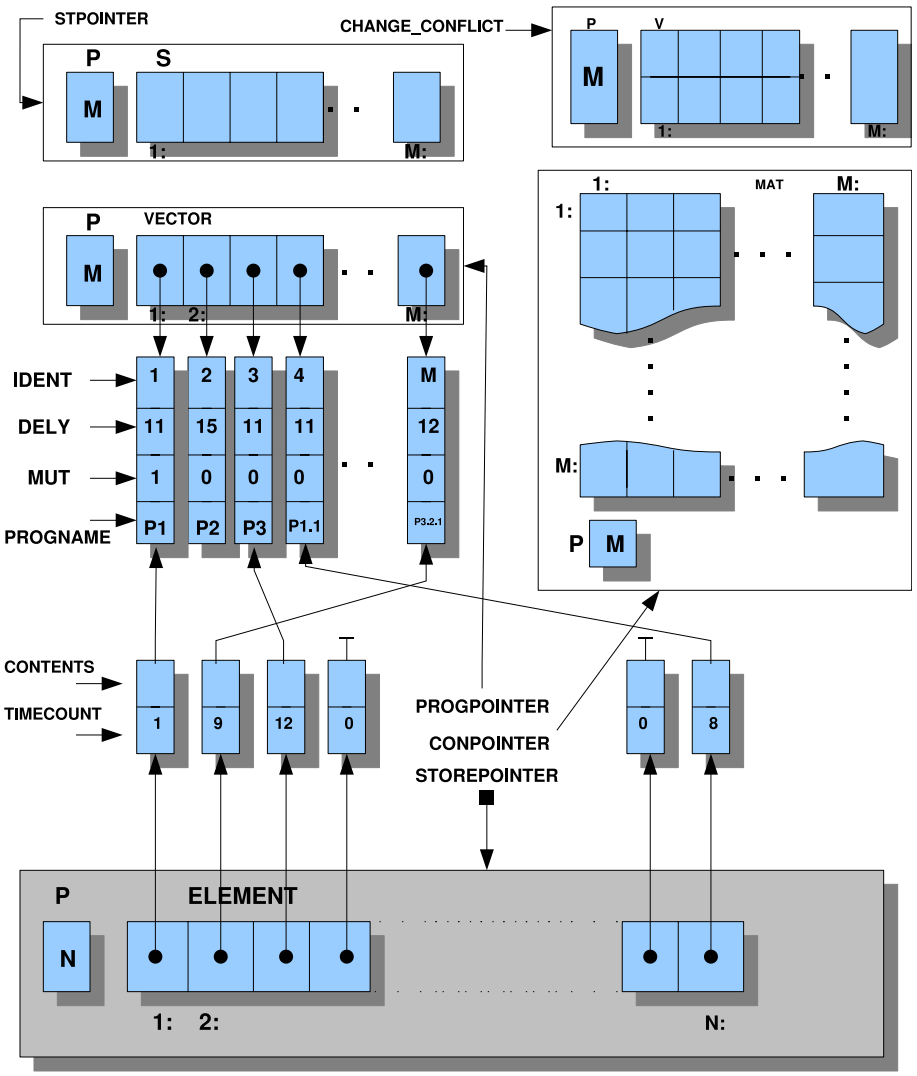


Fig. 15 MOD3: Global data structures

Table 56 MOD3: Memory and runtime complexity

Procedure	Complexity for large M
MUTANT	$O(M)$
CREATE_NAME	$O(1)$
NEW_PROG	$O(M)$
NEW_ST	$O(M)$
NEW_CONFLICT	$O(M^2)$
M = Current program type count	

References

1. Beilner, H.: Betriebssysteme. Vorlesungsskript, Universität Dortmund, Winter (1976/1977)
2. von Bertalanffy, L.: General Systems Theory. George Braziller, New York (1968)
3. Brainerd and Landweber: Theory of Computation. Wiley, NY (1974)
4. Claus, V.: Rekursive Funktionen. Vorlesungssbegleitmaterial, Universität Dortmund, Winter (1974/1975)
5. Ehrich, D.: Berechenbarkeit. Vorlesungsskript, Universität Dortmund, Winter (1977/1978)
6. Geschwind, H.W.: Design of Digital Computers—an Introduction. Springer, New York (1967)
7. Hoffmann, G.: Programmiersprachen und ihre Übersetzer. Vorlesungssbegleitmaterial, Universität Dortmund, Sommer (1977)
8. Holland, J.: Automata, Languages, Development, chapter Studies of the spontaneous emergence of self-replicating systems using cellular automata and formal grammars. North-Holland Publishing Company, Amsterdam (1976)
9. Hopcroft, U.: Formal Languages and their Relation to Automata. Addison-Wesley, Reading (1969)
10. Jensen, K., Wirth, N.: Pascal User Manual And Report, 2nd edn. Springer, New York (1978)
11. Kaplan, R.W.: Der Ursprung des Lebens, 2nd edn. Georg Thieme Verlag, Stuttgart (1978)
12. Kästner, H.: Rechnerfeinstrukturen. Vorlesungsskript, Universität Dortmund, Sommer (1976)
13. Kästner, H.: Architektur und Organisation digitaler Rechenanlagen. Teubner Stuttgart (1978)
14. Lee, J.A.N.: Computer Semantics. Van Nostrand Reinhold Company, New york (1972)

15. Linder, H.: Biologie. J.B. Metzlerische Verlagsbuchhandlung Stuttgart
16. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill, NY (1974)
17. Reusch, B.: Grundlagen der theoretischen Informatik. Vorlesungsbegleitmaterial, Universität Dortmund, Sommer (1977)
18. Rogers, H. Jr.: Theory of Recursive Functions and Effective Computability. McGraw-Hill, NY (1967)
19. Rohlfing, H.: SIMULA—Eine Einführung, vol. 747. B.I. Hochschultaschenbücher
20. Schnorr, C.P.: Rekursive Funktionen und ihre Komplexität, vol. 24. Teubner Studienbücher
21. Schnupp, P.: Rechnernetze—Entwurf und Realisierung. De Gruyter, Berlin (1978)
22. Siemens Aktiengesellschaft. Siemens-System 7000 Beschreibung und Befehlsliste (1976)
23. Siemens Aktiengesellschaft. Siemens-System 7000 Siemens-System 4004 Betriebssystem BS 1000 F-Assembler Betriebssystem BS2000 Assembler Beschreibin (1977)
24. Siemens Aktiengesellschaft. SIMULA Programmer's Guide Siemens-System BS 2000 (1978)
25. Siewing, R. (ed.): Evolution. Gustav Fischer, Stuttgart (1978)
26. Stone, H.S. (ed.): Introduction to Computer Architecture. SCIENCE RESEARCH ASSOCIATES INC. (1975)
27. Vogel and Angermann. DTV-Atlas zur Biologie, vol. 2. Deutscher Taschenbuch Verlag (1968)
28. Wirth, N.: Algorithmen und Datenstrukturen. Teubner Studienbücher