Summary

This paper attempts to explore the logical foundations of computer programming, by use of techniques which were first applied in the study of geometry, and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuit of these topics.

1. Introduction

Computer programming is an exact science, in that all the properties of a program and all the consequences of executing it in any given environment can be found out from the text of the program itself, by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underly our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustration purposes, this paper confines itself to a very simple language, which is effectively a subset of all current procedure-oriented languages, and yet is theoretically as powerful as any of them, in the sense of being able to program any computable function.
1. \text{true \{r := x \} Fr_o (r = x \land \text{true})}

2. \text{Fr_o (r = x \land \text{true}) \{q := 0\} \forall q_o (q = 0 \land Fr_o (r = x \land \text{true}))}

3. \exists q_o (q = 0 \land Fr_o (r = x \land \text{true})) \supset x = r + y \times q

4. \text{Fr_o (r = x \land \text{true}) \{q := 0\} x = r + y \times q}

5. \text{true \{(r := x; q := 0)\} x = r + y \times q}

6. \text{x = r + y \times q \land y \leq r \{r := r - y\} Fr_o (r = r_o - y \land x = r_o + y \times q \land y \leq r_o)}

7. \exists q_o (q = 1 + q_o \land Fr_o (r = r_o - y \land x = r_o + y \times q \land y \leq r_o)) \supset x = r + y \times q

8. \exists q_o (q = 1 + q_o \land Fr_o (r = r_o - y \land x = r_o + y \times q \land y \leq r_o)) \supset x = r + y \times q

9. \exists q_o (r = r_o - y \land x = r_o + y \times q \land y \leq r) \{q := 1 + q\} x = r + y \times q

10. \text{x = r + y \times q \land y \leq r \{(r := r - y; q := 1 + q)\} x = r + y \times q}

11. \text{x = r + y \times q \{while \ y \leq r \ do \ (r := r - y; q := 1 + q)\} \neg y \leq r \land x = r + y \times q}

12. \text{true \{(r := x; q := 1 + q)\} \{while \ y \leq r \ do \ (r := r - y; q := 1 + q)\} \neg y \leq r \land x = r + y \times q}

\text{Notes.}

1. The left-hand column is used to number the lines, and the right-hand column to justify each line, by appealing to an axiom, \textit{Lemma}, or a rule of inference applied to one or two previous lines, indicated in brackets. Neither of these columns is part of the formal proof.

2. For example, line 4 is an instance of the axiom of assignment (DO), line 12 is obtained from lines 5 and 11 by application of the rule of composition (D2).

3. Lemma 2 may be proved from axioms A12 and A13.

\text{Table 3.}
2. Computer Arithmetic.

The first requirement in valid reasoning about a program is to know the properties of the elementary operations which it invokes, for example, addition and multiplication of integers. Unfortunately, in several respects computer arithmetic is not the same as the arithmetic familiar to mathematicians, and it is necessary to exercise some care in selecting an appropriate set of axioms. For example, the axioms displayed in Table 1 is within a small selection of axioms relevant to integers.

<table>
<thead>
<tr>
<th>Table 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1.</td>
<td>$x + y = y + x$</td>
</tr>
<tr>
<td>A2.</td>
<td>$xxg = xgx$</td>
</tr>
<tr>
<td>A3.</td>
<td>$(x + y)z = x(z + y)$</td>
</tr>
<tr>
<td>A4.</td>
<td>$(xxg)z = x(xz)$</td>
</tr>
<tr>
<td>A5.</td>
<td>$x(y + z) = x(y + z)$</td>
</tr>
<tr>
<td>A6.</td>
<td>$y \leq x \Rightarrow (x - y) + y = x$</td>
</tr>
<tr>
<td>A7.</td>
<td>$x + 0 = x$</td>
</tr>
<tr>
<td>A8.</td>
<td>$x \times 0 = 0$</td>
</tr>
<tr>
<td>A9.</td>
<td>$x \times 1 = x$</td>
</tr>
</tbody>
</table>

From this incomplete set of axioms it is possible to deduce such simple theorems as:

$x = x + y \times 0$

$\forall x, \exists y, x + y \times 0 = (x - y) + y \times (1 + y)$

The proof of the second of these is:

$(x - y) + y \times (1 + y) = (x - y) + (y \times 1 + y \times y) = (x - y) + (y + y \times y) = (x - y) + y + y \times y = x + y \times y \times 0$ provided $y \leq x$  

The axioms A1 to A9 are, of course, some of the traditional infinite set of integers in mathematics. However, they are also some of the finite sets of "integers" which are manipulated by computers. Their truth is independent of the size of the set, and furthermore, it is largely independent of the choice of technique applied in the event of "overflows"; for example:

provided that they are confined to non-negative numbers.
(1) The strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Form boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated by addition and multiplication tables for a trivially small model, in which 0, 1, 2, and 3 are the only integers represented.

(1) the strict interpretation

```
|   | 0 | 1 | 2 | 3 |
-|-|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 3 | * |
| 2 | 2 | 3 | * | * |
| 3 | * | * | * | * |
```

```
|   | 0 | 1 | 2 | 3 |
-|-|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 |
| 2 | 2 | 0 | 2 | * |
| 3 | 3 | 0 | 3 | * |
```

* nonexistent

(2) form boundary

```
|   | 0 | 1 | 2 | 3 |
-|-|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 3 | 3 |
| 2 | 2 | 3 | 3 | 3 |
| 3 | 3 | 3 | 3 | 3 |
```

```
|   | 0 | 1 | 2 | 3 |
-|-|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 |
| 2 | 2 | 0 | 2 | 3 |
| 3 | 3 | 0 | 3 | 3 |
```

(3) modulo arithmetic

```
|   | 0 | 1 | 2 | 3 |
-|-|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 3 | 0 |
| 2 | 2 | 3 | 0 | 1 |
| 3 | 3 | 0 | 1 | 2 |
```

```
|   | 0 | 1 | 2 | 3 |
-|-|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 |
| 2 | 2 | 0 | 2 | 0 |
| 3 | 3 | 0 | 3 | 2 |
```

It is interesting to note that in the three different systems satisfying axioms A1 to A9, may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

\[ A10 \_ I \rightarrow \exists x \, \forall y \,(y \leq x) \]

whereas all finite arithmetic satisfy:

\[ A10 \_ F \rightarrow \forall x \,(x \leq \text{max}) \]

where “max” denotes the largest integer represented.
Similarly, the three treatments of overflow may be distinguished by a choice of one of the following axioms, relating to the value of max + 1:

\[ \text{All}_s : \neg \exists x (x = \text{max} + 1) \quad (\text{strict interpretation}) \]

\[ \text{All}_b : \text{max} + 1 = \text{max} \quad (\text{from boundary}) \]

\[ \text{All}_m : \text{max} + 1 = 0 \quad (\text{modulo arithmetic}) \]

Having selected one of these axioms, it is possible to use it in deducing the properties of programs; however, these properties will not necessarily obtain, unless the program is executed on an implementation which satisfies the chosen axiom.

3. Program Execution.

As mentioned above, the purpose of this study is to provide a logical basis for proofs of the properties of a program. One of the most important properties of a program is whether or not it carries out its intended function. The intended function of a program, or part of a program, can be specified by making general assertions about the values which variables will take after execution of the program. These assertions will usually not ascribe particular values to each variable, but will rather specify certain (properties of the values, and the relationships holding between them.

We shall use the usual notation of mathematical logic to express these assertions. The only symbols which may be novel to programmers familiar with an ALGOL-like language are the quantifiers:

\[ \forall x. P \quad (P \text{ is true for all } x) \]

\[ \exists x. P \quad (\text{there is an } x \text{ such that } P) \]

In many cases, the validity of the results of a program (or part of a program) will depend on the values taken by the variables before that program is initiated. These initial preconditions of successful use can be specified by the same type of general assertion as is used to describe the results obtained on termination. In order to state the required connection between a precondition \( P \), a program \( P \),
and a description of the result of its execution (R), we introduce a new notation:

\[ P \{ Q \} R \]

This may be interpreted "If the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion."

If there are no preconditions imposed, we write \[ \text{true} \{ Q \} R \].

### 3.1 Axiom of assignment

Assignment is undoubtedly the most characteristic feature of programming languages, and one that most clearly distinguishes practical programming from other branches of mathematics. However, the notion governing assignment is rather complex, and will be introduced here in a gradual fashion.

Consider the assignment statement

\[ x := f \]

where \( x \) is an identifier for a simple variable,

\( f \) is an expression of a programming language

without side-effects.

Let us suppose first of all that the variable \( x \) does not enter into the expression \( f \). Then it is obviously true to state:

\[ \text{true} \{ x := f \} x = f \quad (\text{where } f \text{ does not contain } x) \quad (1) \]

Furthermore, if \( P \) is any assertion which does not mention the variable \( x \) at all, the assignment to \( x \) cannot affect the truth of \( P \), so we get:

\[ P \{ x := f \} x = f \land P \quad (\text{where } f \text{ and } P \text{ do not contain } x) \quad (2) \]

Now let us consider an assertion \( P(x) \), which does say something about the value of \( x \). Thus when a new assignment is made to \( x \), \( P \) will be no longer true of \( x \). However, \( P \) is still true of the previous value of \( x \), which we shall denote \( x_0 \). Thus, at least \( f \) must satisfy a value satisfying \( P \), and we write:

\[ P(x) \{ x := f \} \exists x_0 (x = f \land P(x_0)) \quad (\text{where } f \text{ does not contain } x) \quad (3) \]

Finally, if we allow \( f \) also to contain occurrences of \( x \), we must remember that in computing \( f(x) \), it is the previous value of \( x \) which is
Thus we must change the \( x \) in \( f(x) \) into \( x_0 \), as we did in the case of \( P(x) \). Thus the axiom assumes the form

\[
P(x) \{ x := f(x) \} \ \exists x_0 \ (x = f(x_0) \land P(x_0))
\]  

(4)

It appears that (4) is the most general result obtainable, and that the previous three assertions can be directly deduced from it as special cases. Giving a more formal description of (4) we obtain:

\[
\text{DO Axiom of Assignment} \\
\vdash P \{ x := f \exists x_0 \ (x = f_0 \land P_0) \\
\text{where } x \text{ is a variable} \\
f \text{ is an expression} \\
x_0 \text{ is a variable not free in } f \text{ or } P \\
f_0 \text{ and } P_0 \text{ are obtained from } f \text{ and } P \\
\text{by substituting } x_0 \text{ for all free occurrences of } x.
\]

It may be noticed that DO is not really an axiom at all, but rather an axiom schema, describing an infinite set of axioms which share a common pattern. This pattern is described in purely syntactic terms, and it is very easy to check whether any finite text conforms to the pattern, thereby qualifying as an axiom, which may validly appear in any line of a proof.

3.2 Rule of Consequence.

In addition to axioms, a deductive system requires at least one rule of inference, which permits the deduction of new theorems from one or more axioms or theorems already proved. A rule of inference takes the form "If \( X \) and \( Y \) then \( Z \)", i.e., if assertions of the form \( X \) and \( Y \) have been proved as theorems, then \( Z \) also is thereby proved as a theorem.

The simplest example of an inference rule states that:

- If the execution of a program \( Q \) ensures the truth of the assertion \( P \), then it also ensures the truth of every assertion logically implied by \( P \).

This can be more formally expressed:

- **Rule of Consequence**
  
  \[ Q \text{ if } \vdash P \text{ and } \vdash Q \text{ then } \vdash P \implies Q \]

It is this rule which makes it possible to deduce the special cases from the axiom of assignment.
3.3. Rule of Composition

A program generally consists of a sequence of statements which are executed one after another. The statements may be separated by a semicolon or equivalent symbols \((Q_1; Q_2; \ldots; Q_n)\). In order to avoid the awkwardness of dots, it is possible to deal initially with only two statements \((Q_1; Q_2)\), since larger sequences can be reconstructed by nesting \((Q_1; (Q_2; \ldots; (Q_{m-1}; Q_m))),\) The removal of the brackets may be regarded as a convention based on the associativity of the ";" operator, in the same way as brackets are removed from an arithmetic expression \(\downarrow; (x + (y + (\ldots (z + w) \ldots)))\).

The inference rule associated with composition states that if the final result of the first part of a program is identical with the precondition under which the second part of the program produces its intended result, then the whole program will produce the intended result, provided that the precondition of the first part is satisfied. In more formal terms:

\[
\text{D2 Rule of Composition: } \quad \text{if } P \{Q_1; Q_2\} R_1 \text{ and } R_1 \{Q_2\} R \text{ then } P \{Q_1; Q_2\} R
\]

3.4. Rule of Iteration

The essential feature of a stored-program computer is the ability to execute some portion of program \(S\) repeatedly until a condition \(B\) goes false. A simple way of expressing such an iteration is to adopt the ALGOL 60 while notation:

\[
\text{while } B \text{ do } S
\]

In executing this statement, a computer first tests the condition \(B\). If \(B\) is false, \(S\) is omitted, and execution of the loop is complete. Otherwise, \(S\) is executed and \(B\) is tested again. This action is repeated until \(B\) is found to be false.

The reasoning which leads to a formulation of an inference rule for iteration is as follows. Suppose \(P\) to be an assertion which is always true on completion of \(S\), provided that it is also true on initiation. Then obviously \(P\) will still be true after any number of iterations of the statement \(S\) (even no iterations). Furthermore, it is known that the controlling condition \(B\) is false when the iteration finally terminates. A slightly more powerful formulation is possible in light of the fact that \(B\) may be assumed to be true on initiation of \(S\):

\[
\text{D3 Rule of Iteration: } \quad \text{if } P \{S\} P \text{ then } P \{\text{while } B \text{ do } S \} B \Rightarrow P
\]
3.5. Example.

The axioms quoted above are sufficient to construct the proof of properties of simple programs. For example, a routine intended to find the quotient $q$ and remainder $r$ obtained on dividing $x$ by $y$. All variables are assumed to range over a set of non-negative integers, conforming to the axioms listed in Table 1. For the sake of simplicity, we use the trivial but inefficient method of successive subtraction. The proposed program is:

$$( r := x; q := 0); \text{ while } y \leq r \text{ do } ( r := r - y; q := q + 1)$$

An important property of this program is that when it terminates, we can recover the numerator $x$ by adding to the remainder $r$ the product of the divisor $y$ and the quotient $q$ (i.e., $x = r + y \cdot q$). Furthermore, the remainder $r$ is less than the divisor. These properties may be expressed formally:

$$\text{true } \{ \theta \} \rightarrow \forall x \cdot (x = r + y \cdot q)$$

where $\theta$ stands for the program displayed above.

A formal proof of this theorem is given in Table 3. Like all formal proofs, it is excessively tedious; and it would be fairly easy to introduce notational conventions which would significantly shorten it. An even more powerful method of reducing the tediousness of formal proofs is to devise 'general rules' for proof construction out of the simple rules accepted as postulates. These general rules would be shown to be valid by demonstrating how every theorem proved with their assistance could equally well (if more tediously) have been proved without. Once a powerful set of supplementary rules has been developed, a 'formal proof' reduces to little more than an informal indication of how a formal proof could be constructed.
When stapling, please put the tables at the end of the text.

Table 1

insert 11 multiplication signs
one = sign (A6)
The second 1 in A9 should be deleted.

Table 2

insert 3 multiplication signs.

Table 3

line 1 insert \( \land \) sign (upside-down \( \lor \) well do)

2 insert 3 \( \land \) signs

3 insert 2 \( \lor \) signs and \( \land \) and \( \lor \)

4 insert \( \land \) and \( \lor \)

5 insert \( \lor \)

6 insert 3 \( \land \) signs and 2 \( \lor \)

7 insert 5 \( \land \) signs and 2 \( \lor \); also pull back
   the go on the last line.

8 \( \land \) and 2 \( \lor \) and \( \lor \)

9 \( \land \) and 2 \( \lor \)

10 \( \land \) and \( \lor \)

11 1 \( \land \) and 2 \( \lor \) and are \( \lor \); also change 8 to 10

12 1 \( \land \) and 1 \( \lor \) and are \( \lor \)
page 1
9 x signs and one $C_x$

page 2
lines 17-19 please move left to the margin. Also capital T for "table"
line 22 delete "in"
line 35 put "c" for "s"
line 27 insert $\frac{b}{(1+z^3)}$
line 37 insert $\pi$

page 3
insert 3 $\Lambda$ signs $\times$

page 4
line 8 insert $+$ and $\Lambda$ $\times$
line 23 should be "text continues"
line 30 delete "can" from "cannot"
line 32 3 $+$ signs $\times$
line 41 3 $+$ signs and $\Lambda$ $\times$
line 55 insert ", thus:

page 5
line 7 3 $+$ signs $\times$
carry back the "Q2)3R" from next line
line 36 two $+$, two $\Lambda$ and one $\rightarrow X$
53 $\times$ $\times$
56 $\Lambda$ and $\times$ $\times$

page 7
line 51 insert "at end" and
line 65 replace "ul" by "at"
\underline{If should read "annotated"}

page 9
line 15 write insert "s" after "provide"
Page 9  line 50 at 51
should read:

5. R.M. Burstall  Proving properties of programs
by structural induction.

Experimental Programming Reports: No. 17 (Feb 1968)
DMIP Edinburgh.
4. General Reservations

The axioms and rules of inference quoted in this paper have implicitly assumed the absence of side effects or the evaluation of expressions and conditions. In proving properties of programs expressed in a language permitting side effects, it would be necessary either to prove their absence or to apply an appropriate proof technique, or else translate all function calls of the program into procedure calls with explicit sequencing. If the main purpose of a high-level programming language is to assist in the construction and verification of correct programs, it is doubtful whether the use of functional notation to call procedures with side effects is a genuine advantage.

Another deficiency in the axioms and rules quoted above is that they give no basis for a proof that a program successfully terminates. Failure to terminate may be due to an infinite loop; or it may be due to violation of an implementation-defined limit, for example, the range of numeric operands, the size of storage, or an operating system timeout. Thus, the notation "P terminate" should be interpreted as "if the program successfully terminates, the properties of its results are described by P". It is fairly easy to adapt the results of the axioms so that they cannot be used to predict the results of non-terminating programs; but the actual use of the axioms would now depend on knowledge of many implementation-dependent features, for example, the size and speed of the computer, the range of numbers, and the choice of overflow technique. Apart from proofs of the avoidance of infinite loops, it is probably better to prove the "conditional correctness" of a program, and rely on an implementation to give a warning if it has led to abnormal execution of the program as a result of violation of an implementation limit.

Finally, it is necessary to deal with some of the areas which have not been covered, for example, real arithmetic, bit and character manipulation, complex arithmetic, functional arithmetic, arrays, records, overlays, definitions, files, input/output, declarations, ... There seems to be no other way of presenting them. The programming language is kept simple. Areas which do present real difficulties are labels and jumps, pointers, and name parameters. Proofs of programs which make use of these features are likely to be elaborate, and it is not surprising that this should be reflected in the complexity of the underlying axioms.

In this manner, the characterization of integer arithmetic is far from complete.
5. Proof of Program Correctness

The most important property of a program is whether it accomplishes the intentions of its user. If these intentions can be described rigorously by making assertions about the values of variables at the end (or at intermediate points) of the execution of the program, then the techniques described in this paper may be used to prove the correctness of the program, provided that the implementation of the programming language conforms to the axioms and rules which have been used in the proof. This fact may also be re-established, by deductive reasoning, using an axiom set which describes the logical properties of the hardware circuitry. When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics. Even electronic reliability is likely to increase if the engineer no longer has the excuse of program or software errors to avoid dealing with occasional errors.

... (because widespread until considerably more powerful proof techniques became available, and even then will not) ... (became widespread until considerably more powerful proof techniques became available, and even then will not) ... be easy. But the practical advantages of program proving will eventually outweigh the difficulties, in view of the surprising costs of programming errors. At present, the method which a programmer uses to convince himself of the correctness of his program is to try it out on particular cases, and to modify it if the results produced do not correspond to his intentions. After he has found a reasonably wide variety of example cases on which the program seems to work, he believes that it will always work. The advice repeated in this...
and implement and expensive surprise can result when attempting to transfer it to another machine.
will then have the choice of formulating his algorithm in a machine-independent fashion, possibly with the help of environment enquiries; or if this involves too much effort or inefficiency, he can deliberately construct a machine-dependent program, and rely for his proof on some machine-dependent axiom, for example, one of the versions of A11 (Section 9).

In the latter case, the axiom must be explicitly quoted as one of the preconditions of successful use of the program. The program can with complete confidence be transferred to any other machine which happens to satisfy some machine-dependent axiom; but if it becomes necessary to transfer it to an implementation which does not, then all the places where changes are required will be clearly annotated by the fact that the proof at that point appeals to the truth of the machine-dependent axiom.

Thus this practice of proving programs would seem to lead to solution of some of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving certainly at present, will certainly be difficult even for programmers of high calibre, and may be applicable only to quite simple program designs. As in other areas, reliability can be purchased for a price, only at the price of simplicity.


A high-level programming language, such as PL/1, FOCAL, COBOL, is usually intended to be implemented on a variety of computers of differing size, configuration, and design. It has been found a good practice to define these languages with sufficient vigour to ensure sufficient rigor to define these languages with sufficient vigour. Since the purpose of compatibility among all implementations. Since the purpose of compatibility is to facilitate interchange of programs expressed in the same language, one way to achieve this would be to insist that all implementations of the language shall "satisfy" the axioms and rules rules of inference which underlie proofs of the properties of programs expressed in the language. So that all specifications based on these proofs will be falsifiable. Except in the event of hardware failure, of course.

Apart from giving an immediate and possibly direct approach to formal language definition is that it aims to be the simplest and most readily comprehensible of all techniques proposed for the purpose. It bears the same relationship to an algorithm for implementing a language as the criterion of program concreteness does to a program itself. Thus it is the only known type of description, which is equally suitable...
Apart from giving an immediate, and possibly even provable, criterion for the correctness of an implementation, the axiomatic technique for the definition of programming language semantics appears to be like the formal syntax of the ALGOL '60' report, in that it is sufficiently simple to be understood both by the implementor and by the reasonably sophisticated user of the language. It is only by bridging this widening communication gap in a single document (perhaps even provably consistent) that the maximum advantage can be obtained from a formal language definition.

In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.
design of better programming language 1.6: a system for

Another interesting point is that a language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodology for developing programming languages. The methodology includes a general approach to developing programming languages. The language designer may be led to similar conclusions on the basis of a general methodolo
The importance of program proof is clearly emphasised in [9], and an informal technique for providing them is described. The suggestion that the specification of formal proof techniques provide an adequate formal definition of a programming language first appears in [8]. The formal treatment of program execution presented in this paper is clearly derived from Floyd. The main contributions of the author appear to be:

1. A suggestion that axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined.
2. A comprehensive evaluation of the possible benefits to be gained by adopting this approach both for program proving and for formal language definition.

However, the formal material presented here has only an expository status, and represents only a minute proportion of what remains to be done. It is hoped that many of the fascinating problems involved will be taken up by other more skilled hands.
    Kybernetica I Moscow 1958

2. S. Igarashi  An Axiomatic Approach to Equivalence Problems

3. T.W. de Bakker  Axiomatics of Simple Assignment Statements.

4. J. McCarthy  Towards a Mathematical Theory of Computation

5. R. Burstall

    BIT (1966) Vol 6 pp 66-81

7. J. Laski  Sets and Other Types
    Algol Bulletin 24 (1968)

8. R.H. Floyd  Assigning Meanings to Programs

9. P. Naur  Proof of Algorithms by General Snapshots
    BIT (1966) Vol 6 pp 310-316