REPORT ON THE PROGRAMMING NOTATION 3R

Andrew P. Black

Technical Monograph PRG-17
August 1980

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
OXFORD, OX2 6PE

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
Oxford, OX2 6PE

Contents

Foreword (by Brian Shearing)

If an engineer designs a bridge using a computer program written by someone else, the person responsible if the bridge falls down is the engineer, not the author of the program. Similar statements can be made in most professions. It is essential that responsible users of programs should be able to inspect them and be satisfied that they are sound and applicable.

The programming languages of today are so poor that few programmers can understand another programmer's work, or even their own work after a few months; it is not reasonable to expect an intelligent user to understand it.

The traditional response to this (and other) problems has been to design high-level languages of increasing power. In contrast, this report is part of a continuing experiment to discover programming notations whose emphasis is on simplicity rather than power for its own sake. The eventual aim of the experiment is that a program should be understandable not just to other programmers but also to those with only a layman's knowledge of the essentials of programming.

The first published version of 3R (December 1977) was created by Alcock Shearing & Partners to fulfil a contract with the Design Office Consortium†. The requirement was the production of a "publishable program" for use in the Construction Industry. That program - the Forpa Program - is published as a book in which the first part "describes the notation used throughout the rest of the manual" [10]. This notation is 3R. It was most encouraging that the Programming Research Group at Oxford took an interest in 3R. Andrew Black has here produced a concise but rigorous definition of the syntax and semantics of a notation capable of describing non-trivial programs with great clarity.

† The Design Office Consortium is an association supported by the Departments of Environment and Industry, and aims to encourage the use of computers in the Building Industry.

We have been using an evolving 3R for, amongst other things, a program of 20 000 Fortran statements [11], and we are confident that the program does what it should. We are also confident that we could convince a modestly well informed user that the program does what it should, simply by "reading him through" the 3R description.

The idea of a notation based on simplicity has been presented informally to several conferences and also in writing [1] [8]. The response to these presentations, our own use of the notation and the work at Oxford have resulted in some six dialects. The Oxford dialect presented here is the most rigorously defined and makes the fewest concessions to translation into currently available computer languages.

Much remains to be done. At the moment the two characters "3R" stand for an idea. The idea is taking shape. How it will turn out no one can tell. But this report is an important milestone in its development.

Brian Shearing,
Alcock Shearing & Partners.

0. Introduction

The first version of 3R (mentioned in the Foreword) was described very briefly, but in sufficient detail to make the Forpa Program unambiguous. However, some parts of the language were left in need of clarification; we hope this report provides it. In the process of completing the definition further simplifications have been made; the sequel defines the Oxford version of the notation as it stood in November 1978.

0.1. On Implementations

3R is a notation for describing solutions to problems which require the use of computing machinery. The use of some formal notation is necessary because a process must be rigorously defined before it can be mechanised. It should not be inferred, however, that the process definition, i.e. the program, must be in a form which can be used directly to instruct a machine. The main part of the programming problem is solved once the program is written in a machine independent, easily understandable notation such as 3R.

There remains the problem of *coding*, of creating a realisation of the program (in some computer language) which can be used to control the hardware. The Forpa Program was transliterated into Fortran by hand, and realisations in other computer languages are under construction.

However, because 3R is a very simple notation and programs in it specify every detail of the problem solving procedure, there is no reason why this transliteration should not be mechanised. The resulting code may not be as efficient as that produced by an experienced coder, but this is becoming a less important consideration as computer hardware becomes cheaper. So feel free to write a compiler for 3R.

The important point is the converse, however: lack of a compiler does not limit the usefulness of 3R in any serious way. It has been and will be used in real, large scale, projects. The separation of these projects into a programming and a coding stage facilitates a useful separation of concerns. When writing in 3R attention can be concentrated on the problem and the algorithm used to solve it. Considerations of efficiency can be dealt with later, when realising the program in some more machine oriented computer language.

0.2. An Overview

The name 3R is the well-known acronym for reading, writing and arithmetic, and summarises the main features of the notation. It is designed for readability; ease of writing comes a poor second. 3R is distinguished by its lack of "features" and novel ideas: it has been produced by extracting common factors from other current programming languages, designing a uniform notation for them, and ruthlessly throwing out any constructs which were obscure, ambiguous, dangerous or unnecessary. Those that remain form the minimal set necessary for writing large programs, or so we conjecture; the object of publishing this definition is to enable this conjecture to be validated by wide-ranging experiment. The most obvious attributes of the notation are:

(i) Support for program development by stepwise refinement [12];
(ii) Acceptance of Dijkstra's alternative construct [2];
(iii) Avoidance of defaults [7];
(iv) Absence of a loop construction [5].

These features are now discussed in a little more detail.

(i) 3R does not have an Algol-like structure of nested blocks. Instead the structure is "flat": if any complicated action is required it is necessary to invent a name for that action and later define, or rather *refine*, the name. In this way the programmer is encouraged to make the design process obvious from the text. It also becomes unnecessary for a reader of the program to have an arbitrarily deep stack of definitions in his head: instead he need remember exactly two levels. Although there will be more names visible at each level, the result seems to be more readable than conventional block structure.

(ii) The Choice Command of 3R is closely modelled on Dijkstra's alternative construct, and thus allows non-determinism. In our notation Dijkstra's classic example becomes

<u>*where*</u> *set maximum* <u>*is*</u>
    <u>*if*</u> $x \leq y$
        $maximum := y$
    <u>*if*</u> $x \geq y$
        $maximum := x$
    <u>*otherwise*</u> <u>*chaos*</u>

(iii) Defaults may make programs easier to write, but the price paid when they later come to be read is unacceptably high. We aim for everything to be obvious from the text. That is why Dijkstra's <u>*fi*</u> has been replaced by <u>***otherwise***</u> <u>***chaos***</u>, which we hope implies the consequences of failing to ensure that at least one of the conditions is satisfied.

(iv) There is no loop construction in 3R. Instead, since the piece of program we may require to repeat will invariably have a name, the repetition is obtained simply by using that name. We hesitate to use the term recursion as this has come to imply an implementation in terms of stacks and calls, which is unnecessary and undesirable in most circumstances.

1. The Method of Description

The whole of 3R is defined in this report, which takes the unusual (but in many ways more natural) step of describing the language from the top downwards. First we define the notion of a **program**, in terms of its (as yet undefined) subcomponents; then come the definitions of those subcomponents, and of their subcomponents, until eventually everything is defined in terms of the *lexemes* and *tokens* (q.v. Section 1.2), which are the basic units from which programs are built. Since the structure of 3R is not recursive (except for such things as **expressions**), this report makes only a small number of back references but a large number of forward references. We do not feel that this impairs readability because it is central to the design of 3R that every construct means what it appears to mean. On first acquaintance there is no need to look up the forward references: they are there to reassure rather than to perplex. Similarly we have not hesitated to use as yet undescribed constructions in the examples.

1.1. Technical Terms

In an effort to avoid the excessive use of abbreviations, some long and turgid phrases have been used in the text. However, this has not been taken to extremes, and where necessary a technical term has been introduced, indicated by its name appearing in *italics*. All the technical terms are defined at an appropriate place in the text; the location of any particular definition can be found from the index on page 57.

1.2. Syntactic Description

The syntax will be described by a metalanguage which represents a context-free grammar [3] [6]. This grammar generates a language larger than 3R, and is accompanied by context restrictions expressed in English. As has been indicated above, the terminal symbols of the grammar are *tokens* and *lexemes*.

1.2.1. *The Tokens*

A *token* abstracts from a basic symbol; within the grammar it is easily recognised by its name, which ends with **token**, e.g. **let token**. The representation of a token is meaningless except in so far as the symbol chosen has a mnemonic quality.

Below are listed the representations of the tokens which will be used in this report, and in a few cases suggested alternatives.

token	representation	
abs token	*abs*	
arctangent token	*arctan*	
array token	*array*	
at token	*at*	
be token	*be*	
becomes token	:=	
char of token	*char of*	
close parenthesis token	)	
comma token	,	
conjunction token	∧	*and*
cosine token	*cos*	
degree token	*degree*	
differs from token	≠	
disjunction token	∨	*or*
divided by token	÷	*div*
dummy token	*pass*	*skip*
e to the power of token	*exp*	
end block token	*end of block*	
end choice token	*otherwise chaos*	
end test token	*end of test*	
equals token	=	
exponentiation token	↑	
fail token	*fail*	
failure token	*on failure*	
finish token	*finish*	
if token	*if*	
integer from real token	*integer from real*	
integer token	*integer*	
invariable token	*invariable*	
is at least token	≥	
is at most token	≤	
is greater than token	>	
is less than token	<	
is token	*is*	
length of token	*length of*	
let token	*let*	
log base e token	*ln*	
log base ten token	*log*	
minus token	-	
modulo token	*mod*	
negation token	¬	*not*

newline token

of token	*of*
open parenthesis token	(
over token	/
parameter token	*parameter*
plus token	+
radian token	*radian*
real from integer token	*real from integer*
real token	*real*
result token	*result*
sine token	*sin*
success token	*on success*
test token	*test*
text token	*text*
times token	×
upto token	..
uses token	*uses*
variable token	*variable*
where token	*where*
with token	*with*
zero token	*zero*

1.2.2. *The Lexemes*

A *lexeme* is an abstraction of a class of user-defined objects, each member being similar to but distinct from the others, e.g. integer denotation. The representation of a lexeme is structured, and the structure conveys information.

The following meta-variables are lexemes.

name
name with arguments
text denotation
integer denotation
real denotation

Names are used to label values. The only property required of them is that it be possible to determine if any two names are *identical*. The following are examples of names.

frame
Pattern
move to first month of next year

Names with arguments are used to name and refer to blocks. The arguments are always optional, so the class name with arguments includes the class **name**. Examples:

print['Answer is']
position of [x] in [table]
character [3] of [Heading]
tab to column [7] of [typewriter]
random

The *argument list* of a **name with arguments** is the list of **expressions** (q.v. Section 7) within the brackets. The argument lists of the first four examples are thus

'Answer is'
x *table*
3 *Heading*
7 *typewriter*

whilst ' *random* ' has an empty argument list.

Text denotations, **integer denotations** and **real denotations** are the constants of the language. Examples are

'This is a text denotation'
57
49.35

which mean just what they appear to mean. All the *lexemes* are defined formally in Section 9.2.

1.2.3. *The Productions*

The production rules of the context-free grammar will be presented in the same form as the following examples.

vehicle:
  bus;
  car;
  bicycle;
  lorry.

convoy:
  vehicle, vehicle;
  vehicle, convoy.

safe convoy:
  man with red flag, convoy, man with red flag.

The words in **gothic** type are the symbols of the grammar. The remaining marks are connectives and have the following meanings.
  : means "consists of"
  ; means "or"

, means "followed by"
. means "end of production"

Thus the examples define a **vehicle** as either a **bus**, **car**, **bicycle** or **lorry**, and a **convoy** as a sequence of two or more **vehicles**. A **safe convoy** is a **convoy** preceded and followed by a **man with red flag**.

One definition which is used continuously throughout the syntax of 3R (and logically ought to be given at the end) is given here to avoid unnecessary suspense. It is

empty:

{i.e. the empty sequence of grammatical symbols.}

The Start Symbol of the grammar is **program**. The production defining a given non-terminal symbol can be found using the index on page 57.

1.3. Semantic Description

The semantics of 3R are described with the aid of a notation similar to Weakest Precondition predicate transformers [2]. For those meeting both predicate transformers and 3R for the first time, the combined effect may be a little overwhelming. For this reason the semantics of each construct are given informally when it is first encountered and the predicate tranformers are reserved until Section 8.

1.4. Notation

The meta-linguistic variables are used in the text to denote the objects which can be derived from them; we have allowed ourselves the freedom to capitalise their initial letters and make them plural where this is required by the English language.

Certain passages of this report appear between the braces '{' and '}'. The meaning of the report is unaffected by their presence. {They are included to help the reader understand the intentions and implications of the definition.}

In all the examples, lines starting at the left margin are comments.

2. Programs

2.1. Context-free Syntax

program:
    argument declarations option, program body, newline token,
        finish token.

program body:
    program element, newline token, program body ;
    program element.

program element:
    block;
    program statement.

program statement:
    statement.

2.2. Examples

The following are complete, if trivial, programs.

2.2.1.

```
pass
finish
```

2.2.2.

```
parameter input is integer
result output is integer
invariable three is 3
output := input + three
finish
```

2.3. Context Restrictions

2.3.1. The *global list* of a program is the concatenation of its *parameter list, result list, variable list* and *invariable list* (q.v. Sections 4.2 and 5.1.2.1). It is required that it contain no name more than once. {The *global list* of a program contains all the names declared in its program body and argument declarations option. The requirement ensures that each name means exactly one thing.}

2.3.2. The *block list* of a **program** consists of all the **formal block names** (q.v. Section 3.1) which occur in that **program**; it may not contain two names which *match*.

Two **names with arguments** *match* if, after the **expressions** within the brackets in both names (if there are any) have been deleted the resulting names (including the empty brackets) are *identical*.

2.3.3. No name may appear in both the *global list* and the *block list* of a program.

2.3.4. The *scope* of a name in the *global list* extends from the declaration which introduces it until the end of the program, including **blocks** whose *usage lists* (q.v. Section 3.1.2.2) contain the name, but excluding all other blocks.

2.3.5. The *scope* of a **formal block name** is the whole program, excluding those **blocks** which have an *identical* **name** in their *local lists* (q.v. Section 3.1.2.1).

2.4. Semantics

The meaning of a **program** is obtained from the meaning of the **argument declarations** (if any) and the **program body** of which it is composed.

3. Blocks and Blocklets

3.1. Blocks

A 3R **block** associates a name with some statements in order that these statements may be referenced, by name, from other parts of the program. The statements make up the **block tail**; the other parts of the block supply corroborative detail.

3.1.1. Context-free Syntax

```
block:
        block head, block body, block end.

block head:
        let token, result list option, formal block name, be token,
            newline token.

block body:
        usage list option, argument declarations option, block tail.

block tail:
        statement;
        statement, newline token, block tail.

block end:
        newline token, blocklets option, end block token,
            newline token.

formal block name:
        name with arguments.

result list option:
        empty;
        joined name list, becomes token.

usage list option:
        empty;
        uses token, joined name list, newline token.

joined name list:
        name;
        name, comma token, joined name list.
```

3.1.2. Context Restrictions

3.1.2.1. The *local list* of a block is the concatenation of its *parameter list*, *result list*, *invariable list* and *variable list* (q.v.

Sections 4.2 and 5.1.2.1). It must not contain any name more than once.

3.1.2.2. The ***usage list*** of a **block** consists of all the names in the **usage list option** of that block; if it is **empty** the list is empty. The concatenation of the *local list* and the ***usage list*** is the *name list* of the block: it must not contain any name more than once.

3.1.2.3. The *argument list* of the **formal block name** must be identical in both content and order to the *parameter list* (q.v. Section 4.2) of the block.

3.1.2.4. The **joined name list** of the **result list option** must be identical in both content and order to the ***result list*** (q.v. Section 4.2) of the block.

3.1.2.5. All names in the *usage list* of a **block** must also be in the *global list* of the **program** which contains that **block**.

3.1.2.6. The *scope* of a name in the *local list* extends from its declaration to the end of the block.

3.1.3. Examples

3.1.3.1.

```
let ratio := tan[theta] be
    parameter theta is 0 .. 2×pi
    result ratio is real
    ratio := (sin theta) / (cos theta)
end of block
```

3.1.3.2.

```
let line from [Start Column] to [End Column] be
    parameter Start Column is 1..80
    parameter End Column is 1..80
    variable number of dashes is 0..80
    Tab to [Start Column]
    number of dashes := 1 + End Column - Start Column
    Write dashes
```

Now we need to define "Write dashes"

```
where Write dashes is
    if number of dashes = 0
        pass
    if number of dashes > 0
        Write['_']
        number of dashes := number of dashes - 1
        Write dashes
    otherwise chaos
```

That was a blocklet (see Section 3.2)

```
end of block
```

3.1.4. Semantics

The association between a **formal block name** and its **block body** is permanent and holds everywhere in the **program body**. The **statements** in the **block tail** are executed when required by means of an **invocation** (q.v. Section 6.3). In particular it should be noted that an invocation of a block may textually precede, succeed or be contained in the block itself.

3.2. Blocklets

A 3R blocklet associates a name with some commands. It is thus less general than a block; blocklets do not have arguments or contain declarations.

3.2.1. Context-free Syntax

```
blocklets option:
        empty;
        blocklet, blocklets option.

blocklet:
        where token, blocklet name, is token, blocklet body,
            newline token.

blocklet body:
        command chain, newline token.

command chain:
        command;
        command, newline token, command chain.

blocklet name:
        name.
```

3.2.2. Context Restrictions

The *blocklet list* of a block consists of all the blocklet names which occur in that block. No name may occur more than once in the *blocklet list*, nor may it occur in both the *blocklet list* and the *name list* of the same block.

The *scope* of a blocklet is the whole of the block in which it occurs, including the blocklet itself.

3.2.3. Example

```
        where select a range is
This blocklet will never be used when table at middle = value
        if table at middle < value
            bottom := middle
        if table at middle > value
            top := middle
        otherwise chaos
```

3.2.4. *Semantics*

The **commands** which make up the **command chain** of the blocklet describe some process. The purpose of the **blocklet** is to enable that process to be performed anywhere within its *scope* simply by using the **blocklet name** in a **substitution** (q.v. Section 6.4).

4. Arguments

The arguments of a block provide the means by which it communicates with its environment, that is, the piece of 3R program which invoked the block.

The arguments of a program perform a similar function, but in this case the environment is outside the program. The way in which these values are transferred is thus beyond the scope of this report.

4.1. Context-free Syntax

```
argument declarations option:
        empty;
        argument declaration, newline token,
            argument declarations option.

argument declaration:
        result declaration;
        parameter declaration.

result declaration:
        result token, name, is token, type indicator.

parameter declaration:
        parameter token, name, is token, type indicator.
```

4.2. Context Restrictions

The *parameter list (result list)* of a block or program consists of all the names in the parameter declarations (result declarations) of the argument declarations option of that block or program.

No name may occur more than once in the concatenation of the *parameter list* and the *result list*.

4.3. Examples

result t *is* *integer*
parameter line *is* *text*
parameter Page *is* *array* *zero* .. 66 *of* *text*

4.4. Semantics

4.4.1. Results

A **result declaration** occurring in a **block body**, or directly in a **program**, introduces a **name** in the same way as does a **variable declaration** (q.v. Section 5.1). {Such a **name** may appear to the left of the **becomes token** in a **computation**, and its value may thus be changed.}

If **result declarations** occur outside of all blocks, i.e. directly in a **program**, the output of the program is the list of values of the names. If **result declarations** occur in a **block** the names are used to establish the result of an invocation of that **block**, as described in Section 6.3.4.

4.4.2. Parameters

A **parameter declaration** occurring in a **block body** or directly in a **program** associates a name with a value; the association cannot be changed within the *scope* of the name. Different invocations of a block, or runs of a program, may initialise a **parameter** to different values. The *type* of the value must correspond to the **type indicator** in the declaration. {The name of a parameter may not appear to the left of the **becomes token** in a **computation**.}

If **parameter declarations** occur in a **block**, the values to be associated with the names for the duration of a particular invocation are obtained from the *argument list* of the **actual block name** in that invocation (q.v. Section 6.3.4). If **parameter declarations** occur outside of all blocks, i.e. directly in a **program**, the values associated with the names are the input of the program.

5. Statements

Statements are the primary constituents of 3R programs. They may be **declarations**, which describe the objects the program manipulates, or **commands**, which specify what actions are to be performed on these objects.

```
statement:
        declaration;
        command.
```

5.1. Declarations

5.1.1. Context-free Syntax

```
declaration:
        variable declaration;
        invariable declaration.

variable declaration:
        variable token, name, is token, type indicator.

invariable declaration:
        invariable token, name, is token, expression.

type indicator:
        integer token;
        real token;
        text token;
        subrange indicator;
        array indicator.

array indicator:
        array token, array bound, of token, base type indicator.

array bound:
        zero token, upto token, expression.

base type indicator:
        type indicator.

subrange indicator:
        expression, upto token, expression.
```

5.1.2. *Context Restrictions*

5.1.2.1. The *invariable list (variable list)* of a block or program consists of all the names introduced by **invariable declarations** (**variable declarations**) in its **block body** or **program statements**. If there are no such declarations the list is empty. A name is an *invariable name (variable name)* if it occurs in the *invariable list (variable list)*.

No name may occur more than once in the concatenation of the *invariable list* and the *vaiable list* of a given block or program.

5.1.2.2. The **expressions** in a **subrange indicator** must both be of the same type.

5.1.2.3. The **expression** in an **array bound** must be of type integer.

5.1.3. *Examples*

> *<u>variable</u> i <u>is</u> <u>integer</u>*
> *<u>invariable</u> Page Size <u>is</u> 66*
> *<u>variable</u> Line Number <u>is</u> 0..Page Size*
> *<u>variable</u> Page <u>is</u> <u>array</u> <u>zero</u> .. Page Size <u>of</u> <u>text</u>*

5.1.4. *Semantics*

{As mentioned above (q.v. Section 1.2.2), **names** are used to label values. This usage is a little different from that of many programming languages. A graphic description of the usage of names in 3R is given in [4]. Each *type* corresponds to a data space containing all the values of that type, e.g. *<u>integer</u>* corresponds to the number line, *<u>array</u> <u>zero</u> .. 1 <u>of</u> <u>real</u>* to the cartesian plane, etc.. In 3R one speaks about "assigning a **name** to a value". This may be visualised as the act of pinning a flag bearing the **name** to the point in the value space representing the value.} A **declaration** indicates that a **name** may be assigned only to values of a specified type. {It corresponds to the manufacture of a new flag, which can be attached only to values in the appropriate value space.}

5.1.4.1. The **name** introduced by a **variable declaration** is not initialised. It must be assigned to some value before it can be used in an **expression**. {This is achieved by a **computation** in which the **name** appears to the left of a **becomes token**, which corresponds to moving the flag bearing the **name** to a place in the value space.}

5.1.4.2. The **name** introduced by an **invariable declaration** is assigned to a value obtained from the **expression** according to the rules given in Section 7. The type of the **name** is the same as that of the **expression**. {An invariable **name** may not appear to the left of a **becomes token** in a **computation**. Thus the flag bearing the **name** cannot be moved.}

5.1.4.3. Values and Types.

This section describes the *types* corresponding to the various **type indicators**. **Integer token** corresponds to the countably infinite set of negative, zero and positive integral values. **Real token** corresponds to the continuum of real numbers. **Text token** corresponds to values in the set *Char**, where *Char* is some {implementation defined} set of characters.†

A **subrange indicator** corresponds to the type of the expressions which make up the indicator. It also makes manifest an assertion on the part of the programmer that the first expression has a value less then the second expression and that the value of the variable introduced by the declaration will always lie in the closed interval defined by these expressions. {An implementation may use this information (for example to save store by packing values asserted to be small) or it may ignore it altogether. In either case, provided the assertion is correct, the meaning of the program is the same.} If the assertion is ever false, *chaos* results.

An **array indicator** specifies a base type B corresponding to the **base type indicator** and a domain size $n+1$ where n is the value of the **expression** in the **array bound**. n must not be negative. The type corresponding to the **array indicator** is the cartesian product

---

† C^* denotes the Catenation (or Kleene) Closure of the set C, i.e. $\bigcup_{n=0}^{\infty} C^n$. See [9]. Thus *Char** includes all finite sequences (of length zero or more) of characters.

of *n+1* replications of *B*, i.e. the set B^{n+1}. {Since the **base type indicator** may itself be an **array indicator** this definition is recursive.}

5.2. Commands

In addition to **simple commands**, which describe a single imperative action, there are two compound commands which enable a choice to be made between different sequences of simple commands. The **choice command** is used to express the solution of a problem by cases; the **tested invocation** is used to detect (and possibly recover from) program failure.

```
command:
        simple command;
        choice command;
        tested invocation.
```

5.3. The Choice Command

5.3.1. Context-free Syntax

```
choice command:
        guarded command chain, newline token, end choice token.

guarded command chain:
        guarded command;
        guarded command, newline token, guarded command chain.

guarded command:
        if token, guard, newline token, simple command chain.

simple command chain:
        simple command;
        simple command, newline token, simple command chain.
```

5.3.2. Example

```
if a > b
    Compute results for case where a is larger
if a < b
    Compute results for case where b is larger
if a = b
    fail Print['a = b']
otherwise chaos
```

5.3.3. *Semantics*

A **choice command** is composed of several **guarded commands**, each of which is appropriate in different circumstances. A **guarded command** can only be executed when its **guard** evaluates to *true* (q.v. Section 5.4). A **choice command** specifies execution of exactly one **guarded command** from its **guarded command chain**. If it is impossible to do this because all the **guards** are *false* then *chaos* results. {If more than one of the **guards** is *true* then it is not specifiec which guarded command is chosen.}

5.4. Guards

The **guards** in a choice command yield truth values, represented below by *true* and *false*.

5.4.1. *Context-free Syntax*

```
guard:
        conjunctive formula;
        disjunctive formula;
        negation token, boolean;
        boolean.

conjunctive formula:
        boolean, conjunction token, conjunctive formula;
        boolean, conjunction token, boolean.

disjunctive formula:
        boolean, disjunction token, disjunctive formula;
        boolean, disjunction token, boolean.

boolean:
        relational expression;
        parenthesised guard.

relational expression:
        expression, relator, expression.

parenthesised guard:
        open parenthesis token, guard, close parenthesis token.

relator:
        equals token;
        differs from token;
        is greater than token;
        is at least token;
```

is at most token;
is less than token.

5.4.2. *Context Restrictions*

In a **relational expression**, both the expressions must be of the same type (q.v. Section 7) and the **relator** must be *defined* for that type (q.v. Section 5.4.4.2).

5.4.3. *Examples*

Author < *'zzzz'*
abs tolerance < *0.000 001*
0 ≤ *theta* ∧ *theta* ≤ *90* ∧ *r* ≤ *1*
(*x* < *1* ∧ *x* > *-3*) ∨ (*y* < *7* ∧ *y* > *5*)
(*i* < *2* ∨ *j* ≥ *2*) ∧ ...
(*page length* = *line number* ∨ *page length* = *0*)

{Note that the syntax requires the parentheses in both of the last two examples.}

5.4.4. *Semantics*

5.4.4.1. The **conjunction token** and **disjunction token** represent ordinary logical conjunction and disjunction; the **negation token** represents logical negation. This meaning is given in the following table.

left operand b1 right operand b2	*true* *true*	*true* *false*	*false* *true*	*false* *false*
conjunction: b1 ∧ b2 disjunction: b1 ∨ b2 negation: ¬b2	*true* *true* *false*	*false* *true* *true*	*false* *true* *false*	*false* *false* *true*

Since the operands must be evaluated before these definitions can be applied, if either operand is undefined the value of the formula is also undefined.

5.4.4.2. A **relational expression** is evaluated by first evaluating the **expressions** (q.v. Section 7) and then evaluating the relation according to the ordinary mathematical meaning conveyed by the token which forms the **relator**. All the **relators** are *defined* for integers, but the relations denoted by the **equals token** and the **differs from token** are not defined for reals. All the **relators** are also defined for text values; the **equals token** and the **differs from token** have their obvious meanings and the remaining relations test lexicographic ordering of the text. {Thus *'a'* < *'aa'*, *'aa'* < *'ab'*, etc..} The relations denoted by the **equals token** and the **differs from token** are defined for arrays provided they are defined for the base type, but the other relations are not.

5.4.4.3. The value of a **parenthesised guard** is the value of the **guard** it contains.

5.5. The Tested Invocation

The **tested invocation** is used in conjuction with the **fail command** (q.v. Section 6.2); it enables failure to be detected and appropriate action to be taken.

5.5.1. Syntax

```
tested invocation:
        test token, invocation, success and failure clauses,
            end test token.

success and failure clauses:
        success clause, failure clause;
        failure clause, success clause.

success clause:
        success token, simple command chain.

failure clause:
        failure token, simple command chain.
```

5.5.2. *Examples*

```
test object code := compile[expression]
on success
    evaluate[object code]
on failure
    Print['Syntax errors prevent evaluation']
end of test


test n := integer from text[number]
on failure
    pass
We have now dealt with all the numbers and go on to look at the words
on success
    Sum := Sum + n
    Sum of Squares := Sum of Squares + (n↑2)
    Item count := Item count + 1
    continue summation of numbers
end of test
```

5.5.3. *Semantics*

The execution of a **test construct** commences with the execution of the **invocation** it contains (q.v. Section 6.3.4). Subsequently, either the **failure clause** or the **success clause** is executed: the choice depends on whether the invocation was terminated by a **fail command** (q.v. Section 6.2) or was successfully completed.

Execution of a **failure clause** or **success clause** consists of the execution of its **simple command chain**. {After completion of a **tested invocation**, execution continues with the statements which follow it. A **fail command** within the **success** or **failure clause** will, of course, cause the whole tested invocation to be terminated as described in Section 6.2.}

6. Simple Commands

```
simple command:
        dummy command;
        fail command;
        computation.

computation:
        invocation;
        substitution;
        assignment.
```

6.1. The Dummy Command

6.1.1. Syntax

```
dummy command:
        dummy token.
```

6.1.2. Example

pass

6.1.3. Semantics

A dummy command performs no operation. {*It is used when the syntax demands a command but no action is required.*}

6.2. The Fail Command

6.2.1. Syntax

```
fail command:
        fail token, computation;
        fail token.
```

6.2.2. Examples

fail with Message['Output too big for field']
fail

6.2.3. Semantics

The **fail command** is used for handling errors: it causes early termination of all or part of the program. If the fail command contains a **computation** this is executed *before* the termination takes place.

If the **fail command** is a **program statement**, execution of the program is terminated. If it occurs within a **block**, the invocation of that block is terminated (q.v. Sections 5.5 and 6.3).

6.3. Invocations

6.3.1. Context-free Syntax

```
invocation:
        invocation without results;
        invocation with results.

invocation without results:
        actual block name.

invocation with results:
        joined name list, becomes token, actual block name.

actual block name:
        name with arguments.
```

6.3.2. Context Restrictions

The **actual block name** in an **invocation** must *match* the **formal block name** of some **block**, which will be referred to as the *invoked block*. {Section 2.3.2 ensures that the **actual block name** *matches* at most one **formal block name**.}

6.3.2.1. If the **invocation** is part of a **block**, then the *usage list* of that block must include all the names in the *usage list* of the *invoked block*.

6.3.2.2. The *argument list* of the **actual block name** must have the same number of entries as the *parameter list* of the *invoked block*.

6.3.2.3. The *invoked block* of an **invocation without results** must have an empty *result list*.

6.3.2.4. For an **invocation with results** it is required that:

(a) There are as many names in the **joined name list** as in the *result list* of the invoked block;

(b) The **invocation** is within the scope of these names;

(c) Each such name is a *variable name*;

(d) The types of these names correspond to the types of the names in the *result list* of the invoked block.

6.3.3. Examples

Line from [margin + 10] to [margin + 10 + <u>length</u> <u>of</u> item]
*alpha, **beta** := Roots of [6] **xsq** [+5] **x** [-1]*
t := tan[psi]

6.3.4. Semantics

An **invocation** calls for the execution of a block, which has the effect of the following algorithm {but may be implemented differently. In particular, the method by which an implementation passes its parameters is not specified}.

First, the **expressions** in the *argument list* of the **actual block name** are evaluated (q.v. Section 7) to yield a list of values. The **names** in the *parameter list* of the invoked block are assigned to these values by taking the entries of the two lists in the same order. {Section 6.3.2.2 ensures that the lists have the same number of entries, which may be zero.}

Secondly, the **statements** which comprise the **block tail** of the *invoked block* are executed in order.

Subsequent action depends on whether the execution of the *invoked block* was successfully completed or was terminated by a **fail command**.

Providing the execution was successful, the final stage of the invocation is the transfer of results, and occurs only in the case of an **invocation with results**. A list of values is constructed by taking in order the values of the names in the *result list* of the *invoked block*. The names in the joined name list of the invocation with results are then assigned to the corresponding elements of this list of values, the correspondence being obtained by taking the entries in the same order. If any of the names in the *result list* do not have defined values (e.g., because they have never been assigned to a value or because of the effect of this section) then the corresponding names in the joined name list are likewise not defined. {Section 6.3.2.4 ensures that the lists have the same number of entries.}

If the execution was terminated by a fail command, the names in the joined name list are not assigned to any values. Thus, no attempt may be made to use these names in an expression. Values which have been assigned names in the *global list* before execution of the fail command retain those names. Otherwise, the invocation as a whole behaves as if it were a fail command.

{Thus the invocation ' *Execute some block* ' has an effect identical to that of the following tested invocation.

```
test Execute some Block
on success
    pass
on failure
    fail
end of test
```

}

6.4. Substitutions

6.4.1. Context-free Syntax

substitution:
    name.

6.4.2. Context Restrictions

A substitution may only occur in a block: it may not form a program statement. The name which comprises a substitution must be in the *blocklet list* of that block. {This includes the restriction that the **substitution** must occur within the *scope* of its name.}

6.4.3. Example

select a range {*q.v. Section 3.1.3*}

6.4.4. Semantics

The restriction of Section 6.4.2 means that, within the block in which the substitution occurs, there must exist exactly one **blocklet** whose **blocklet name** is identical to the **name** comprising the substitution. The effect of the substitution is to insert the **blocklet body** of that blocklet in place of the **substitution** and to execute it. {It is left to the implementation to decide whether this effect should be achieved by in-line code or routine call.}

6.5. Assignments

6.5.1. Context-free Syntax

assignment:
    name, becomes token, expression.

6.5.2. Context Restrictions

An **assignment** must occur within the *scope* of the **name** which appears to the left of the **becomes token.** The **name** must be a *variable name*, and the type of that name must correspond to the type of the **expression** (q.v. Section 7).

6.5.3. Examples

Title := 'Report on the Notation SR'
Number of labels := Number of labels + 1
Vector := <u>array</u> (0, 2, -1)

6.5.4. *Semantics*

The **expression** is evaluated and the **name** is assigned to the result. This assignment supersedes any previous assignment to another value.

7. Expressions

An **expression** is a rule for calculating a value, which will be of one of the *types* described in Section 5.1.4.3, i.e. integer, real, text or an array type.

```
expression:
        unary formula;
        binary formula;
        ternary formula;
        primary.
```

7.1. Unary Formulae

7.1.1. Context-free Syntax

```
unary formula:
        unary operator, primary;
        unary operator, unary formula.

unary operator:
        abs token;
        plus token;
        minus token;
        sine token;
        cosine token;
        arctangent token;
        degree token;
        radian token;
        log base e token;
        log base ten token;
        e to the power of token;
        length of token;
        real from integer token;
        integer from real token.
```

7.1.2. Context Restrictions

The *operand* of a **unary formula** is the object which follows the **unary operator**: it is thus either a **primary** or another **unary formula**. It is required that the operator be defined for the type of the operand.

7.1.3. *Examples*

<u>sin</u> -theta
<u>ln</u> <u>abs</u> x
<u>length</u> <u>of</u> 'Report on 3R'

7.1.4. *Semantics*

The value of a **unary formula** is obtained by finding the value of the *operand* and performing the operation denoted by the **unary operator**. The following table specifies these operations, and gives the type of operand for which each operator is *defined* and the type of the result.

operator token	operation denoted	*operand type*	*result type*
abs token	modulus (absolute value)	*real* *integer*	*real* *integer*
plus token	null operation	*real* *integer*	*real* *integer*
minus token	negation	*real* *integer*	*real* *integer*
sine token	trigonometric sine (of angle in radians)	*real*	*real*
cosine token	trigonometric cosine	*real*	*real*
arctangent token	principal value of arctangent	*real*	*real*
degree token	conversion of radians to degrees	*real*	*real*
radian token	conversion of degrees to radians	*real*	*real*
log base e token	natural logarithm	*real*	*real*
log base ten token	logarithm to base ten	*real*	*real*

e to the power of token	exponential function (e^x)	*real*	*real*
length of token	number of characters in the text	*text*	*integer*
real from integer token	type conversion	*integer*	*real*
integer from real token	rounding towards zero (applicable only to non-negative operands)	*real*	*integer*

7.2. Binary Formulae

Formulae in 3R differ from those in mathematics in several ways. There is no precedence of operators: one cannot write $a+b\times c$ in 3R but must specify either $(a+b)\times c$ or $a+(b\times c)$ as required. Neither is left to right evaluation assumed: $a/b\times c$ is not allowed, only $(a/b)\times c$ or $a/(b\times c)$. Where the operators are associative parentheses can be omitted without ambiguity. Thus 3R allows $a+b+c$, $a+b+c-d$, $a\times b\times c$ and $a\times b\times c/d$.

7.2.1. *Context-free Syntax*

```
binary formula:
        additive formula;
        additive formula, minus token, primary;
        primary, minus token, primary;
        multiplicative formula;
        multiplicative formula, over token, primary;
        primary, over token, primary;
        primary, exponentiation token, primary;
        primary, divided by token, primary;
        primary, modulo token, primary;
        primary, at token, primary;
        primary, char of token, primary.

additive formula:
        additive formula, plus token, primary;
        primary, plus token, primary.
```

multiplicative formula:
    multiplicative formula, times token, primary;
    primary, times token, primary.

7.2.2. *Context Restrictions*

A binary operator has two operands, and in 3R all binary operators are written using infix notation, i.e. the symbol denoting the operator appears between its operands. The binary operators are listed in Section 7.2.4: it is required that an operator be *defined* for its operands.

7.2.3. *Examples*

Serial of Master + 1
First Name + Surname
(Contributory factor × Days in Month) - Basic Rate
Year mod 4
(x+2) + (2×a×b) + (b+2)
Singular + 's'

7.2.4. *Semantics*

The value of a **binary formula** is obtained by finding the values of the operands of the binary operator and performing on them the operation it denotes. These operands will either be **primaries**, whose values are obtained as described in Section 7.4.4, or **multiplicative** or **additive formulae**, whose values are obtained by a recursive application of these rules. The binary operators with their meanings and the types for which they are *defined* are as follows.

Numerical Operators

plus token	addition	defined between integers giving an integer result, and between any other combination of integers and reals giving a real result.
minus token	subtraction	
times token	multiplication	
over token	division	defined between integers and reals in any combination giving a real result.

exponentiation token	exponentiation	the right operand must be an integer and must not be negative. The result is of the same type as the left operand, which may be real or integer.
divided by token	integer division	defined between integers, which must not be negative.
modulo token	positive remainder after division	defined between integers, which must not be negative.

Text Operators

plus token	concatenation, defined between two text values.
times token	replication, defined between an integer and a text value in both combinations.
char of token	selection. The left operand is a positive integer not exceeding the length of the right operand, which is a text value. The result is a text value of length 1, being the appropriate character.

Array Operator

at token	selection of an array element, defined between an <u>*array*</u> <u>*zero*</u> .. *n* <u>*of*</u> *atype* value and an integer *i*, yields the value of the *i*th element of the array (counting from zero). The result is of type *atype*. *i* must not be negative or greater than *n*.

7.3. Ternary Formulae

A ternary operator takes three operands. There is only one such operator in 3R; it is represented by two tokens which separate the three operands.

7.3.1. *Context-free Syntax*

```
ternary formula:
        ternary formula, with token, primary, at token, primary;
        primary, with token, primary, at token, primary.
```

7.3.2. *Context Restrictions*

The left operand (which is either a **ternary formula** or a **primary**) must yield some *__array__ __zero__ .. n __of__ atype* value. The inner operand (the **primary** between the **with token** and the **at token**) must yield an *atype* value.. The right operand must yield an *integer*.

7.3.3. *Examples*

buffer __with__ 'Hello' __at__ 0
vector __with__ 1 __at__ 2 __with__ 2 __at__ 1 __with__ 3 __at__ 0
transcendental table __with__ 3.14159265 __at__ 3

7.3.4. *Semantics*

The ternary formula constructs a new array value from an old one. First the operands are evaluated: let their values be A, i and x respectively. The value of the formula is the same as that of A except that the i^{th} component of the tuple (counting from zero) has value x.

{Note that the second example is unambiguous: *2 __with__ 2 __at__ 1* is meaningless, so it is clear without inspecting the grammar that association is to the left.}

7.4. Primaries

Primaries are the basic data objects from which expressions are constructed.

7.4.1. *Context-free Syntax*

```
primary:
        denotation;
        name;
        array expression;
        parenthesised expression.
```

parenthesised expression:
    open parenthesis token, expression, close parenthesis token.

denotation:
    text denotation;
    integer denotation;
    real denotation.

array expression:
    array token, open parenthesis token, joined expression list,
        close parenthesis token.

joined expression list:
    expression;
    expression, comma token, joined expression list.

7.4.2. Context Restrictions

7.4.2.1. A **name** forming a **primary** must occur within its *scope*.

7.4.2.2. In the **joined expression list** comprising an **array expression**, all the constituent **expressions** must yield values of the same type. The type of the **array expression** corresponds to the cartesian product of as many replications of the set corresponding to this type as there are **expressions** in the **joined expression list**.

7.4.3. Examples

57
'Mary'
array (3.14159, 2.71828, 1.4142)
array ('doubtless', 'no doubt', 'undoubtedly')
((Stock number + increment) × percentage / 100)

7.4.4. Semantics

7.4.4.1. The value of a **denotation** is apparent from its representation (q.v. Section 9.2).

7.4.4.2. The value of a **name** is the value to which that **name** is assigned. In the case of an *invariable name* the name will have been assigned to a value when it was declared, and this assignment cannot change. In the case of a *variable name* the name may have been assigned to many different values, but we are only interested in the current (i.e. most recent) assignment. {The type of the value is obtained as described in Section 5.1.4.} If the **name** is not assigned to a value, the result of attempting to evaluate it is undefined.

7.4.4.3. The value of an **array expression** is the tuple formed by taking in order the values of the constituent **expressions**.

7.4.4.4. The value of a **parenthesised expression** is the value of the **expression** it contains.

8. Axiomatic Semantics

This section defines the semantics of 3R more formally by means of a function wp, closely related to that of Dijkstra [2]. wp maps each construct in the language into a predicate transformer which describes the effect of that construct.

The principle of defining semantics by means of predicate transformers, and then using these transformers to aid the program design process, is expounded in many places (including the above reference). This report will not repeat the exposition but will limit itself to a brief summary of the properties of wp and its definition for 3R.

8.1. Notation

Q, R and S will be used to represent predicates. All the objects in the syntactic class **guard** (q.v. Section 5.4) are predicates, but we also include other connectives of the predicate calculus with their usual meaning (particularly $\Rightarrow$ for material implication), the constants <u>*true*</u> and <u>*false*</u>, and the symbol <u>*failed*</u> which is used to define the semantics of the **fail command**.

$[e \rightarrow x]R$ (read: e for x in R) denotes a predicate obtained by substituting e for all free occurrences of x in R. Thus $[7 \rightarrow y](x > y) \equiv (x > 7)$. x occurs free in R if it occurs in R after all substitutions have been made. In $[7 \rightarrow y](x > y)$, x is free but y is not. Similarly $[e,f \rightarrow x,y]R$ denotes simultaneous substitution of e and f for x and y.

Eventually, for the predicate transformers to be of any use, the predicates must be given some meaning in terms of the objects the programmer manipulates, i.e. the values of program variables. We will not introduce an explicit evaluation function, arguing instead that much of the power of the method comes from the ease with which it is possible to alternate between regarding $x > 7$ as a purely syntactic predicate and as an assertion that the value corresponding to the name x is greater than 7.

The symbol '$\triangleq$' is used to mean 'is defined to be equivalent to'. A will be used to represent part of a program, defined in terms of the grammar and the context restrictions on it.

wp A is a function which maps predicates to predicates, the predicate transformer for A. A itself might be, for example, **simple command** or **choice command**. We argue that wp A captures the semantics of A, so of course the function represented by wp **simple command** depends on the composition of **simple command**. Thus wp for 3R is defined by first giving wp **program** and so on for all the classes in the syntax. wp itself can be considered as a family of functions; each member describes the semantics of a particular construct in 3R and is obtained by applying wp to that construct.

Suppose a program is required to achieve R, some condition on its parameters and results. Additionally, suppose that S is specified as being true before execution commences. A **program** such that

wp **program** R = S

will achieve the desired result. wp A R can be interpreted as the weakest precondition under which program A is guaranteed to terminate with R satisfied.

It is obvious that the syntax of 3R given in Sections 1 to 7 of this report contains many redundant productions: extra productions have been deliberately introduced so that each construct referred to in the text has a name. For example, consider

```
command:
        simple command;
        choice command;
        tested invocation.

simple command:
        dummy command;
        fail command;
        computation.
```

In the definition of wp which follows, there are references to wp **command** but no direct definition. The reader is expected to examine the **command** in question and to decide if it is a simple command, a choice command or a tested invocation; if it is the first he must decide whether it is a dummy command, fail command or computation. The definition of wp for these constructions will then be found in Section 8.6.

In order to make this pattern matching easier, the grammar has been reproduced production-by-production where required. Also, each sub-section corresponds numerically to the section of the report which deals with the same construction, e.g. Section 8.6.1 and Section 6.1 both deal with the dummy command.

8.2. Programs

program:
    argument declarations option, program body, newline token, finish token.

wp (argument declarations option, program body, newline token, finish token) R

$\triangleq$ $[\underline{false} \rightarrow \underline{failed}]$wp (argument declarations option) (wp (program body) R)

program body:
    program element, newline token, program body;
    program element.

wp (program element, newline token, program body) R

$\triangleq$ wp (program element) (wp(program body)$R \wedge \neg\underline{failed}$)

$\vee$ wp (program element) $(R \wedge \underline{failed})$

program element:
    block;
    program statement.

program statement:
    statement.

8.3. Blocks and Blocklets

block:
    block head, block body, block end.

wp (block head, block body, block end) R $\triangleq$ R

{The declaration of a block does not affect the state.}

8.4. Arguments

{The significance of Arguments is explained in Section 8.6.3 - Invocations.}

8.5. Statements

```
statement:
        declaration;
        command.
```

8.5.1. Declarations

```
declaration:
        variable declaration;
        invariable declaration.

variable declaration:
        variable token, name, is token, type indicator.
```

wp (variable token, name, is token, type indicator) $R \;\triangleq\; R$

provided name does not occur free in R. {Thus no assumptions can be made about uninitialised variables.}

```
invariable declaration:
        invariable token, name, is token, expression.
```

wp (invariable token, name, is token, expression) R

$\triangleq$ [expression $\rightarrow$ name] R

8.5.2. Commands

```
command:
        simple command;
        choice command;
        tested invocation.
```

8.5.3. The Choice Command

```
choice command:
        guarded command chain, newline token, end choice token.

guarded command chain:
        guarded command;
        guarded command, newline token, guarded command chain.
```

guarded command:
    if token, guard, newline token, simple command chain.

Informally, the structure of this command is

```
if guard 1
    simple command chain 1
if guard 2
    simple command chain 2
  .
  .
  .
  .
if guard n
    simple command chain n
otherwise chaos
```

For all the constructs encountered so far it has been possible to define wp by a recursive rule mirroring the recursive syntax. This is not so for the choice command†. Instead we have

wp choice command R

$$\begin{aligned} \triangleq\ & (\text{guard } 1 \vee \text{guard } 2 \vee \dots \vee \text{guard } n) \\ & \wedge\ \text{guard } 1 \Rightarrow \text{wp (simple command chain 1) } R \\ & \wedge\ \text{guard } 2 \Rightarrow \text{wp (simple command chain 2) } R \\ & \wedge\ \dots \\ & \wedge\ \text{guard } n \Rightarrow \text{wp (simple command chain n) } R \\ \equiv\ & \bigvee_{i=1}^{n} (\text{guard } i) \wedge \bigwedge_{i=1}^{n} \text{guard } i \Rightarrow \text{wp (simple command chain i) } R \end{aligned}$$

Although n is arbitrarily large, it is finite and there is no complication in introducing the quantified connectives: $\bigvee_{i=1}^{n}$ is simply a shorthand for something which, if written out in full, would occupy an arbitrarily large (but finite) piece of paper.

It will be seen that if all the guards in a choice command are *false*, wp choice command R = *false*, i.e. there is no pre-condition

---

† At least, it cannot be done without introducing a lot more notation, which is less desirable than the use of the ellipsis. This is because the recursive rule we wish to unravel defines a guarded command chain as a succession of guarded commands and guarded command chains, and these syntactic entities *have no semantics*. It is not possible to split off one of the guarded commands and describe the semantics of a guarded command chain in terms of the two parts thus formed: in doing so we irrevocably lose the non-determinism of the choice command.

which enables the desired post-condition to be reached. This is the meaning of *chaos*: it is not possible to prove anything about a program containing such a choice command.

simple command chain:
    simple command;
    simple command, newline token, simple command chain.

wp (simple command, newline token, simple command chain) R
  $\triangleq$ wp (simple command) (wp(simple command chain)$R \wedge \neg \underline{failed}$)
    $\vee$ wp (simple command)($R \wedge \underline{failed}$)

8.5.4. Guards

Guards are evaluated to yield truth values as described in Section 5.4.

8.5.5. Tested Invocations

tested invocation:
    test token, invocation, success and failure clauses,
      end test token.

success and failure clauses:
    success clause, failure clause;
    failure clause, success clause.

wp (test token, invocation, success and failure clauses,
                end test token) R
  $\triangleq$ wp invocation ((wp success clause R) $\wedge$ $\neg\underline{failed}$)
    $\vee$ wp invocation ((wp failure clause R) $\wedge$ $\underline{failed}$)

success clause:
    success token, simple command chain.

wp (success token, simple command chain) R

  $\triangleq$ [$\underline{false} \rightarrow \underline{failed}$] (wp simple command chain R)

failure clause:
    failure token, simple command chain.

wp (failure token, simple command chain) R

  $\triangleq$ [$\underline{false} \rightarrow \underline{failed}$] (wp simple command chain R)

8.6. Simple Commands

simple command:
  dummy command;
  fail command;
  computation.

computation:
  invocation;
  substitution;
  assignment.

8.6.1. Dummy Command

wp dummy command $R \triangleq R$

8.6.2. Fail Command

fail command:
  fail token, computation;
  fail token.

wp (fail token, computation) R

$\triangleq$ wp computation (wp fail token R)

wp fail token $R \triangleq [\underline{true} \rightarrow \underline{failed}]\, R$

8.6.3. Invocations

invocation:
  invocation without results;
  invocation with results.

actual block name:
  name with arguments.

8.6.3.1. Invocations without results.

invocation without results:
  actual block name.

wp actual block name R

$\triangleq$ [*argument list* → *parameter list*]wp block tail R

Argument list is that extracted from the name with arguments which forms the invocation. Block tail, *parameter list* and *result list* are those of the *invoked block* (q.v. Section 6.4.2).

8.6.3.2. Invocations with results.

invocation with results:
    joined name list, becomes token, actual block name.

wp (joined name list, becomes token, actual block name) R
  $\triangleq$ [*argument list* → *parameter list*]wp block tail
    (([*result list* → joined name list]R ∧ ¬*failed*) ∨ (R ∧ *failed*))

provided that the value of (R ∧ *failed*) is independent of the values of the names in the joined name list. {This forbids any assumptions about the result of a failed invocation.}

block tail:
    statement;
    statement, newline token, block tail.

wp (statement, newline token, block tail) R
  $\triangleq$ wp statement (wp block tail R ∧ ¬*failed*)
    ∨ wp statement (R ∧ *failed*)

8.6.4. Substitution

substitution:
    name.

wp name R $\triangleq$ wp command chain R

where command chain directly derives from the blocklet body whose blocklet name is name (q.v. Sections 3.2.1 and 6.4.4).

8.6.5. Assignment

assignment:
    name, becomes token, expression.

wp (name, becomes token, expression) R $\triangleq$ [expression → name]R

8.7. Properties of wp

The wp function defined above has the following properties, which may be proved from the definition.

8.7.1. Strictness

For all constructs A

wp A *false* = *false*

8.7.2. Distribution over $\wedge$

For all constructs A and predicates Q and R

(wp A Q) $\wedge$ (wp A R) = wp A ($Q \wedge R$)

8.7.3. Continuity

Given an infinite sequence of predicates Q_i, $i \geq 0$, such that $Q_i \Rightarrow Q_{i+1}$

$$\text{wp A } \left(\bigvee_{i\geq 0} Q_i\right) = \bigvee_{i \geq 0} (\text{wp A } Q_i).$$

From property 8.7.2 it is easy to prove the following property of Monotonicity: if $Q \Rightarrow R$ then wp A $Q \Rightarrow$ wp A R. Note that in general wp does not distribute over $\vee$, i.e.

$$(\text{wp A } Q) \vee (\text{wp A } R) \neq \text{wp A } (Q \vee R).$$

(In particular, consider wp **choice command.**) However, the weaker condition

$$(\text{wp A } Q) \vee (\text{wp A } R) \Rightarrow \text{wp A } (Q \vee R)$$

follows trivially from monotonicity.

9. Terminal Symbols

This section contains the syntax and semantics of the *lexemes* and outlines a comment convention.

A Sk program consists of a sequence of *symbols*. The *tokens* are *symbols*, as are the *letters* and *digits* and any other characters we wish to include because they are available on our typewriter or line printer. In most representations the *tokens* will be composed of multiple characters, e.g. the suggested representation for the **where token** is *where*, and the **newline token** might be represented as the pair of characters carriage return and line feed. No difficulty should arise so long as the designers of representations ensure that it is easy to map multi-character sequences into the appropriate tokens.

9.1. Representation of Tokens

Where two tokens are juxtaposed they should be separated by at least one space; additional spaces before or after a token are optional and may be used to improve readability.

The list of recommended representations given in Section 1.2.1 uses underlining to create new symbols. Underlined words have the advantage of standing out from the page. Possible alternatives, if underlining is unavailable, are the use of bold face or capital letters. Stropping with quotes or points is *not* recommended. Stropping has the effect of reducing readability rather than enhancing it. If capital letters are used to create new symbols, the term *letter* should be understood to exclude them. {It must always be clear whether a given sequence of characters is a *token* or a **name**.} The term *digit* means any of the ordinary decimal digits.

9.2. Syntax and Semantics of the Lexemes

The following symbols, in addition to *letters* and *digits*, are used in the construction of the lexemes.

symbol	representation
point symbol	.
times ten to the power symbol	$\underline{e}_{10}$
minus symbol	-
plus symbol	+
quote symbol	'
escape symbol	*
open bracket symbol	[
close bracket symbol	]

9.2.1. Text denotations

text denotation:
    quote symbol, item sequence, quote symbol.

item sequence:
    item, item sequence;
    empty.

item:
    any character other than that representing newline token
        escape symbol or quote symbol;
    newline representation;
    quote representation;
    escape representation.

The last production will not be made more formal. To avoid a multiplicity of conventions, newline representation will be **n*, quote representation will be **'* and escape representation will be **.

The value of a text denotation is the sequence of characters obtained by replacing the newline, quote and escape representations by the appropriate characters.

Using the representations given above, the following are text denotations.

'This is Text'
'Everything's been said'*
''All words are pegs to hang ideas on.*'*n(H.W.Beecher)'*

Note that it is not possible for a **text denotation** to extend over more than one line.

9.2.2. Integer denotations

Integers are denoted by a sequence of digits in the normal scale of 10. There are no denotations for negative integers.

9.2.3. Real denotations

The denotation must contain either a **point symbol** or a **times ten to the power symbol**, or both; a **point symbol** must always be followed by a digit. The integer exponent following the **times ten to the power symbol** may be preceded by a **plus symbol** or a **minus symbol** if required. Thus the following are equivalent **real denotations**.

5.7
57₁₀-1
.57₁₀+1
0.57₁₀1

9.2.4. Names

A **name** consists of a *letter* followed by a (possibly empty) sequence of *letters*, *digits*, spaces and any other symbols a representation can allow without introducing ambiguity. {Thus the **minus token** could not be used, but '&' might be allowable, depending on the representation of the tokens.}

Two names are *identical* if they differ only in that one name contains multiple spaces where the other contains a single space. Upper and lower case letters are different. Examples:

theta
Start Column
Line printer

9.2.5. *Names with arguments*

A **name with arguments** consists of a **name** together with zero or more **arguments**, where

argument:
    open bracket symbol, expression, close bracket symbol.

The **arguments** may precede, intersperse or succeed the characters of the **name**. Examples:

Line from [*Start Column*] *to* [*End Column*]
[*a*] *minus* [*b*]
tan [*theta*]
This one happens to have no arguments
Ackermann[*3*][*2*]

The *argument list* of a **name with arguments** is the list of the **expressions** taken in order.

9.3. Comments and Continuations

The syntax does not explicitly permit comments. This is not meant to discourage their use, but reflects the view that the commentary on a program is not itself part of that program.

Since the **newline token** is both part of the syntax and the only means whereby a newline may be started, a means of breaking inconveniently long lines is provided.

The following conventions are recommended for comment and layout; they do not apply inside denotations. The symbols used are:

symbol	representation
start comment symbol	{
end comment symbol	}
continuation symbol	...

(i) The **start comment symbol**, the matching **end comment symbol** and all the symbols between them are equivalent to a space. By "matching" we intend to allow nested comments.

(ii) Multiple spaces are equivalent to a single space.

(iii) Where a **newline token** is immediately followed by a symbol which is not a space, that **newline token** and all succeeding symbols up to and including the next **newline token** are equivalent to a **newline token**.

(iv) Multiple **newline tokens** are equivalent to a single **newline token**.

(v) A **newline token** preceded by the **continuation symbol** is equivalent to a single space.

{Note that it is possible for more than one character to represent "space", e.g. in a particular representation "horizontal tab" may be considered as a space.}

10. Acknowledgements

As was mentioned in the Foreword, the original design of 3R was a by-product of a contract let by the Design Office Consortium, who have consented to the publication of a modified version of the notation. The changes have been initiated by both the search for simple formal semantics and the experience of using the language; reassuringly often these two avenues led to the same destination. I am grateful to Brian Shearing for giving freely of his time so that this report could benefit from his experience of 3R.

Throughout the development process Professor C. A. R. Hoare has provided helpful suggestions and constructive criticism, and all the members of the Programming Research Group have played their part by generating an atmosphere in which all the drawbacks of an idea are rapidly exposed. J. Mack Adams, Malcolm Harper, Andrew Newman, and Joe Stoy have all provided special help, from proof-reading and assisting with the text-processing system to discussing issues of formal semantics.

11. References

[1] Alcock, D. G. *Readability of Design Programs.* Proceedings of Colloquium on Interface Between Computing and Design in Structural Engineering, ppIII.1 - III.10, Bergamo; September 1978

[2] Dijkstra, E. W. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs.* Comm ACM Vol 8 Nr 8, pp453 - 457. *Also* Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, 1976

[3] Gries, D. *Compiler Construction for Digital Computers.* Chapter 2, pp12 - 48; Wiley, 1971

[4] Hehner, E. C. R. *On Removing the Machine from the Language.* Acta Inf. Vol 10 Fasc 3 pp 229 - 243 (1978)

[5] Hehner, E. C. R. *do Considered od: A contribution to the Programming Calculus.* Acta Inf. Vol 11 Fasc 4, pp287 - 304 (1979)

[6] Hopcroft, J. E. and Ullman, J. D. *Formal Languages and their Relation to Automata.* Addison-Wesley, 1969

[7] National Standards Institute, American. *American National Standard Programming Language PL/I. X3.53, 1976*

[8] P.S.A. *3R - A notation for Describing Computer Programs.* Directorate of Architectural Services, Property Services Agency, Department of Environment; April 1978

[9] Salomaa, A. *Formal Languages.* Academic Press, 1975

[10] Shearing, B. H. *The Forpa Programmer's Manual.* Design Office Consortium, Guildhall Place, Cambridge, 1977

[11] Shearing, B. H. *Nustress Programmer's Manual (Part 1).* S.I.A. Ltd., 23 Lower Belgrave Street, London, SW1W ONW; October 1978

[12] Wirth, N. *Systematic Programming: An Introduction.* Prentice-Hall, 1973

Index to Grammar and Technical Terms

This index lists all the technical terms defined in the report (printed in *italics*) and all the non-terminal symbols of the grammar (printed in gothic). Each listing gives the section and page numbers on which the appropriate definition may be found. Uses of the terms are not listed.