S-algol Reference Manual

Ron Morrison

University of St. Andrews,
North Haugh,
Fife,
Scotland.
KY16 9SS

CS/79/1

Contents

Chapter

Appendices

1. Preface

Programming language design is probably the most emotive subject in Computational Science today. Nearly everyone uses a programming language and most people have something to say about their design.

S-algol is presented as a member of a family of languages in the algol tradition that are constrained by a design methodology. This design methodology is based on the belief that most programming languages are too big and intellectually unmanageable. In addition it is believed that these problems arise in part from the languages being too restrictive. The number of rules to define a language increases when a general rule has additional rules attached to constrain its use in certain cases. Ironically these additional rules usually make the language less powerful. Power through simplicity, simplicity through generality is the guiding tenet.

The design methodology is based on three semantic principles which can be attributed to Strachey [9] and Landin [5] and later developed by Tennent [10] and Morrison [7]. These are

(a) The principle of data type completeness

(b) The principle of abstraction

(c) The principle of correspondence

The principle of data type completeness states that the rules for using data types must be complete with no gaps. This does not mean that all operators in the language need be defined on all data types but rather that general rules have no exceptions. Examples of lack of completeness can be seen in Pascal [12], for example, where only some data types are allowed as members of sets. The Principle of Data Type Completeness is bound to lead to simpler languages since it avoids the complexity of special cases.

Abstraction is a process of extracting the general structure to allow the inessential details to be ignored. It is a facility well known to mathematicians and programmers since it is usually the only tool they have to handle complexity. The principle of abstraction when applied to language design is invoked by identifying the semantically meaningful syntactic categories in the language and allowing abstractions over them. The most familiar form of abstraction is the function which is an abstraction over expressions.

Finally, the principle of correspondence states that the rules for introducing and using names should be the same everywhere in a program. In particular there should be a one to one correspondence between introducing names in declarations and introducing names as parameters.

Armed with the above rules the language may be designed as follows

Data Types

Decide which data types, both simple and compound, are required by the language and define the operations on these data types. The flavour of the language will be determined by the data objects it can manipulate. The principle of data type completeness is invoked to ensure that all data objects have the same civil rights. Ignoring the principle means introducing rules to handle exceptions thus making the language more complex. For example, if arrays are allowed then arrays of arrays should be allowed as should expressions and functions with array results.

The Store

Introduce the store, if any, and the manner in which it may be used. First of all the relationship between the store and the data types should be defined. This includes the implementation of pointers, data locations and protection on these locations. For example, constants may be regarded as storage locations which may not be updated [4].

The introduction of the store forces consideration of the language control structures. The designer can take his pick and is well advised to read the excellent paper by Ledgard and Marcotty [6] on the subject.

Abstraction

Tennent [10] has suggested that the method of applying the principle of abstraction is to identify the semantically meaningful syntactic categories and invent abstractions for each. This he does for Pascal and proposes some extensions to complete the abstractions. However, he points out that it is not always an easy matter to identify these categories in the first place. Examples of abstractions are functions which are abstractions over expressions and procedures which are abstractions over statements. An important point about abstractions is that they may be parameterised.

Declaration and Parameters

Invent the declarations and the parametric objects together. There must be a one to one correspondence between the two. This does not mean that they necessarily have the same syntax but that for every type of declaration there is an equivalent parametric type. Parameter passing modes are also included in this correspondence. For example, the declarative equivalent of call by value is an initialising declaration. If functions, record classes etc can be declared then they can be passed as parameters even though they are not defined as data objects in the language. Finally, if the language has a facility to define new data types and give them a name then the type can be passed as a parameter.

Input and Output

The I/O models for most high level languages tend to reflect the environment in which they were designed. Some attempts have been made to design and implement comprehensive I/O systems. Unfortunately where it has not been tied to particular hardware it has never been very successful. Nowhere else in the design of a programming language does the hardware intervene as much as it does in the I/O system. When a new I/O device becomes available the language must be able to make use of it. Of course, this situation is hopeless and perhaps the wisest approach to I/O is to allow the implementor to deal with it for a particular environment, as the Algol 60 designers proposed.

Iterate

Re-evaluate the language and correct or justify any idiosyncrasies in the design. Hopefully the design process will converge.

Concrete Syntax

The final stage of language design is to propose a concrete syntax. Ideally different groups of workers could have a different syntax. However, there are many users who do not wish to design their own syntax and so the language must provide at least one possibility.

It seems very obvious to say that the syntax should be simple and easy to learn. That may be so but there is no doubt that some of the success of the language depends on the cosmetics. Also, a carefully chosen syntax can ease the problem of compilation.

How often the rules can be broken is for the designer's own conscience. However, every time the rules are broken the language becomes more complex. The rules were introduced to help design simpler and more intellectually manageable languages and should only be ignored with great care.

S-algol is the first of the family of algols to be produced under this design methodology. It reflects my own personal preference for data objects and syntax. It also ignores the principle of abstraction with regard to sequencers and declarations. However there are no exceptions to the principle of correspondence or the principle of data type completeness in the language. I see this as the strength of the language.

A number of people must be thanked for their help with the S-algol project. Firstly, Jack Cole whose enthusiasm and encouragement were responsible for the development of the system. Peter Bailey must be thanked for his help in testing the initial system on the PDP11 and in proof reading the original manual, as well as his enormous willingness to help when difficulties arose. Pete also wrote the VAX interpreter with Paul Maritz and the S-algol code generators. Paul Maritz implemented S-algol on the Zilog Z80 and pointed out a discrepancy in the application of the principle of correspondence to the language. Paul must also be blamed for the old binder for separately compiled modules. Fred Brown implemented a fast version of the Outline graphics for the Z80. Mike Livesey must be thanked for our discussions of the semantic principles on which the language is based and David Turner, now in Canterbury, with whom I started on language design deserves a special mention for his contribution [11]. Tony Davie, Michael Weatherill, Hamish Gunn and Ian Sommerville also may useful comments during the early design and implementation of S-algol. Tony also made numerous comments for improving this reference manual.

In 1983, S-algol was used as a basis for PS-algol as part of the research in the PISA project. Al Dearle wrote versions of the interpreter in C and altered the compiler to generate new object code. It is on these versions that Dave Munro tailored the system to fit into the Macintosh environment. For this he rewrote the compiler in C, adapted the interpreter and surrounded it all with a Mac environment. To the language he added the raster graphics facilities of PS-algol [8] built by Al & Fred [2]. In parallel with this Dave made an equivalent system available on the SUN workstations. Al, Ray Carrick and Tony Davie have now written an introductory text based on S-algol. I hope you enjoy reading it and using the system.

Ron Morrison
18-9-88

2. Syntax Specification

It is important that a programming language can be formally defined since it gives implementors a standard to work on. There are two levels of definition, syntactic and semantic. A formal semantic description, a denotational semantics, of S-algol is given by Adamson [1]. Here we will deal with the formal syntactic rules giving an informal semantic description of the syntactic categories. That is, we will define the set of all syntactically legal S-algol programs remembering that the meaning of any one of these programs is defined by the semantics.

To define the syntax of a language we need another notation which we will call a **meta language** and in this case we choose a variation of Backus-Naur form.

The syntax of S-algol is specified by a set of rules or **productions** as they are normally called. Each production specifies the manner in which a particular syntactic category (e.g. a clause) can be formed. The syntactic category name is enclosed in the meta symbols '<' and '>' thus distinguishing it from names or reserved words in the language. These syntactic categories can be mixed in productions with terminal symbols which are actual symbols of the language itself. Thus, by following the productions until we have only terminal symbols, we can derive legal programs. We can also use the productions in a compiler to check that a program is legal and a very efficient method of doing this, called Recursive Descent is described by Davie & Morrison [3].

Other meta symbols include '|' which allows a choice in a production. The square brackets '[' and ']' are also used in pairs to denote an object is optional. When used with a '*' we have a zero or many times repetition. You should not confuse the meta symbols '|','<', '>', '*', and '[', ']' with the actual symbols in S-algol and we will be careful to keep the two concepts completely separate in our description. In the syntactic description of S-algol these symbols are defined by the following non-terminal symbols:

<bar> ::= |

<lab> ::= <

<rab> ::= >

<star> ::= *

<lsb> ::= [

<rsb> ::=]

As you may expect with any reasonable programming language, the productions for S-algol are recursive which means that there are an infinite number of legal S-algol programs. However the syntax of S-algol can be described in about 60 productions. We will break ourselves in gently with an example.

<integer_literal> ::= [<add_op>]<int_literal>

<int_literal> ::= <digit>[<digit>]*

<add_op> ::= + | -

<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0

The above syntax indicates that an integer can be formed as an optional + or - followed by one or many digits. The full context-free syntax of S-algol is given in Appendix I.

3. Types and Type Rules

The S-algol type system is based on the notion of types as sets of objects from the value space. These sets may be built in, like integer, or they may be formed by using one of the built in type constructors, like **structure**. The constructors obey the Principle of Data Type Completeness. That is, where a type may be used in a constructor, any type is legal without exception. This has two benefits. Firstly it allows a very rich type system to be described using a small number of defining rules since all the rules are very general without exception to them. This reduces the complexity of the defining rules. The second benefit is that the type constructors are as powerful as they can be since there is no way to restrict their domain. This increases the power of the language.

3.1 Universe of Discourse

There is an infinite number of data types in S-algol defined recursively by the following rules.

1. The scalar data types are integer, real, boolean, string, picture, pixel and file.

2. #pixel is the type of an image made up of pixels arranged as a rectangular matrix.

3. For any data type T, *T is the data type of a vector with elements of type T.

4. The data type pntr comprises a structure with any number of fields, and any data type in each field.

In addition to the above data types there is a number of other objects in S-algol to which it is convenient to give a type in order that the compiler may check their use for consistency.

5. The type of a procedure with parameters T_1,...,T_n and result type T_m is (T_1,...,T_n -> T_m).

6. Clauses which yield no value are of type void.

7. The class of a structure with fields of type T_1,...,T_n is of type (T_1,...,T_n)-structure and its fields are of type T_i-field.

The universe of discourse of the language is defined by the closure of rules 1 and 2 under the recursive application of rules 3 and 4.

3.2 Type Rules

The type rules form a second set of rules to be used in conjunction with the context free syntax to define well formed programs. The generic types that are required for the formal definition of S-algol can be described by the following:

type arith **is** int | real

type ordered **is** arith | string

type writeable **is** ordered | bool

type literal **is** writeable | pixel | pntr | file

type image **is** #pixel | #cpixel

type nonvoid **is** literal | pic | image | vector

type vector **is** *nonvoid | *cnonvoid

type type **is** nonvoid | void

For example, in the above, the generic arith can be either an int or a real, representing the types integer and real in the language. In the type rules, the language types and generic types are written in shadow font to distinguish them from the reserved words.

To check that a syntactic category is correctly typed the context free syntax is used in conjunction with a type rule. For example, the type rules for the two-armed **if** clause is

t : type, **if** <clause> : bool **then** <clause> : t **else** <clause> : t => t

This rule may be interpreted as follows. t is given as a type from the table above. That is, it can be any type including void. Following the comma, the type rule states that the reserved word **if** may be followed by a clause which must be of type boolean. This is indicated by : bool. The **then** and **else** alternatives must have clause of type t, whichever type it is. That is, both alternatives must have the same type. The resultant type, indicated by =>, of this production is also t, the same as the alternatives.

The type rules will be used throughout this manual, in conjunction with the context-free syntax rules to describe the language. A complete set of type rules for S-algol is given in Appendix II.

4. Literals

Literals are one of the basic building blocks of a program and allow values to be introduced. A literal is defined by

<exp6> ::= <literal>

<literal> ::= <integer_literal> | <real_literal> | <bool_literal> |
<string_literal> | <pixel_literal> | <pntr_literal> | <file_literal>

4.1 Integer Literals

These values of type int are defined by

<integer_literal> ::= [<add_op>]<int_literal>

<int_literal> ::= <digit>[<digit>]*

<add_op> ::= + | -

<integer_literal> => **int**

An integer literal is one or more digits possibly preceeded by a sign.

e.g. 1 0 -1256 8797

4.2 Real Literals

These are of type real and are defined by

<real_literal> ::= <integer_literal>.[<int_literal>][e< integer_literal >]

<real_literal> => **real**

Thus, there are a number of ways of writing a real literal. For example

1.2 3.1e2 5.e5
1. 3.4e-2 3.4e+4

3.1e-2 means 3.1 times 10 to the power -2 (i.e. 0.031)

4.3 Boolean Literals

There are only two literals of type bool. They are the symbols **true** and **false**. They may be used with obvious meaning when a boolean literal is required.

<boolean_literal> ::= **true | false**

<boolean_literal> => **bool**

4.4 String Literals

A string literal is a sequence of characters in the character set (ASCII) enclosed by double quotes. The syntax is

<string_literal>::= <double_quote>[<char>]*<double_quote>
<char> ::= any ascii character except ' or " | <special_character>
<special_character> ::= <single_quote><special_follow>
<special_follow> ::= n | p | o | t | b | <single_quote> | <double_quote>

<string_literal>=> **string**

The empty string is denoted by "". Examples of string literals are

"This is a string literal"
"I am a string"

The programmer may wish to have a double quote itself inside a string literal. This requires using a single quote as an escape character and so if a single or double quote is required inside a string literal it must be preceded by a single quote. For example

"a'"" has a value a" and
"a'" has a value a'

There are a number of other special characters which may be used inside string literals. They are

'n newline
'p newpage
'o carriage return
't horizontal tab
'b backspace

These characters are normally used in strings to represent special ASCII characters.

4.5 Pixel Literals

A pixel literal specifies the values in the planes of the pixel. It is defined by

<pixel_literal> ::= **on**[&<pixel_literal>] | **off**[&<pixel_literal>]

<pixel_literal> => **pixel**

For example

on **off** & **on** & **off**

4.6 File Literal

There is only one file literal. It is used to denote an inactive file.

<file_literal> ::= **nullfile**

nullfile => **file**

4.7 pntr Literal

There is only one **pntr** literal. It is used to denote an empty structure.

<pntr_literal> ::= **nil**

```
nil                    =>    pntr
```

5. Primitive Expressions and Operators

The order of execution of an S-algol program is strictly from left to right and top to bottom. This rule becomes important in understanding side-effects in the store.

5.1 Boolean Expressions

Objects of type bool in S-algol can have the value true or false. There are only two boolean literals, **true** and **false** and three operators. There is one unary operator, ~, and two binary operators, **and** and **or**. They are defined by the truth table below.

a	b	~a	a or b	a and b
true	false	false	true	false
false	true	true	true	false
true	true	false	true	true
false	false	true	false	false

The precedence of the operators is important and is defined in descending order as

~
and
or

Thus

~a **or** b **and** c

is equivalent to

(~a) **or** (b **and** c)

This is reflected in the syntax rules which are

<expression> ::= <exp1>[**or**<exp1>]*
<exp1> ::= <exp2>[**and**<exp2>]*
<exp2> ::= [~]<exp3>

<expression> : **bool or** <expression> : **bool => bool**
<expression> : **bool and** <expression> : **bool => bool**
[~]<expression> : **bool => bool**

The evaluation of a boolean expression in S-algol is non-strict. That is, in the left to right evaluation of the expression, as soon as the result is found, no more computation is performed on the expression. For example

true or <expression>

gives the value **true** without evaluating <expression> and

false and <expression>

gives the value **false** without evaluating <expression>.

5.2 Comparison Operators

Expressions of type boolean can also be formed by some other binary operators. For example, a = b is either true or false and is therefore boolean in nature. The operators are called the comparison operators and are

<	less than
≤	less than or equal to (written <=)
>	greater than
≥	greater than or equal to (written >=)
=	equal to
≠	not equal to (written ~=)
is	a pntr has the given class
isnt	a pntr does not have the given class

The syntactic rules for these are

<exp2> ::= [~]<exp3>[<rel_op><exp3>]
<rel_op> ::= <eq_op> | <compar_op> | <type_op>

t : nonvoid, <expression> : t <eq_op> <expression> : t => bool
where <eq_op> ::= = | ≠

t : ordered,<expression> : t <compar_op> <expression> : t => bool
where <compar_op> ::= < | ≤ | > | ≥

<expression> : pntr <type_op><identifier> => bool
where <type_op> ::= **is** | **isnt**

Note that the operators <, ≤, > and ≥ are defined on integers, reals and strings whereas = and ≠ are defined on all S-algol data types. Their interpretation for these data types is given with each data type as it is introduced below. The operators **is** and **isnt** are for testing a structure class.

5.3 Arithmetic Expressions

Arithmetic may be performed on data objects of type integer and real. The syntax of arithmetic expressions is

<exp3> ::= <exp4>[<add_op><exp4>]*
<exp4> ::= <exp5>[<mult_op><exp5>]*
<exp5> ::= [<add_op>]<exp6>
<add_op> ::= + | -
<mult_op> ::= <star> | / | **div** | **rem**

t : arith, <expression> : t <add_op> <expression> : t => t
t : arith, <add_op> <expression> : t => t

<expression> : int <int_mult_op> <expression> : int => int where <int_mult_op> ::= <star> | **div** | **rem**

<expression> : real <real_mult_op><expression> : real => real where <real_mult_op> ::= <star> | /

The operators mean

+	addition
-	subtraction
*	multiplication
/	real division
div	integer division throwing away the remainder
rem	remainder after integer division

In both **div** and **rem** the result is negative only if exactly one of the operands is negative.

Some examples of arithmetic expressions are

a + b 3 + 2 1.2 + 0.5 -2 + a / 2.0

The language provides automatic coercion from integer to real where necessary.

5.4 Arithmetic Precedence Rules

The order of evaluation of an expression in S-algol is from left to right and based on the precedence table

* / **div** **rem**
\+ -

That is, the operations *, /, **div**, **rem** are always evaluated before the binary versions of + and -. However, if the operators are of the same precedence then the expression is evaluated left to right. For example

6 **div** 4 **rem** 2 gives the value 1

Brackets may be used to override the precedence of the operator or to clarify an expression.

<exp6> ::= (<clause>)

t : nonvoid,(<clause> : t) => t

For example

3 * (2 - 1) yields 3 not 5

5.5 String Expressions

There is only one string operator ++ defined on strings. It concatenates the two operand strings to form a new string. For example

"abc" ++ "def"

results in the string

"abcdef"

The syntax rule is

<exp4> ::= <exp5>[++<exp5>]*

<expression> : string ++ <expression> : string => string

A new string may be formed by selecting a substring of an existing string. For example, if s is the string "abcdef" then s(3 | 2) is the string "cd". That is, a new string is formed by selecting two elements from s starting at element 3. The syntax rule is

<exp6> ::= <expression>[(<clause><bar><clause>)]

<expression> : string(<clause> : int <bar> <clause> : int) => string

For the purposes of substring selection the first character in a string is numbered 1. The selection values are the start position and the length respectively.

The characters in a string are ordered according to the ASCII character code. Thus

"a" < "z"

To compare two strings, the characters are compared in pairs, one from each string, from left to right. Two strings are considered equal only if they have the same characters in the same order and are of the same length. Otherwise they are not equal.

The null string is less than any other string. Thus the less-than relation can be resolved by taking the characters pair by pair in the two strings until one is found to be less than the other. This is the lexicographic order commonly used in dictionaries. The other relations can be resolved by using '=' and '<'.

5.6 Picture Expressions

The picture drawing facilities of S-algol allow the user to produce line drawings in two dimensions. The system provides an infinite two dimensional real space. Altering the relationship between different parts of the picture is performed by mathematical transformations which means that pictures are usually built up of a number of sub-pictures. The syntax of picture expressions is

<exp4> ::= <exp5>[<pic_op><exp5>]*
<pic_op> ::= ^ | &

<expression> : pic <pic_op><expression> : pic => pic

<picture_constr> ::= **shift**<clause>**by**<clause>,<clause> |
scale<clause>**by**<clause>,<clause> |
rotate<clause>**by**<clause> |
colour<clause>**in**<clause> |
text<clause>**from**<clause>,<clause>**to**<clause>,<clause> |
<lsb><clause>,<clause><rsb>

shift<clause> : pic **by**<clause> : real,<clause> : real => pic
scale<clause> : pic **by**<clause> : real,<clause> : real => pic
rotate<clause> : pic **by**<clause> : real => pic
colour<clause> : pic **in**<clause> : pixel => pic
text<clause> : string **from**<clause> : real,<clause> : real
to<clause> : real,<clause> : real => pic
<lsb><clause> : real,<clause> : real <rsb> => pic

In a line drawing system the simplest picture is a point. For example, the expression

[0.1,2.0]

defines the point 0.1,2.0.

Points in pictures are implicitly ordered. A binary operation on pictures operates between the last point of the first picture and the first point of the second. The resulting picture has as its first point, the first point of the first picture, and as its last, the last point of the second.

There are two infix picture operators. They are '^', which joins one picture to another by a straight line from the last point of the first picture to the first point of the second, and '&' which includes one picture in another. The other transformations and operations on pictures are

shift — The new picture consists of the points obtained by adding the x and y shift values and the x and y co-ordinates of the points in the old picture. The ordering of the points is preserved.

scale — The new picture consists of the points obtained by multiplying the x and y scale values with the x and y co-ordinates of the points in the old picture, respectively. The ordering of the points is preserved.

rotate — The new picture consists of the points obtained by rotating the x and y co-ordinates of the points in the old picture about the origin by the angle indicated in degrees. The ordering of the points is preserved.

colour — The new picture is the old one in a new colour.

text — Form a picture consisting of the text string. The two co-ordinates represent the start and end points of the string which will be scaled to fit.

5.7 Pixel Expressions

Pixels may be concatenated to produce another pixel of a greater depth using the operator '&'.

<exp4> ::= <exp5>[&<exp5>]*

<expression> : **pixel** &<expression> : **pixel** => **pixel**

A pixel has depth representing the number of planes in the pixel. The planes are numbered from 0 and new pixels can be formed from subpixels of others. The syntax is

<exp6> ::= <expression>[(<clause><bar><clause>)]

<expression> : **pixel** (<clause> : **int** <bar><clause> : **int**) => **pixel**

For example

let b = **on** & **off** & **off** & **on**
b (1 | 2) is the pixel **off** & **off**

This last expression is interpreted as the pixel formed by starting at plane 1 in 'b' and selecting 2 planes.

5.8 Precedence Table

The full precedence table for S-algol is now

/ * **div** **rem** ^ &
+ - ++
~
= ≠ < ≤ > ≥ **is** **isnt**
and
or

5.9 Other Expressions

All the S-algol operators have now been described but we have not exhausted all the ways of writing down expressions. There are vector expressions, pntr expressions, vector and structure indexing expressions, if and case expressions as well as block expressions and the fact that literals, identifiers and procedure calls on their own constitute expressions. All of these are dealt with elsewhere in this manual.

6. Declarations

Declarations are of four kinds defined by the following syntax:

<declaration> ::= <let_decl> | <structure_decl> | <proc_decl> | <forward>

<declaration> => void

Procedure, forward and structure declarations are dealt with in later chapters.

6.1 Identifiers

In S-algol an identifier may be given to a data object, a procedure parameter, a structure field and a structure class. An identifier may be formed by the syntactic rule

<identifier> ::= <id> | <standard_id>

<id> ::= <letter>[<id_follow>]

<id_follow> ::= <letter>[<id_follow>] | <digit>[<id_follow>] | .[<id_follow>]

That is, an identifier consists of a letter followed by any number of dots, letters or digits. For example

x1 ron look.for.record1 Ron

Note that case is significant in identifiers. The standard identifiers are those predefined for the user. They are given in section 11.2.

6.2 Variables, Constants and Declaration of Data Objects

Before an identifier can be used in S-algol it must be declared. The action of declaring a data object associates an identifier with a location of a certain type which can hold values that the identifier may take. In S-algol the programmer may specify whether the value is constant or variable. A constant may be manipulated in exactly the same manner as a variable except that it may not be updated.

When introducing an identifier the programmer must indicate the identifier, the type of the data object, whether it is variable or constant and its initial value. Variables are declared by

<let_decl> ::= **let** <identifier><init_op><clause>

let <identifier><init_op><clause> : nonvoid => void

For example

let a := 1

introduces an integer variable with initial value 1. Notice that the compiler deduces the type.

A constant is declared by

let <identifier> = <clause>

For example

let discrim = b * b - 4.0 * a * c

introduces a real constant with the calculated value. The compiler will detect and flag as an error any attempt to assign to a constant.

6.3 Sequences

A sequence of clauses is made up of any mixture, in any order, of declarations and clauses. The type of the sequence is the type of the last clause or declaration. If there is more than one clause in a sequence then all but the last must be of type void.

<program> ::= <sequence>?

<sequence> ::= <declaration>[;<sequence>] |
<clause>[;<sequence>]

<sequence> : void => void
t : type, <declaration> : void ;<sequence> : t => t
t : type, <clause> : void ;<sequence> : t => t

6.4 Brackets

Brackets are used to make a sequence of clauses and declarations into a single clause.

<exp6> ::= **begin**[<sequence>]**end** |
{[<sequence>]}

begin end => void
{ } => void
t : type, **begin**[<sequence> : t]**end** => t
t : type, {[<sequence> : t]} => t

The {} method is there to allow a clause to be written clearly on one line. For example

let i := 2
for j = 1 **to** 5 **do** { i := i * i ; **write** i }

However, if the clause is longer than one line the first alternative should be used for greater clarity. Nonvoid blocks are sometimes called block expressions.

6.5 Scope Rules

Any identifier that is declared has its scope limited to the following sequence. This means that the scope of an identifier starts immediately after the declaration and continues up to the next unmatched } or **end**. If the same identifier is declared in an inner sequence, then while the inner identifier is in scope the outer one is not.

7. Clauses

The expression is a special type of clause which allows the operators in the language to be used to produce data objects. There are other clauses in S-algol which allow the data objects to be manipulated and some which are there to control the flow of the program.

7.1 Assignment Clause

The assignment clause has the following syntax.

<clause>::= <name>:=<clause>

t : nonvoid, <name> : t :=<clause> : t => void

For example

discriminant := b * b - 4.0 * a * c

gives 'discriminant' the value of the expression on the right. Of course, the identifier must have been declared as a variable and not a constant. The clause alters the value denoted by the identifier.

7.2 if Clause

There are two forms of the **if** clause defined by

<clause>::= **if**<clause>**do**<clause> |
if<clause>**then**<clause>**else**<clause>

if <clause> : bool **do** <clause> : void => void
t : type, **if** <clause> : bool **then** <clause> : t **else** <clause> : t => t

In the single pronged version, if the condition after the **if** is true then the clause after the **do** is executed. For example

if a < b **do** a := 3

The second version allows a choice between two actions to be made. If the first clause is true the second clause is executed, otherwise the third clause is executed. Notice that the second and third clauses are of the same type and the result is of that type. For example

if x = 0 **then** y := 1 **else** x := y - 1
let temp = **if** a < b **then** 1 **else** 5

7.3 case Clause

The **case** clause is a generalisation of the **if** clause which allows the selection of one item from a number of possible ones. The syntax is

<clause>::= **case**<clause>**of**<case_list>**default** :<clause>
<case_list> ::= <clause_list>:<clause>;[<case_list>]

t : type ; t1 : nonvoid, **case** <clause> : t1 **of** <case_list>
default : <clause> : t => t
where <case_list> ::= <clause_list>:<clause> : t ; [<case_list>]
where <clause_list> ::= <clause> : t1 [,<clause_list>]

An example of the use of the **case** clause is

```
case next.car.colour of
1,4     :     "green"
2       :     "blue"
default :     "any"
```

The value 'next.car.colour' is compared in strict order, i.e left to right, top to bottom, with the expressions on the left hand side of the colon. When a match is found the clause on the right hand side is executed. Control is then transferred to the next clause after the **case** clause. If no match is found then the default clause is executed. The above **case** clause has result type **string**.

7.4 repeat ... while ... do Clause

There are three forms of this clause which allow loops to be set up with the test at the start, the end or the middle of the loop. The three forms are encapsulated in the two productions.

```
<clause>::=  repeat<clause>while<clause>[do<clause>] |
                  while<clause>do<clause>

repeat <clause> : void while <clause> : bool
                  [do <clause> : void] => void
while <clause> : bool do <clause> : void => void
```

In each of the three forms the loop is executed until the boolean clause is **false**. The **while do** version is used to perform a loop zero or many times whereas the **repeat while** is used for one or many times.

7.5 for Clause

The **for** clause is included in the language as a bit of syntactic sugar where it is known in advance how many times the loop will be executed. It is defined by

```
<clause>::=  for<identifier>=<clause>to<clause>[by<clause>]do<clause>

for <identifier>=<clause> : int to <clause> : int
                  [by<clause> : int ]do<clause> : void => void
```

The clauses are the initial value, the limit, the increment and the clause to be repeated respectively. The first three are of type int and are calculated only once at the start. If the increment is 1 then the **by** clause may be omitted. The identifier or control constant is declared at the start of the void clause taking on the values defined by initial value, increment and limit. The scope of 'i' is the void clause. An example of the **for** clause is

```
let factorial := 1 ; let n = 8
for i = 1 to n do factorial := factorial * i
```

With a positive increment, the **for** loop terminates when the control constant is initialised to a value greater than the limit. With a negative increment, the **for** loop terminates when the control constant is initialised to a value less than the limit.

7.6 abort Clause

The reserved word **abort** simply stops the program when used.

<clause>::= **abort**

abort => void

8. Procedures

8.1 Declarations and Calls

Procedures in S-algol constitute the abstractions over expressions, if they return a value, and clauses of type void if they do not. In accordance with the principle of correspondence, any method of introducing an identifier in a declaration has an equivalent form as a parameter.

Thus, in declarations of data objects, giving an identifier an initial value is equivalent to assigning the actual parameter value to the formal parameter. Since this is the only type of declaration for data objects in the language, it is also the only parameter passing mode and is commonly known as **call by value**.

Like declarations, the formal parameters representing data objects must have an identifier, a type and an indication of whether they are variable or constant. A procedure which returns a value must also specify its return type. The scope of the formal parameters is from their declaration to the end of the procedure clause. Procedures are defined with the following syntax.

<proc_decl> ::= **procedure**<identifier>[([<parameter_list>]
[<arrow><type_id>])] ; <clause>

<parameter_list> ::= <parameter>[;<parameter_list>]

<parameter> ::= <type1><identifier_list> |
<structure_decl> |
<proc_type><identifier_list>

<proc_type> ::= ([<ptype_list>][<arrow><type_id>])

<ptype_list> ::= <type1>[,<ptype_list> |
<proc_type>[,<ptype_list>] |
<s_decl>[,<ptype_list>]

<s_decl> ::= **structure**(<type1>[,<type1>]*)

<type1> ::= [**c**]<type_id>

<type_id> ::= **int** | **real** | **bool** | **string** | **pixel** | **pic** | **pntr** | **file** | <star><type1> |
#pixel | **#cpixel**

Thus we may declare the integer identity procedure, which we will call 'int.id' by

procedure int.id(**int** n -> **int**) ; n

The syntax of the procedure call is

<exp6> ::= <application>

<application> ::= <identifier>[([<clause_list>])]

There must be a one-to-one correspondence between the actual and formal parameters and their types. Thus to call the integer identity procedure given above we could use

int.id (42)

which will evaluate to the integer 42. The type of 'int.id' is written (**int** -> **int**).

To complete the Principle of Correspondence for procedures, the parameters may be made constant. Variable parameters may be assigned to but since they are local variables this only has local effect. Constant parameters may not be assigned to. All parameters are passed by value. For example, the parameter in 'int.id' is not assigned to and is more appropriately a constant. Therefore the declaration should be

procedure int.id(**cint** n -> **int**) ; n

Note that the constancy of the parameter is not part of the type, a notion that is important when deciding type equivalence.

8.2 forward Declarations

In S-algol all names must be declared before they can be used and a identifier comes into scope immediately after its declaration. This is awkward when recursive procedure definitions are involved. Therefore in the case of the procedure, the identifier comes into scope after the parameter list has been specified allowing procedures to call themselves.

A forward declaration is required for mutually recursive procedures. Its syntax is

<forward> ::= **forward**<identifier><proc_type>

This is merely an aid to the compiler. However, there is one rule with forward declarations that has nothing to do with the one pass nature of the compiler but with the decision to allow clauses and declarations to be freely mixed. To avoid the possibility of an uninitialised identifier being used by jumping over a declaration, a forward declaration and the actual declaration may only be separated by structure class declarations and other procedure declarations. In other words its use is restricted to mutual recursion situations.

9. Aggregates

S-algol allows the programmer to group together data objects into larger aggregate objects which may then be treated as single objects. There are three such object types in S-algol : vectors, structures and images. If the constituent objects are of the same type a vector is used and a structure for non-homogeneous collections. Images are collections of pixels. Vectors, structures and images have the full **civil rights** as any other data object in S-algol.

All aggregate data objects in S-algol have pointer semantics. That is, when a aggregate data object is created, a pointer to the locations that make up the object is also created. The object is always referred to by the pointer which may be passed around by assignment and tested for equality. The location containing the pointer and the constituent parts of the aggregate data object may be independently constant or variable.

<exp6> ::= <value_constructor>

<value_constructor> ::= <vector_constr> | <image_constr> |
<subimage_constr> | <picture_constr>

9.1 Vectors

9.1.1 Creation of Vectors

A vector provides a method of grouping together objects of the same type. Since S-algol does not allow undefined values all the initial values of the elements must be specified. The syntax is

<vector_constr> ::= **vector**<range>**of**<clause> |
@<clause>**of**<type1><lsb><clause_list><rsb>
<range> ::= <clause> :: <clause>[,<range>]

t : nonvoid, **vector**<range>**of**<clause> : t => *t
t : nonvoid,@<clause> : int **of**<type1><lsb><clause> : t
[,<clause> : t]* <rsb> => *t
where <range> ::= <clause> : int :: <clause> : int[,<range>]

For example

@1 **of int** [1,2,3,4]

is a vector of integers, i.e *int, with lower bound 1 and values 1, 2, 3 and 4. The elements are variable. Also

let abc := @1 **of int**[1,2,3,4]

introduces a variable 'abc' of type *int and that initial value.

Multi dimensional vectors, which are not necessarily rectangular, can also be created. For example

```
let Pascal = @1 of c*cint[   @1 of cint [ 1 ],
                             @1 of cint [ 1,1 ],
                             @1 of cint [ 1,2,1 ],
                             @1 of cint [ 1,3,3,1 ],
                             @1 of cint [ 1,4,6,4,1 ],
                             @1 of cint [ 1,5,10,10,5,1 ] ]
```

'Pascal' is of type **c*c*cint**. It is constant as are all its elements. This is a fixed table. The use of the word **cint** indicates that the elements are to be constant.

The above form of vector expression is sometimes very tedious to write especially for large rectangular vectors with a common initial value. Therefore another form of vector expression is available. For example

vector -1 :: 3 **of** -2

produces a five element integer vector with all the elements variable and initialised to -2. The lower bound of this vector is -1 and the upper bound is 3. The element initialising expression is evaluated only once and assigned to each of the elements. The elements are variable.

9.1.2 upb and lwb

Since vectors may be assigned, it is often necessary to interrogate the vector to find its bounds. The reserved words **upb** and **lwb** are provided in S-algol for this purpose.

<exp6> ::= <bounds_op>(<clause>)
<bounds_op> ::= **lwb** | **upb**

<bounds_op>(<expression> : *nonvoid) => int

An example of the use of thes procedures is

let a = **vector** 1 :: 19 **of** -42

upb (a) yields 19

9.1.3 Indexing

To obtain the elements of a vector, indexing is used. For vectors the index is always an integer value. The syntax is

<exp6> ::= <expression>(<dereference>)
<dereference> ::= <clause_list>

t : nonvoid,<clause> : *t (<clause> : int) => t

t : nonvoid,<clause> : *ct (<clause> : int) => t

For example

a (3 + 4)

Before indexing is performed, the bounds of the vectors are checked against the index.

9.1.4 Equality and Equivalence

Two vectors are equal if they are the same vector. That is the same pointer. Two vectors are type equivalent if they have the same dimensionality and equivalent base types. Notice that the bounds are not part of the type. In order to ensure that constancy can always be checked statically objects of type *T and *cT are not assignment compatible. That is, they may not be assigned to the other or substituted for the other in parameterisation.

9.2 Structures

9.2.1 Creation of Structures

Objects of different types can be grouped together into a structure. The fields of a structure have identifiers that are unique within the current scope. The structures are sets of labelled cross products from the value space.

Structures may be created from a declaration. The syntax of structure class declarations is

<structure_decl > ::= **structure**<identifier>[([<field_list>])]
<field_list> ::= <type1><identifier_list>[;<field_list>]

For example, we may declare such a type by

structure person(**cstring** name ; **bool** sex ; **int** age, height)

This declares a structure class, 'person', with four fields of type string, bool, int and int respectively. The string field is constant. It also declares the field identifiers, 'name', 'sex', 'age' and 'height'.

To create a structure from a declaration we use the class identifier as a constructor, followed by the initialising values for the fields. The syntax is

<structure_creation> ::= <identifier>[([<clause_list>])]

For example

let ron = person("Ronald Morrison", **true**, 42, 175)

creates the appropriate structure. The initialising values must be in one-one correspondence with the structure type declaration. Each dynamic use of a class identifier with fields creates a pointer to a new instance of the structure class.

9.2.2 Equality and equivalence

Two structures are equal if they have the same pointer. All pointers to structures belong to the universal union pntr.

9.2.3 Indexing

To obtain the field of a structure, the field identifier is used as an index. For example

ron (age)

yields 42 from the above. For the indexing operation to be legal the structures must contain a field with that identifier.

A comma notation may be used for vectors or structures when the elements or fields are themselves pointers or vectors. The indexing of vectors and structures may therefore be freely mixed. For example, if 'v' is a vector of vectors of persons then 'v(i)(j)(name)' and 'v(i,j,name)' and 'v(i,j)(name)' would be equivalent expressions.

9.3 Images

9.3.1 Creation of Images

An image is a rectangular grid of pixels. Images may be created and manipulated using the raster operations provided in the language. The creation of images is defined by

```
<image_constr>      ::=   image<clause>by<clause>of<clause>
<subimage_constr>   ::=   limit<clause>[to<clause>by<clause>]
                                        [at<clause>,<clause>]

image<clause> : int by<clause> : int of<clause> : pixel =>#pixel
t : image,limit<clause> : t [to<clause> : int by<clause> : int]
                            [at<clause> : int ,<clause> : int] => t
```

The integer values above must be ≥ 0 and are subjected to an upper bound check. An image is a 3 dimensional object made up of a rectangular grid of pixels. To form an image we could write

let c = **image** 5 **by** 10 **of on**

which creates 'c' with 5 pixels in the X direction and 10 in the Y direction all initially on. The origin of all images is 0,0 and in this case the depth is 1.

Full three dimensional images may be formed by, for example

let d = **image** 64 **by** 32 **of on** & **off** & **on** & **on**

Images are first class data objects and may be assigned, passed as parameters or returned as results. For example

let b := a

will assign the image 'a' to the new one 'b'. In order to map the operations usual on bitmapped screens, the assignment does not make a new copy of 'a' but merely copies the pointer to it. Thus the image acts like a vector or pointer on assignment.

There are 8 raster operations which may be used as described in the following syntax.

```
<clause>::=     <raster>

<raster> ::=    <raster_op><clause>onto<clause>
<raster_op>     ::=   ror | rand | xor | copy | nand | nor | not | xnor

<raster_op><clause> : image onto<clause> : #pixel => void
```

thus

xor b **onto** a

performs a raster operation of 'b' onto 'a' using **xor**. Notice that 'a' is altered 'in situ' and 'b' is unchanged. Both images have origin 0,0 and automatic clipping at the extremities of the destination image is performed.

The raster operations are performed by considering the images as bitmaps and altering each bit in the destination image according to the source bit and the operation. The operations mean

ror	inclusive or
rand	and

xor	exclusive or
copy	overwrite
nand	not and
nor	not inclusive or
not	not the source
xnor	not exclusive or

9.3.2 Indexing

The limit operation allows the user to set up windows in images. For example

let c = **limit** a **to** 1 **by** 5 **at** 3,2

sets 'c' to be that part of 'a' which starts at 3,2 and has size 1 by 5. 'c' has an origin of 0,0 in itself and is therefore a window on 'a'.

Rastering sections of images on to sections of other images can be performed by, for example

xor **limit** a **to** 1 **by** 4 **at** 6,5 **onto**
limit b **to** 3 **by** 4 **at** 9,10

Automatic clipping on the edges of the limited region is performed. If the starting point of the limited region is omitted 0,0 is used and if the size of the region omitted then it is taken as the maximum possible. That is, from the starting point to the edges of the host image. Limited regions of limited regions may also be defined.

If the source and destination images overlap, then the raster operation is performed in such a manner that each bit is used as a source before it is used as a destination.

The standard identifier 'screen' is an image representing the output screen. Performing a raster operation onto the image 'screen' alters what may be seen by the user. The standard identifier 'cursor' is an image representing the cursor. The cursor may be altered in the same manner as any image.

9.3.3 Depth Selection

In systems that support multiple planes the standard identifiers 'screen' and 'cursor' will have a depth greater than 1. All the operations that we have already seen on images(raster, limit and assignment) work more generally with depth. Thus the raster operations perform the raster plane by plane in one to one correspondence between source and destination. Automatic depth clipping at the destination is performed and if the source is too small to fill all the destination's planes then these planes will remain unaltered. The limit and assignment operations also work with the depth of the image.

The depth of the image may be restricted by the depth selection operation. For example

let b = a(1|2)

yields 'b' which is an alias for that part of 'a' which has the two depth planes 1 and 2. 'b' has depth origin 0 and dimensions 64 by 32.

The full syntax of the depth selection operation is

<exp6> ::= <expression>[(<clause><bar><clause>)]

t ; image,<expression> ; t (<clause> ; int <bar><clause> ; int) => t

9.3.4 Equality and Equivalence

Two images are equal if they have the same pointer. All images have equivalent types. In order to ensure that constancy can always be checked statically objects of type #pixel and #cpixel are not assignment compatible. That is, they may not be assigned to the other or substituted for the other in parameterisation.

10. Input and Output

The S-algol I/O system operates on files. A file is a collection of data. Two files, standard input and standard output which have the reserved identifiers s.i and s.o respectively, are of special interest. They represent in any implementation the standard input and output devices. In the language s.i and s.o are variables of type file. Other files may be created but this is implementation dependent.

10.1 Input

The read functions regard the file as a series of ascii characters. The functions mean

read	read the next character in the file.
readi	skip tab, space and newline characters and read an integer literal.
readr	skip tab, space and newline characters and read a real literal.
readb	skip tab, space and newline characters and read a boolean literal.
reads	skip tab, space and newline characters and read a string literal.
peek	look at the next character and return it without reading it.
read.a.line	read from the current position up to a newline symbol. Give the result
read.byte	read one 8 bit byte as an integer.
read.16	read 16 bits to form an integer
read.32	read 32 bits to form an integer
eof	test for end of file.

Of these functions read, reads, peek and read.a.line are of type string, readi, read.byte, read.16 and read.32 of type int, readr of type real and readb and eof of type bool.

The characters in the file are read until the literal of the required type is formed.

The file clause in brackets may be omitted if the standard input is used. Foe example

let a = **readr**

'a' is now a constant of type real with the value of the real literal just read from the standard input. In the input stream, spaces, tabs and newlines before numbers are elided.

let c = **reads** (s.i)

reads a string literal, also from the standard input, and initialises the constant 'c' to that value.

The read functions perform their activity on files of ascii characters. It is sometimes useful to read 8,16 or 32 bits from a file rather than an ascii character. The procedures read.byte, read.16 and read.32 are provided for this. For example

let q = **read.byte** (in)

will read an 8 bit byte from the file 'in' and form it into an integer.

let q = **read.16**(in)

will read 16 bits from the file 'in' and form them into an integer. The significance of the individual bits is determined by the computer's architecture.

let q = **read.32**(in)

will read 32 bits from the file 'in' and form them into an integer. The significance of the individual bits is determined by the computer's architecture.

10.2 Output

Output to files may be performed in S-algol by the following syntax:

<clause> ::= <write_clause>

<write_clause> ::= **write**<write_list> |
output <clause>,<write_list> |
out.byte<clause>,<clause>,<clause> |
out.16<clause>,<clause>,<clause> |
out.32<clause>,<clause>

<write_list> ::= <clause>[:<clause>][,<write_list>]

<write_clause> => void
where<write_list>::= <clause>:writeable [:<clause> : int][,<write_list>]

output<clause> : file,<write_list> => void
where<write_list>::= <clause>:writeable [:<clause> : int][,<write_list>]

out.byte<clause> : file,<clause> : int,<clause> : int => void

out.16<clause> : file,<clause> : int,<clause> : int => void

out.32<clause> : file,<clause> : int => void

write allows output to the standard output, s.o, only whereas output takes a file descriptor as the first data object. For example

write "I am O.K.",32160,3.4e-2,"Done"

will write that list of objects to the standard output and

write 13.2 : 16, 3 : 10, 152 : 3

will write 13.2, 3 and 152 to the standard output right justified in fields of width 16, 10 and 3 respectively.

Note that the field size may be any integer expression, allowing variable formats. If the field size is missing when writing out reals or integers then 'r.w' and 'i.w' respectively are used as

the field sizes. If the object cannot fit into the field size then the object is written out in its exact size.

The operations out.byte, out.16 and out.32 are the reverse of read.byte, read.16 and read.32 respectively. For example

out.byte out.file,a,0

will write the zeroth (least significant) byte of 'a' to the file out.file.

out.16 out.file,a,0

will write the zeroth (least significant) 16 bits of 'a' to the file out.file.

out.32 out.file,a

will write the 32 bits of 'a' to the file out.file.

10.3 i.w, s.w and r.w

Types int and real have one other facility. The predefined integer variables

i.w	integer width initially 12
s.w	space width initially 2
r.w	real width initially 14

may be used to control output. s.w spaces are written out after any integer or real. i.w and r.w are used to set the field sizes for integers and real respectively, but may be overridden by the field size in the write clause. If the number fits neither then the exact size is written.

10.4 End of File

A test for end of file can be made using the reserved word **eof**. For example

while ~eof do write read

copies the input stream to the output stream until the end of the input stream is reached.

11. Standard Procedures and Identifiers

11.1 Standard Procedures

procedure sqrt(**creal** x -> **real**)
! the non-negative square root of x where x ≥ 0.

procedure exp(**creal** x -> **real**)
! e to the power x.

procedure ln(**creal** x -> **real**)
! the logarithm of x to the base e where x > 0.

procedure sin(**creal** x -> **real**)
! sine of x(radians).

procedure cos(**creal** x -> **real**)
! cosine of x(radians).

procedure atan(**creal** x -> **real**)
! arctangent (radians) of x where -π / 2 < atan(x) < π / 2.

procedure code(**cint** n -> **string**)
! string of length 1 where s(1|1) = character with numeric code abs(n rem 128).

procedure decode(**cstring** s -> **int**)
! numeric code for s(1|1).

procedure truncate(**creal** x -> **int**)
! the integer i such that | i | ≤ | x | < | i | + 1 where i * x ≥ 0.

procedure rabs(**creal** x -> **real**)
! the absolute value of real number x.

procedure abs(**cint** n -> **int**)
! the absolute value of integer n.

procedure length(**cstring** s -> **int**)
! the number of characters in the string s.

procedure eformat(**creal** n ; **cint** w,d -> **string**)
! the string representing n with w digits before the decimal point and d digits after
! with an exponent.

procedure fformat(**creal** n ; **cint** w,d -> **string**)
! the string representing n with w digits before the decimal point and d digits after.

procedure gformat(**creal** n -> **string**)
! the string representing n in eformat or fformat whichever is suitable.

procedure letter(**cstring** s -> **bool**)
! length(s) = 1 and
! s ≥ "A" and s ≤ "Z" or s ≥ "a" and s ≤ "z".

procedure digit(**cstring** s -> **bool**)
! length(s) = 1 and
! s ≥ "0" and s ≤ "9".

procedure iformat(**cint** n -> **string**)
! integer n as a string of characters.

procedure shift.l(**cint** value,count -> **int**)
! shift the first parameter left 'count' places bringing in zeros at the low order end and
! dropping digits at the left.

procedure shift.r(**cint** value,count -> **int**)
! shift the first parameter right 'count' places bringing in zeros at the high order end and
! dropping digits on the right.

procedure b.and(**cint** value1,value2 -> **int**)
! logical (bitwise) 'and' of value1 and value2.

procedure b.or(**cint** value1,value2 -> **int**)
! logical (bitwise) 'or' of value1 and value2.

procedure b.not(**cint** value -> **int**)
! logical (bitwise) 'not' of value.

procedure b.xor(**cint** value1,value2 -> **int**)
! logical (bitwise) 'exclusive or' of value1 and value2.

procedure fiddle.r(**creal** n -> ***int**)
! split a real into a vector of two integers.

procedure find.substr(**cstring** target,substring -> **int**)
! return the starting position of string 'substring' in 'target', zero otherwise.

procedure random(**cint** x -> **int**)
! takes the non-zero seed 'x' and produces a non-zero random number between -maxint - 1 !
and maxint.

procedure open(**cstring** name ; **int** access.mode -> **file**)
! Open the existing file named. Access mode is '0' for read only, '1' is write only,
! '2' is reading and writing. Return the file descriptor.

procedure create(**cstring** name -> **file**)
! create the named file and return the file descriptor.

procedure close(**cfile** f)
! close the file f.

procedure flush(**cfile** f)
! Clear out the internal buffer for file f. This should be done before closing the file
! after writing.

procedure seek(**cfile** f ; **cint** offset,key)
! This moves the current position in file f 'offset' bytes through the file with 'key'
! interpreted as:-
! '0' - move to offset bytes from start of file.
! '1' - move offset bytes from current position.
! '2' - move back offset bytes from end of file.

procedure trace
! Print a snapshot of the current procedure calls.

procedure date(-> **string**)
! gives the date and time in a format determined by the implementation.

procedure time(-> **int**)
! returns the number of clock ticks (1/60 second) passed since a reference time t = 0.

procedure draw(**c#pixe**l i ; **cpic** p ; **creal** x1,x2,y1,y2)
! draw the picture p on the image i.
! the picture is bounded by x1,x2,y1,y2 in its coordinate space.

procedure X.dim(**c#pixel** i -> **int**)
! return the x dimension of image i.

procedure Y.dim(**c#pixel** i -> **int**)
! return the y dimension of image i.

procedure locator(-> **pntr**)
! returns a structure containing information about the status of the mouse.
! the structure returned is a mouse(defined below).

procedure cursor.tip(**cpntr** the.tip -> **pntr**)
! make the effective tip of the cursor 'the.tip' return old tip
! both pointers are pointers to point.strc(**cint** point.x,point.y)

procedure cursor.on
! make the cursor track the mouse (the default state).

procedure cursor.off
! make the cursor invisible.

procedure Pixel(**c#pixel** i ; **cint** xpos,ypos -> **pixel**)
! return the pixel at xpos,ypos in i.

procedure constant.image(**c#pixel** i -> **#cpixel**)
! return a copy of image i with constant pixels.

procedure variable.image(**c#cpixel** i -> **#pixel**)
! return a copy of image i with variable pixels.

procedure fill(**c#pixel** i ; **cpixel** col ; **cint** xpos,ypos)
! seed fill image i from position xpos,ypos with the pixel col.

procedure make.menu(**c#pixel** title ; **c*c#pixel** entries ; **cbool** vertical -> **pntr**)
! Returns a 'menu' structure to pass to call.menu. 'vertical' indicates the menu orientation.
! The menu has a 'title' and a vector of icons called 'entries'

procedure call.menu(**cpntr** a.menu ; **cint** xpos,ypos -> **int**)
! Causes a menu to appear with its bottom left hand corner at position xpos,ypos relative to
! 'screen'. The integer returned indicates if the user made a selection. If not then the integer
! is one more than the upper bound of the vector of icons.

procedure depth(**c#pixel** i -> **int**)
! return the number of planes in image i.

procedure colour.map(**cpixel** p ; **cint** i)
! when pixel p is displayed the integer i will be sent to the display hardware.

procedure colour.of(**cpixel** p -> **int**)
! return the integer sent to the hardware when pixel p is displayed.

procedure string.to.tile(**cstring** the.string,font ; **cint** font.size -> **#pixel**)
! returns an image which contains 'the.string' in the 'font' in the 'font.size'.
! Any font.size may be specified - scaling is automatically performed.
! Any installed Mac font can be named.

procedure line.end(**c#pixel** i ; **pixel** p ; **cint** x,y,direct -> **int**)
! searches for the first pixel of colour p from position x,y in image i
! direct specifies the search direction
! odd numbers do not look at boundary pixels
! 0,1 - left ; 2,3 - right ; 4,5 - down ; 6,7 - up
! return the position of the pixel or 1 past the position last searched

procedure line(**c#pixel** i ; **cint** x1,y1,x2,y2,style)
! draws a line from x1,y1 to x2,y2 on image i
! style may be 0 - draw: set pixels to **on** ; 1 - erase: set pixels to **off** ; 2 - xor: invert pixels.

procedure clear.output
! Delete all text from the output window and clear graphics from the output window.

procedure plane.of(**c#pixel** i ; **cint** p -> ***int**)
! returns the p'th plane of image i.

procedure pixel.depth(**cpixel** p -> **int**)
! returns the number of planes in pixel p.

procedure input.pending(-> **bool**)
! returns **true** if a **read** or **peek** would complete immediately.

procedure interrupt(-> **bool**)
! returns **true** if an interrupt (control-\ on the Mac) has been received since the last call of
! this procedure or the start of the program.

procedure interrupt.on
! Enable interrupts (control-\ on the Mac) to be trapped.

procedure interrupt.off
! Disable interrupts (control-\ on the Mac) being trapped.

11.2 Standard Identifiers

A number of standard identifiers exist in the language. They are

r.w	variable initially 14
s.w	variable initially 2
i.w	variable initially 12
s.i	variable set to the standard input
s.o	variable set to the standard output
maxint	constant, the maximum integer
epsilon	constant, the largest real e such that 1.0 + e = 1.0
pi	constant, π
maxreal	constant, the largest real
screen	constant, #pixel representing the screen
cursor	constant, #pixel representing the cursor

Note that the minimum integer value is -maxint - 1 and the minimum real value is -maxreal.

11.3 S-algol Prelude

```
structure mouse( cint X.pos,Y.pos ; cbool selected ; c*cbool the.buttons )
! This is the structure returned by the locator standard procedure.

structure point.strc( cint point.x,point.y )
! This is the structure returned by the cursor.tip standard procedure.

structure trnsfrm.strc( cint trnsfrm ; cpntr mrtre ; creal trnsfrm.x,trnsfrm.y )
! The following 5 are used internally by the draw procedure.

structure culr.strc( cpntr nxtre ; cpixel shade )

structure oprtn.strc( cpntr lft,rght ; cint opoo )

structure scrbl.strc( cstring msge ; creal xxl,yyl,xxr,yyr  )

structure poin.strc( creal pnx,pny )

structure menu( c#pixel menu.image ; c*cint menu.dimensions.vector ; cbool vertical )
! This is the structure returned by the make.menu standard procedure and used by
! call.menu.

let Outline.error = "'n***** S-algol Run-Time Error *****'n'n"
let charry = @1 of *c*cint[ frst,scnd ]
let nilpic = text "" from 0,0 to 0,1
```

12. References

1. Adamson, I.A.T.
"The denotational semantics of S-algol". M.Sc. Thesis University of St Andrews. (1982).

2. Brown, A.L. & Dearle, A.
"Implementation Issues in Persistent Graphics". University Computing 8,2 (Summer 1986)

3. Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling". Ellis-Horwood Press. (1981)

4. Gunn, H.I.E. & Morrison, R.
"On the implementation of constants". Information Processing Letters 9,1 (July 1979),1-4.

5. Landin, P.J.
"The next 700 programming languages". Comm.ACM 9, 3 (1966), 157-164.

6. Ledgard, H.F. & Marcotty, M.
"A genealogy of control structures". Comm.ACM 18,11 (November 1975),629-639.

7. Morrison, R.
"On the Development of Algol". PhD Thesis, University of St Andrews, (1979).

8. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
"An integrated graphics programming environment". 4th UK Eurographics Conference, Gla

9. Strachey, C.
"Fundamental Concepts in Programming Languages". Oxford University, 1967.

10. Tennent, R.D.
"Language design methods based on semantic principles". Acta Informatica 8 (1977),97-112.

11. Turner D.A. & Morrison R.
"Towards portable compilers". University of St.Andrews TR/76/5 (1975).

12. Wirth, N.
"The programming language Pascal". Acta Informatica 1 (1971),35-63.

Appendix I

Context Free Syntax

Session:

<program> ::= <sequence>?

<sequence> ::= <declaration>[;<sequence>] |
<clause>[;<sequence>]

Declarations:

<declaration> ::= <let_decl> | <structure_decl> | <proc_decl> | <forward>

<let_decl> ::= **let** <identifier><init_op><clause>

<init_op> ::= = | :=

<structure_ decl > ::= **structure**<identifier>[([<field_list>])]

<field_list> ::= <type1><identifier_list>[;<field_list>]

<proc_decl> ::= **procedure**<identifier>[([<parameter_list>]
[<arrow><type_id>])] ; <clause>

<parameter_list> ::= <parameter>[;<parameter_list>]

<parameter> ::= <type1><identifier_list> |
<structure_decl> |
<proc_type><identifier_list>

<proc_type> ::= ([<ptype_list>][<arrow><type_id>])

<ptype_list> ::= <type1>[,<ptype_list> |
<proc_type>[,<ptype_list>] |
<s_decl>[,<ptype_list>]

<s_decl> ::= **structure**(<type1>[,<type1>]*)

<forward> ::= **forward**<identifier><proc_type>

<type1> ::= [**c**]<type_id>

<type_id> ::= **int** | **real** | **bool** | **string** | **pixel** | **pic** | **pntr** | **file** |
<star><type1> |
#pixel | **#cpixel**

<identifier_list> ::= <identifier>[,<identifier_list>]

Clauses:

<clause> ::= **if**<clause>**do**<clause> |

if<clause>**then**<clause>**else**<clause> |

repeat<clause>**while**<clause>[**do**<clause>] |

while<clause>**do**<clause> |

for<identifier>=<clause>**to**<clause>[**by**<clause>]**do**<clause>|

case<clause>**of**<case_list>**default** :<clause> |

<name>:=<clause> |

<write_clause> |

<raster> |

abort |

<expression>

<case_list> ::= <clause_list>:<clause>;[<case_list>]

<write_clause> ::= **write**<write_list> |
output <clause>,<write_list> |
out.byte<clause>,<clause>,<clause> |
out.16<clause>,<clause>,<clause> |
out.32<clause>,<clause>

<write_list> ::= <clause>[:<clause>][,<write_list>]

<raster> ::= <raster_op><clause>**onto**<clause>

<raster_op> ::= **ror** | **rand** | **xor** | **copy** | **nand** | **nor** | **not** | **xnor**

<clause_list> ::= <clause>[,<clause_list>]

Expressions:

<expression> ::= <exp1>[**or**<exp1>]*

<exp1> ::= <exp2>[**and**<exp2>]*

<exp2> ::= [~]<exp3>[<rel_op><exp3>]

<exp3> ::= <exp4>[<add_op><exp4>]*

<exp4> ::= <exp5>[<mult_op><exp5>]*

<exp5> ::= [<add_op>]<exp6>

<exp6> ::= <standard_exp> |

<literal> |

<value_constructor> |

(<clause>) |

begin[<sequence>]**end** |

{[<sequence>]} |

<expression>(<clause><bar><clause>) |

<expression>(<dereference>) |

<application> |

<structure_creation> |

<name> |

<bounds_op>(<clause>)

<dereference> ::= <clause_list>

<application> ::= <identifier>[([<clause_list>])]

<structure_creation> ::= <identifier>[([<clause_list>])]

<name> ::= <identifier> |

<expression>(<clause_list>)[(<clause_list>)]*

<bounds_op> ::= **upb** | **lwb**

Value constructors:

<value_constructor> ::= <vector_constr> | <image_constr> |

<subimage_constr> | <picture_constr>

<vector_constr> ::= **vector**<range>**of**<clause> |

@<clause>**of**<type1><lsb><clause>[,<clause>]*<rsb>

<range> ::= <clause>::<clause>[,<range>]

<image_constr> ::= **image**<clause>**by**<clause>**of**<clause>

<subimage_constr> ::=
limit<clause>[**to**<clause>**by**<clause>][**at**<clause>,<clause>]

<picture_constr> ::= **shift**<clause>**by**<clause>,<clause> |

scale<clause>**by**<clause>,<clause> |

rotate<clause>**by**<clause> |

colour<clause>**in**<clause> |

text<clause>**from**<clause>,<clause>**to**<clause>,<clause> |

<lsb><clause>,<clause><rsb>

Literals:

<literal> ::= <integer_literal> | <real_literal> | <boolean_literal> | <string_literal> | <pixel_literal> | <pntr_literal> | <file_literal>

<integer_literal> ::= [<add_op>]<int_literal>

<int_literal> ::= <digit>[<digit>]*

<real_literal> ::= <integer _literal>.[<int _literal>][e<integer_literal>]

<boolean_literal> ::= **true** | **false**

<string_literal> ::= <double_quote>[<char>]*<double_quote>

<char> ::= any ascii character except " and ' | <special_character>

<special_character> ::= <single_quote><special_follow>

<special_follow> ::= n | p | o | t | b | <single_quote> | <double_quote>

<pixel_literal> ::= **on**[&<pixel_literal>] | **off**[&<pixel_literal>]

<pntr_literal> ::= **nil**

<file_literal> ::= **nullfile**

Miscellaneous and microsyntax:

<lab> ::= <

<rab> ::= >

<lsb> ::= [

<rsb> ::=]

<star> ::= *

<bar> ::= |

<add_op> ::= + | -

<mult_op> ::= <int_mult_op> | <real_mult_op> | ++ | <pic_op> | <pixel_op>

<int_mult_op> ::= <star> | **div** | **rem**

<real_mult_op> ::= <star> | /

<pic_op> ::= ^ | &

<pixel_op> ::= &

<rel_op> ::= <eq_op> | <compar_op> | <type_op>

<eq_op> ::= = | ≠

<compar_op> ::= | < | ≤ | > | ≥ |

<type_op> ::= **is** | **isnt**

<arrow> ::= ->

<double_quote> ::= "

<single_quote> ::= '

<identifier> ::= <id> | <standard_id>

<id> ::= <letter>[<id_follow>]

<id_follow> ::= <letter>[<id_follow>] | <digit>[<id_follow>] | .[<id_follow>]

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m |

n | o | p | q | r | s | t | u | v | w | x | y | z |

A | B | C | D | E | F | G | H | I | J | K | L | M |

N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<standard_exp> ::= <standard_name>[(<clause>)]

<standard_name> ::= **upb | lwb | eof | read.a.line**
read | readi | readr | readb | peek | reads | read.name |
read.byte
read.16 | read.32

<standard_id> ::= r.w | i.w | s.w | s.o | s.i | maxint | maxreal |
epsilon | pi | cursor | screen

Appendix II

Type Rules

type arith **is** int | real

type ordered **is** arith | string

type writeable **is** ordered | bool

type literal **is** writeable | pixel | pntr | file

type image **is** #pixel | #cpixel

type nonvoid **is** literal | pic | image | vector

type vector **is** *nonvoid | *cnonvoid

type type **is** nonvoid | void

Program :

<sequence> : void ? => void

t : type, <declaration> : void ; <sequence> : t => t

t : type, <clause> : void ; <sequence> : t => t

Declarations :

<declaration> => void
where <let_decl> ::= **let**<init_op><clause> : nonvoid

Clauses :

if <clause> : bool **do** <clause> : void => void

t : type, **if** <clause> : bool **then** <clause> : t **else** <clause> : t => t

repeat <clause> : void **while** <clause> : bool [**do** <clause> : void] => void

while <clause> : bool **do** <clause> : void => void

for <identifier>=<clause> : int **to** <clause> : int
[**by**<clause> : int]**do**<clause> : void => void

t : type ; t1 : nonvoid, **case** <clause> : t1 **of** <case_list>
default : <clause> : t => t
where <case_list> ::= <clause_list>:<clause> ; t ; [<case_list>]
where <clause_list> ::= <clause> ; t1 [,<clause_list>]

t : nonvoid, name ; t := <clause> ; t => void

<write_clause> => void
where <write_list> ::= <clause> : writeable [:<clause> : int][,<write_list>]

output<clause> : file,<write_list> => void
where <write_list> ::= <clause> : writeable [:<clause> : int][,<write_list>]

out.byte<clause> : file,<clause> : int,<clause> : int => void

out.16<clause> : file,<clause> : int,<clause> : int => void

out.32<clause> : file,<clause> : int => void

<raster_op><clause> : image **onto** <clause> : #pixel => void

abort => void

t : nonvoid,<expression> : t => t

Expressions :

<expression> : bool **or** <expression> : bool => bool

<expression> : bool **and** <expression> : bool => bool

[~]<expression> : bool => bool

t : nonvoid, <expression> : t <eq.op> <expression> : t => bool

t : ordered, <expression> : t <compar_op> <expression> : t => bool

<expression> : pntr <type_op><identifier> => bool

t : arith, <expression> : t <add_op><expression> : t => t

t : arith, <add_op> <expression> : t => t

<expression> : int <int_mult_op> <expression> : int => int

<expression> : real <real_mult_op> <expression> : real => real

<expression> : string ++ <expression> : string => string

<expression> : pixel & <expression> : pixel => pixel

<expression> : pic <pic_op> <expression> : pic => pic

t : literal, <literal> : t => t

t : nonvoid, <value_constructor> : t => t

t : type, (<clause> : t) => t

begin end => void

{ } => void

t : type, **begin** <sequence> : t **end** => t

t : type, { <sequence> : t } => t

<expression> : string (<clause> : int <bar> <clause> : int) => string

<expression> : pixel (<clause> : int <bar> <clause> : int) => pixel

t : image,<expression> : t (<clause> : int <bar> <clause> : int) => t

t : nonvoid,<expression> : *[c]t (<clause> : int) => t

Value constructors:

t : nonvoid, **vector**<range>**of**<clause> : t => *t

t : nonvoid, @<clause> : int **of**<type1><lsb><clause> : t
[,<clause> : t]* <rsb> => *t
where <range> ::= <clause> : int :: <clause> : int[,<range>]

image <clause> : int **by**<clause> : int **of** <clause> : pixel => #pixel

t : image,**limit**<clause> : t [**to**<clause> : int**by**<clause> : int]
[**at**<clause> : int ,<clause> : int] => t

shift<clause> : pic **by**<clause> : real ,<clause> : real => pic

scale<clause> : pic **by**<clause> : real ,<clause> : real => pic

rotate<clause> : pic **by**<clause> : real => pic

colour<clause> : pic **in**<clause> : pixel => pic

text<clause> : pic **from**<clause> : real ,<clause> : real
to<clause> : real ,<clause> : real => pic

<lsb><clause> : real,<clause> : real<rsb> => pic

literals :

<digit>[<integer_literal>] => int

<integer_literal>.[<integer_literal>][e<scale_factor>] => real

true | **false** **=> bool**

<double_quote><char><double_quote> => string

on[&<pixel_literal>] | **off**[&<pixel_literal>] => pixel

nil => pntr

nullimage => image

nullfile => file

t : nonvoid, <standard_expression> : t => t

t : nonvoid, <standard_identifier> : t => t

Appendix III

Program Layout

Semi-Colons

As a lexical rule in S-algol, a semi-colon may be omitted whenever it is used as a separator and it coincides with a newline. This allows many of the annoying semi-colons in a program to be left out. However, to help the compiler deduce where the semi-colons should be, it is a rule that a line may not begin with a binary operator. For example

```
a *
b
```

is valid but

```
a
* b
```

is not.

This rule also applies to the invisible operator between a vector and its index list. For example

```
let b = a( 1,2 )
```

is valid but

```
let b = a
        ( 1,2 )
```

will be misinterpreted since vectors can be assigned. The same is true for strings, structures and images.

Comments

Comments may be placed in a program by using the symbol '!'. Anything between the ! and the end of the line is regarded by the compiler as a comment. For example

```
 a + b  ! add a and b
```