S - P O R T:

Definition of S-code V3.0

by

Peter Jensen, Simula a.s.
Stein Krogdahl, Simula a.s.
Øystein Myhre, Simula a.s.
Peter S. Robertson, Program Library Unit, Edinburgh
Gunnar Syrrist, Simula a.s.

This document is under supervision of the Standards Group for the Portable SIMULA System (SG/PSS). The statutes of the Standards Group are found in appendix C.

The document contains all decisions taken by the SG/PSS, whose last meeting was held in Edinburgh 1st and 2nd March 1983.

Simula a.s.,
21st June 1985

REMARK: This document is reassembled from the HTML version donated by Simula a.s.

Oslo 20. March 2020
Øystein Myhre Andersen

Preface to third edition

This report describes version 3 of S-code. Numerous changes of an editorial nature has been made, too many to mention. The language itself has to some extent been revised, due to the experience that we have had with the use of the language since September 1980. The most significant of these changes are

- The introduction of a macro facility in the S-code.
- Introduction of instructions for fetching values directly.
- A total revision of the chapter describing intermediate results, chapter 7, since the previous one contained serious errors.
- Removal of the instructions sconvert and endifc.
- Restrictions in connection with the segmentation control.
- A simplification of the routine definition.
- In case a profile is made visible, the parameter tags may also occur in the tag list (14.1).
- The concept of an interface module is introduced (14.2). In this module all global variables as well as the environment interface is collected.
- The following new symbols are added to the language:
    - fixrep (8), c-size (14), c-raddr (22), nobody (23), range (49), asscall (136), call-tos (137), dinitarea (138), nosize (139), popall (140), repcall (141), interface (142), macro (143), mark (144), mpar (145), endmacro (146), mcall (147), pushv (148), selectv (149), remotev (150), indexv (151), accessv (152) and the predefined tags RADDR (11) and SIZE (12).

The following S-code compilers are currently being implemented:

Computer	**Implementer**	**Under contract to**
VAX 11/780 VMS	PLU, Edinburgh	Simula a.s.
HB level 66	SZKI, Budapest	Honeywell-Bull, Paris
NORD 500	Univ. of Oslo	Norsk Data A/S, Oslo
Univac 1100	Simula a.s., Oslo	Simula a.s.
PR1ME 750	SIMPROG AB, Stockholm	PR1ME Computer, Stockholm
Motorola 68000	SIMPROG AB, Stockholm	SIMPROG AB, Stockholm
CP/M 8086	Simula a.s., Oslo	Simula a.s.
Mach-S	Simula a.s., Oslo	Simula a.s.

For information on the availability of these SIMULA implementations please contact the quoted parties or Simula a.s. Enquiries about the conditions for implementations on other computer systems should be directed to

Simula a.s.
Nedre Klekken vei 33
N-3514 Hønefoss, Norway

1. INTRODUCTION

In 1979 the Norwegian Computing Center (Oslo) and the Program Library Unit (Edinburgh) initiated the "S-PORT" project: implementation of a portable SIMULA-system. The system consists of three parts:

- a portable front-end compiler,
- a portable run time support system, and
- a machine dependent code generator.

The latter includes the interfaces to the compile and run time environment.

The present report defines the intermediate language "S-code" used to transmit an analyzed Simula program from the front end compiler to the code generator.

Both the front end compiler and the run time system will be distributed in S-code. In order to implement Simula it will be necessary to program the code generator (the S-compiler) for the machine in question. This compiler will take care of two tasks:

- it will produce code for the target machine corresponding to the S-program, and
- it will insert the necessary links to the operating system.

This report is intended as an implementation independent definition, therefore the code generation semantics are rather naive. A forthcoming report, "S-compiler implementation guide", will treat the implementation problems more in depth in view of our experience with the S-PORT systems under way on HB DPS8, VAX, PR1ME, NORD 500 and Univac 1100.

The S-code has been designed by the authors. We acknowledge valuable criticism and comment from Peter Håndlykken and Håvard Hegna (NCC), who painstakingly read through the draft of this report. Thanks are also due to Geoffrey E. Millard and Rob Pooley (PLU), Georg P. Philippot (University of Oslo), Andras Gaspar and the rest of the team at SZKI in Budapest, and Karel Babcicky and Birger Møller Pedersen (NCC) for contributions and ideas.

1.1. References

1. "Simula Common Base"
by O.-J. Dahl, B. Myhrhaug and K. Nygaard
Norwegian Computing Center, 1971.

2. "Simula Implementation Guide"
by O.-J. Dahl and B. Myhrhaug
Norwegian Computing Center, 1973

3. "S-compiler Implementation Guide"
Norwegian Computing Center (forthcoming)

4. "The Environment Interface"
by G. E. Millard, Ø. Myhre and G. Syrrist
Norwegian Computing Center, 1981.

5. "Simuletta Language Definition."
by Øystein Myhre.
Simula a.s. 1982.

6. "Run Time System Definition."
by Øystein Myhre.
Norwegian Computing Center, 1982.
(restricted).

7. "Code Generation Schemes for Portable SIMULA."
by Øystein Myhre and Birger Møller-Pedersen
Norwegian Computing Center, 1982.
(restricted).

1.2. Philosophy of the design

It is important to realise, that S-code is not seen as a set of instructions which can be executed (or interpreted) to perform the task specified by the source program. Instead S-code controls a compilation process, the outcome of which is an executable form of the program. Thus S-code comprises the instruction set of a compiler, not a (Simula) machine.

S-code has been designed as a practical tool to facilitate the production of Simula compilers on as wide a range of computers as possible. In addition the language makes it possible to generate efficient object code, where "efficient" can mean different things depending on the context: fast, compact, highly diagnostic, etc.

In order to achieve these aims the language had to be flexible enough to cater to these needs. Unfortunately this flexibility forces a certain vagueness in the definition of the S-code, making precise statements difficult and even dangerous as they may invalidate styles of implemen- tation for no better reason than that other possibilities were not foreseen. Consequently this definition is only precise in terms of those parts of the semantics which relate directly to the problems faced by the processor generating the S-code.

It is appreciated that the task of producing an S-compiler is in many ways much simpler than the task of maintaining such a compiler. The language has therefore been designed so that errors in both the S-program and the S-compiler can be detected as close to the point of error as possible.

To this end operations with far-reaching effects have been avoided as experience has shown them to be excellent at covering up processing errors. A guiding principle adopted is that it is better for the compiler to fail to compile a program (and inform the user of the fact) than to purport to have compiled it but in reality to have generated faulty code. In the former case there is no doubt as to where the problem lies (the compiler is wrong) whereas in the latter vast amounts of time can be spent trying to find non-existent bugs in user's programs.

In this report the language is presented using a conventional notation for the language symbols as if S-programs were created as symbolic strings. This notation has been chosen for publication purposes in order to make the report more readable. Actually all S-programs will be byte encoded, i.e. a program will always be a string of (8-bit) bytes, which encode the program according to the tables given in the appendix. No separating spaces etc. are necessary, and will thus never occur.

1.3. General overview

A system programmed in S-code will in the general case consist of a main program, which will receive control when the translated system is to begin execution, and several modules which may provide type definitions, routine support (in the form of a run time system, as is the case in S-PORT), etc.

Each program or module will contain three main classes of program elements:

- type definitions (record descriptors) govern the structure and manipulation of data quantities,
- data quantity declarations (const, local) control the actual (static) allocation and identification of data, and
- statements and instructions, possibly grouped in routines, specify (together with the type information) the target machine instructions to be generated.

Unlike most assembly languages, S-code contains a number of structured statements such as if-statement or call-statement. But unlike higher level languages (such as Simula) any intermixture of the above elements is permitted, as long as a very few rules are obeyed, the most significant being that

> Any identifier (tag) must have been explicitly "declared" before it may be used in an instruction.

Data to be manipulated may be of one of the usual types, such as integer, or the information may be structured as a record with named (tagged) components, or even into higher order structures such as linked lists or networks. Instructions (such as add) are included for the massage of primitive data, while pointer- and structure- manipulating instructions (such as select component) are included to support general graph traversion.

In principle S-code is a typed language, but type compatibility is enforced only upon the simple types; usually no checking is implied when manipulating the representation of instances of structured types, in particular no qualification check is implied on access to record components. It is considered the responsibility of the S-code generators (e.g. the front-end compiler) to ensure consistent use of pointers.

The main control structuring tool is the routine concept. A routine is inherently non-recursive, all parameter transmissions are by value (but it is possible to transfer pointer values), and strict syntax rules ensure that a routine will exit only through its final end. Unlike other languages, S-code permits explicit naming of the location, in which the return address is saved, so that a routine not necessarily returns to its point-of-call.

Routines also establish name hiding: all names (tags) defined within the routine are invisible from the outside, they lose their meaning when the routine is left. As a matter of fact they may be reused in other routines or even at a later point in the enclosing program, for quite different purposes.

A module defines a closed name scope (just like routines), it is however possible to selectively open the scope, making certain aspects of a module accessible outwith the module, while other aspects remain hidden. Type definitions, routine identifications, labels, and named constants can be made visible in this manner.

The interface module specifies the assumptions made about the environment, in which the translated program is to be executed. This special module also serves as a global area for variables and constants.

1.4. Terminology

Atomic unit
: A data storage unit. The size is the highest common factor of the sizes of all the data quantities which will be manipulated during the execution of a program. The size is implementation dependent. Atomic units may impose a finer resolution on the storage than the machine address allows.

Area
: A vector of one or more consecutive atomic units.

Object unit
: An area of implementation-defined fixed size; the size will always be an integral number of machine addressable storage cells. This is the allocation unit (storage cells may not be directly usable because of alignment problems).

Quantity
: Used with the meaning: something that (at run time) may be manipulated by the executing program.

Record
: An area with a structure imposed by a structured type defined by a record descriptor.

Object
: A record which is not a component of any record. An object will always comprise an integral number of object units.

Static quantity
: The quantity exists throughout the program execution.

Dynamic quantity
: The quantity is created during program execution.

Descriptor
: An abstraction used by the S-compiler to describe properties of quantities existing at run-time. The exact formats of the different kinds of descriptors are implementation dependent.

Complete descriptor
: A descriptor is said to be complete if the quantity described has been allocated, i.e. the actual address is known (otherwise it is incomplete).

Segment
: A contiguous storage area containing machine instructions.

Current program point
: The place which will contain the next target machine instruction generated is called the current program point.

Constant area
: A storage area used for the allocation of constants. Dependent upon the architecture of the target machine constants may be allocated interspersed with instructions (i.e. in program segments), in a separate storage area or elsewhere.

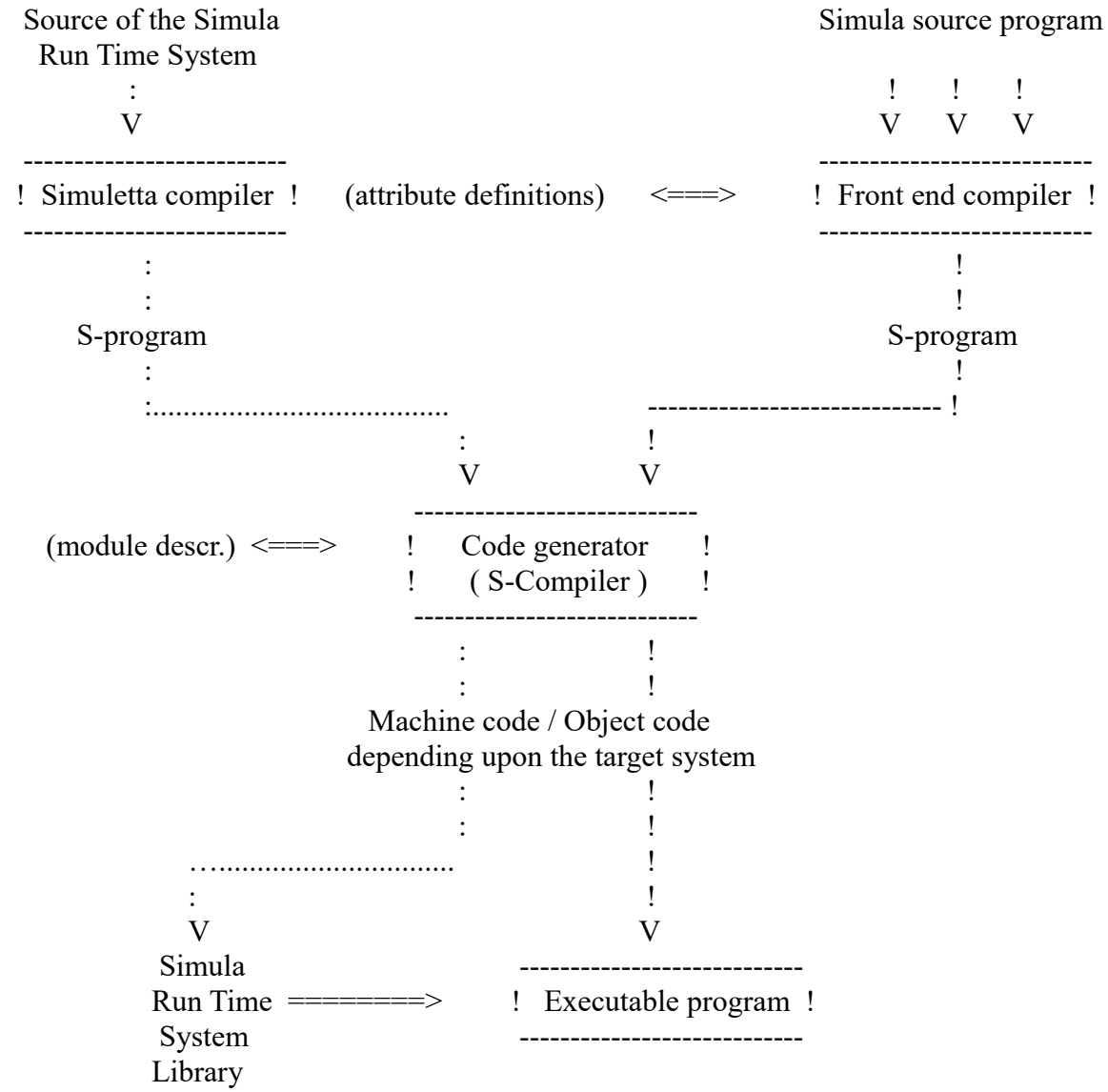
Stack
: A data structure in the S-compiler used in this report as an explanatory tool. See chapter 2.

S-program
: A byte string to be interpreted according to the syntax. Note that since all symbols are of fixed length (an integral number of bytes), no separators such as spaces or line shifts are necessary nor will they occur.

1.5. Description of syntax rules

The S-code syntax is described in BNF with the following additions:

- Meta symbols are written in lower case without brackets, terminal symbols are underlined lower case, and upper case is used for pre- defined tags.
- Alternative right hand sides for a production may be separated by ::= as well as !.
- Productions may be annotated with comments enclosed in parentheses.
- Part of a right hand side may be enclosed in angular brackets followed by one of the characters ?, *, or + with the following meanings:
    - < symbol string >? ("symbol string" is optional; it may occur zero times or once.)
    - < symbol string >* ("symbol string" may occur zero or more times at this point.)
    - < symbol string >+ ("symbol string" must occur one or more times at this point.)
- Spaces and line breaks are used simply to separate various parts of a production; they have no other significance, in particular they will not occur in the S-program.
- Particular instances of a meta-symbol may be given a prefix, separated from the symbol by a colon, e.g. body:tag. The prefix (body) has the sole purpose of identifying the meta-symbol (tag) in the accompanying description; it has no syntactical significance whatsoever.



General overview of an S-PORT system

2. DESCRIPTORS AND THE COMPILE TIME STACK

The actions, which the S-compiler should take upon recognition of a specific program element, are defined in terms of descriptors. These actions will in general involve both the manipulation of descriptors and code generation for the target machine. Exactly when such code generation should take place, as well as the generation order, is to a wide extent left to the S-compiler to decide. However, at certain points in the program code generation must take place. Such points are either implicit in the evaluation of certain instructions, or they will be marked explicitly by the eval instruction (see chapter 16).

The descriptors contain information about various objects which will exist during the computation. The descriptors themselves do not exist at run time; they are compile-time objects used to control the generation of executable code. The exact form of descriptors (and other data structures maintained during the compilation process) depends on both the target machine and the implementation techniques, but for the purpose of explanation we will assume that descriptors will contain at least three fields: MODE, TYPE, and "value".

Once a descriptor has been defined it remains unaltered, but the tagged descriptors are used to create and modify anonymous descriptors held in a stack. This stack need not have any existence in the compiled program, it is purely a device for describing the effect of the S-instructions. At run time intermediate results will exist; such "temporaries" are related to the state of the compile-time stack. In practice the stack will never be very deep, a limit of about fifty descriptors should prove to be over-generous.

2.1. The "value" field (BASE, OFFSET)

This field may hold an address or a value - or even a specification of a computation that will lead to either - depending upon the MODE field. For the purpose of explanation we will adopt the very naive view that this field always contains either an address description or a value. In the former case (when MODE is REF) we will consider this field to comprise two subfields, refered to as BASE and OFFSET; they will contain descriptors of the OADDR and the AADDR components of a general address respectively. Generally the "value" field will be described in very loose terms.

2.2. MODE, the access control field

The MODE is an implementation-dependent specification of an algorithm or "access rule" which, when applied to the rest of the descriptor, will yield the described quantity. Clearly, as these access rules will vary from compiler to compiler it is impossible to give a rigorous definition of them. However from the point of view of describing the effect of certain S-code constructions, access rules can be grouped into two general modes, namely VAL and REF.

VAL	VAL access rules describe values. In this case the "value" field may or may not contain the actual value described. A descriptor with MODE VAL could specify that a value is • the constant 137 • contained in register 4 • the address of label LAB1 • contained in area X • or it could specify an algorithm for obtaining the value, such as "add the value in area X to index register 6" etc.
REF	REF access rules describe references to areas which have an associated address of some sort. In this case the "value" field may or may not hold this address. A descriptor of MODE REF could specify that an area is • pointed at by index register 2 • pointed at by the area X • starting at bit 5 of the area 300 • at the address: displacement + (base) • at the address: (display(5)) + offset

The main significance of this distinction may be stated:

> An assignment of a quantity described by "descr1" to a quantity described by "descr2" is not valid if descr2 do not have mode REF. The descriptor descr1 may be of either mode, if it is REF, it is converted to VAL first.

2.3. TYPE, the data item type

TYPE specifies the values that may be associated with the object described, and hence also defines the interpretations of operations upon that object; the instruction add may operate on integers or reals, and depending on the actual type either a real (floating point) add or an integer add machine instruction may be generated.

Within a descriptor the TYPE field holds the tag of a descriptor defining the type. The simple types refer to pre-defined descriptors, while structured types refer to record descriptors. The descriptors refered to by TYPE have TYPE fields refering to themselves. The TYPEs manipulated by S-code may be grouped as follows:

VOID	This type is used to restrict the operations on certain descriptors. The only general operation permitted is pop, but other operations are defined on specific classes of descriptors of type VOID. VOID is not a type of S-code, purely a convenient specification for certain descriptors.
simple type	The simple types are BOOL, CHAR, INT, REAL, LREAL, SIZE, AADDR, OADDR, GADDR, PADDR, and RADDR, and are described in section 4.1.
structuredtypes	refer to type tags defined by record descriptors, see section 4.3.

2.4. Reference to descriptors (tags)

During the compilation descriptors will be created and destroyed. With the exception of stack items each descriptor is associated on creation with a unique identifying "tag", on destruction the tag becomes available for re-use.

In this document tags will be used in a manner similar to the use of identifiers in other languages, that is they will represent either a particular descriptor or the object described by that descriptor. In the hardware representation of S-code the tags will be positive numbers.

Syntactically a distinction is made between "tag", "newtag", and "spectag". Whenever "tag" occurs the corresponding value must be defined, that is associated with a descriptor. "newtag" signifies that the tag value is undefined, that is it has either never ocurred previously in the program text, or the meaning has been (explicitly or implicitly) deleted. In the case of "spectag", the tag value may either be undefined, otherwise it must have occurred previously in a specification of a routine, a constant, or a label.

2.5. Stack semantics

The effect of instructions upon the items in the stack is described using a procedural notation, and marked with asterisks in the left margin. The following terms are used with a specific meaning:

- the term "error" means that the compilation should be abandoned with a minimum of change to the controlling environment. A suitable error message should be generated.
- "check ..." (e.g. check TOS ref;) means that if the condition is not fulfilled (in the example: if TOS is not mode REF) the S-program is in error (see above).
- The top elements of the stack will be refered to as TOS (top of stack) and SOS (second on stack). If any instruction refers to a non-existing item on the stack, the S-program is in error.
- TOS and SOS (underlined) refer to the elements before the evaluation started.
- the operation "++" implies the concatenation of two OFFSETs, the result (at run time) being the OFFSET of the second operand relative to the BASE of the first operand. This may imply code generation. For an example, see the index instruction in chapter 6.
- "force ... value" means that a descriptor of mode REF is changed to describe the contents of the referenced area, thereby becoming VAL. This may imply code generation. If the mode was VAL, nothing is done.

3. GENERAL STRUCTURE OF S-PROGRAMS

```
S-program
        ::= program program_head:string
            program_body endprogram

program_body
        ::= interface_module
        ::= macro_definition_module
        ::= <module_definition>*
        ::= main <local_quantity>* <program_element>*

program_element
        ::= instruction
        ::= label_declaration
        ::= routine_profile          ! routine_definition
        ::= skip_statement           ! if_statement
        ::= protect_statement
        ::= goto_statement           ! insert_statement
        ::= delete_statement
instruction
        ::= constant_declaration
        ::= record_descriptor        ! routine_specification
        ::= stack_instruction        ! assign_instruction
        ::= addressing_instruction   ! protect_instruction
        ::= temp_control             ! access_instruction
        ::= arithmetic_instruction   ! convert_instruction
        ::= jump_instruction
        ::= goto_instruction
        ::= if_instruction           ! skip_instruction
        ::= segment_instruction      ! call_instruction
        ::= area_initialisation      ! eval_instruction
        ::= info_setting             ! macro_call
```

The program head will contain the identification of the front-end compiler and other information; the exact format is defined in (4). Program elements are not permitted within routine bodies, while instructions are (see section 13.3).

The distinction between program elements and instructions is syntactically ambiguous in the case of if-, skip-, protect- and segment-constructions; we emphasize that the syntax given is intended to be descriptive rather than the base of an automatic compiler generation scheme.

Any local quantities and descriptors are visible only from within the body of the main program; all such tags are destroyed at endprogram and made available for re-use. They are statically allocated and part of the main program throughout an execution, by analogy with Algol's 'own'-variables. Their initial values are undefined.

3.1. The primitive syntax symbols

byte

::= an (8-bit) unsigned integer value in the range 0..255.

number

::= a two-byte value greater than or equal to zero. A number (or any other multi-byte structure) will always be transmitted with the most significant byte first. Let the bytes be <B1><B2> in that sequence; the value will be 256 * B1 + B2 .

ordinal

::= a number with value greater than zero.

tag

::= An ordinal (the "tag-value") associated with a descriptor. See section 2.4.

::= The number zero followed by an ordinal (the "tag value") and an identifying string.

The second form is intended for debugging purposes and is used to associate an identification with the tag.

newtag

::= A tag-value with no association, see section 2.4.

spectag

::= a tag-value which, if not undefined, must have been given an association in a specification (labelspec, constspec, routinespec), see section 2.4.

index

::= a byte within an implementation-defined range which identifies an internal label. An important side effect of its occurrence is that it looses its meaning. Thus an internal label can be the destination of exactly one jump-instruction. See section 11.2.

newindex

::= an index-byte which is undefined; it becomes defined as an internal label through its occurrence. See section 11.2.

string

::= a byte with the value N followed by N "data bytes". The character count N must be greater than zero, thus a string cannot be empty.

4. TYPES AND VALUES

type

::= structured_type ! simple_type

resolved_type

::= resolved_structure

::= simple_type

::= INT Range lower:number upper:number

::= SINT

Any data quantity must belong to some type. The type will define the internal structure of the quantity as well as the operations that may be performed upon it. Types are used as generators in global, constant, local or parameter definitions and as specificators in quantity descriptors. Each type defines a descriptor (of the same type), this descriptor cannot be used on the stack, thus types cannot be used dynamically as e.g. parameters.

The distinction between resolved and non-resolved type is made because of the indefinite repetition, which may occur in structured types. Such a type cannot be used as a generator, or in further type definition, without determining the actual number of elements in the repetition.

Whenever the S-compiler should perform type checking, neither the actual number of elements in such an indefinite repetition, nor the actual range specified for an INT quantity is of any significance, unless it is explicitly indicated in the text. A structured type can be extended by using prefixing. Such two types are not type compatible (but note that the S-code only imposes type checking in a very few cases such as in initialisation of globals).

4.1. Predefined types and their value ranges

simple_type

::= BOOL ! CHAR

::= INT ! REAL ! LREAL ! SIZE

::= OADDR ! AADDR ! GADDR ! PADDR ! RADDR

All simple types are predefined tags. The language offers no possibility to define new simple types.

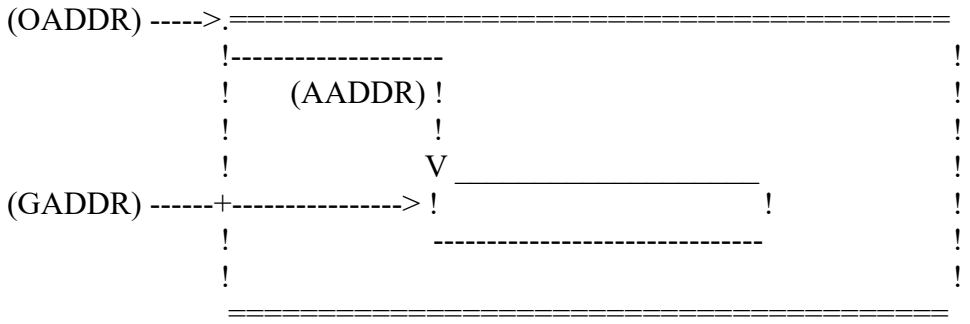
4.1.1. Arithmetic types

BOOL	Boolean. Takes the values true or false.
CHAR	Character. 256 different characters are defined (corresponding to an 8-bit representation). The lower half of this ordered set is the ISO 646 character set (encoded accordingly), the interpretation of the upper half is implementation-dependent.
INT	Integer. Signed integer values with a machine dependent range and representation. The range must correspond at least to a 16-bit representation, but if possible it should be at least 32 bits.
REAL	A subset of the real values, which is representable on the target machine.
LREAL	Long real. A set of real values with greater precision than the REAL values. The range of LREAL may differ from that of REAL. Restricted implementations may choose to ignore the distinction between REAL and LREAL, treating both as REAL.
SIZE	Object size. Values of this type describe object sizes and distances between objects, measured as the distance between two machine addresses. They may be represented as integers (with sign), but they are not type compatible with INT. The empty size (corresponding to zero) is nosize.

4..1..2. Data address types

Addresses in S-code are designed to permit the description of objects which are arranged in implementation and machine-dependent ways. In particular it is envisaged that some S-compilers may pack information into available storage in ways which require to be described using complex addresses. These considerations have led to the following types of address:

OADDR	Object address. Specifies the address of an object unit, in general the first unit in an object. onone designates no object. All other values of type OADDR will correspond to true machine addresses.
AADDR	Attribute address. Specifies the relative address of (the first atomic unit of) a record component. This is of course not a true address in the machine sense of the word, the component will not be accessible without the OADDR base. anone designates no attribute.
GADDR	General address. A pair (OADDR,AADDR) which uniquely identifies a particular atomic unit. The components of a GADDR value are not directly accessible, but they may be extracted by means of convert instructions. gnone designates no unit and is the pair (onone, anone). Values of this type will not necessarily correspond to pure machine addresses.



Correspondence between the different data addresses

4.1.3. Instruction address types

PADDR	Program address. Specifies the address of an instruction. It is independent of the other types of address. nowhere designates no program point.
RADDR	Routine address. Specifies the entry point address of a routine body. It is independent of (and inconvertible to) the other address types. nobody designates no entry point.

4.2. Quantity descriptors

quantity_descriptor

::= resolved_type < Rep count:number >?

Any quantity in the S-program is described by a quantity descriptor. The type will define the internal structure of the quantity as well as the set of operations permitted.

4.2.1. Subrange of INT (range, SINT)

Quantities of type INT may be restricted in range, as shown. A range has a contiguous value domain which is part of the domain for INT.

The motivation for range is storage economy for variables with a restricted value domain. Ranges are used extensively in the S-port run-time system, e.g. for some of the system attributes present in all user-defined objects.

The S-compiler allocates a range to a storage unit which at least comprises the closed interval (lower..upper) specified for the range. The value domain of a range is system dependant and defined to be the domain of the storage unit allocated to the range.

The type SINT corresponds to the type of variables declared short integer in a Simula program. The S-compiler treats SINT as a range specified with a suitable interval, at least including (-32000,32000), and (almost) symmetrical around zero.

Although ranges occur in arithmetic S-code expressions, there is no arithmetic operations (or relations) defined on ranges. It is the responsibility of the S-compiler to convert to INT before the operation is performed, and to convert to range before assignment to a range. (This scheme is consistent with the Common Base specification of short integer which states that all operations on short integer should be performed in full integer arithmetic.)

One consequence of this is that intermediate results (see chapter 7) never can be range restricted.

It is required that the S-compiler checks domain overflow for assignment to ranges originating from SINTs, while such a check is not required for other ranges, e.g. ranges declared in the run-time system. The S-compiler may choose to always generate domain checks on assignment, hence treating any range as it must treat SINT.

The predefined range SINT is the only possible tagged quantity descriptor.

4.2.2. Repetitions (rep)

A quantity may be defined as a repetition. If the count:number is greater than zero a vector of identical elements is defined, containing that number of elements and accessed through indexing (see index, chapter 6) with indices starting at zero. The allocation of the elements must be done in such a way that the size of each element provides enough information to permit access to one relative to another. A count of zero indicates that the number of elements is indefinite. Such indefinite repetitions are only permitted in record descriptors.

It is important to realise that the repetition concept does not impose any structure upon the quantity (as e.g. an array declaration does). Whenever a repeated quantity is selected (by push, select, or remote) the first element is selected directly.

4.3. Type definition

```
record_descriptor
        ::= record record_tag:newtag <record_info>?
                <prefix_part>? common_part
                <alternate_part>* endrecord
record_info
        ::= info "TYPE"   !   info "DYNAMIC"

prefix_part
        ::= prefix resolved_structure

common_part
        ::= <attribute_definition>*

alternate_part
        ::= alt <attribute_definition>*

attribute_definition
        ::= attr attr:newtag quantity_descriptor

resolved_structure
        ::= structured_type < fixrep count:ordinal >?

structured_type
        ::= record_tag:tag
```

A record descriptor defines a structured type with the tag 'record tag'. This new type may then later be used either as a generator in variable or constant definitions, as a prefix or as an attribute type in further record descriptors; in all these cases an indefinite repetition must be resolved through the use of fixrep. Or the record tag may be used as a parameter to some instructions, in which case an indefinite repetition need not be so resolved.

For debugging purposes the newtags defined in a record descriptor may be associated with a line number and an identification string. In that case the keywords record, resp. attr, are replaced by the rec-id (attr-id) construction as shown. The line:number and id:string may safely be ignored by the S-Compiler.

A set of incomplete type descriptors will be associated with the record tag:newtag. These descriptors may be divided into several subsets: one describing part of the type with a fixed interpretation and the remainder describing a number of interpretations for the rest of the type. The SIZE of the structure will be the SIZE of the prefix plus the SIZE of the common part plus the SIZE of the "largest" alternative, that is the alternative occupying the largest number of object units (disregarding a possible indefinite repetition). Which of several alternatives determines the SIZE is implementation dependent.

4.3.1. Prefixing

If specified, the prefix must refer to a defined type descriptor, resulting in a type which is the prefix type extended by the specified attributes (an indefinite repetition in the prefix type must be resolved).

4.3.2. Attributes

Each attribute definition defines the type of an accessible attribute of the structure. There is no inherent correspondence between the order of the attributes in the record descriptor and the allocation order inside a record, but when the record descriptor is processed by the S-compiler, an AADDR value will be associated with each attribute tag defined. These associations cannot later be changed, which implies that the order prefix...common part must be preserved.

4.3.3. Indefinite repetitions and fixrep

An indefinite repetition in the record descriptor must be the (lexically) very last attribute defined, i.e. it occurs immediately before endrecord. The type so defined cannot generally be used (except in a few instructions) without being resolved through the application of the fixrep construction; this will determine the actual number of elements in the repetition. Note however that type compatibility checking is connected to the record tag, so that structures are of the same type even if they are resolved with different counts.

4.3.4. Alternatives (records with variants)

If several alternate parts are given, it specifies alternative interpretations of the same area, the correspondence between pairs of attributes from different alternatives is not defined. The alternate part(s) must be allocated following the common part.

4.3.5. Allocation order

The S-compiler is free to reorder or pack the attributes in any convenient way, as long as the above mentioned restrictions are observed, in summary:

- the order prefix...common part...alternatives must be preserved,
- once ordered and packed the order is invariant, that is a prefix cannot be re-packed,
- an indefinite repetition must be allocated at the end of the structure,
- records are always allocated with a size corresponding to the largest alternative (at allocation an indefinite repetition cannot occur).

Examples:

```
record COMPLEX
        attr RE REAL attr IM REAL
endrecord

record C
        attr D BOOL
        attr E COMPLEX rep 2
endrecord

record F prefix C
        alt attr G REAL
        alt attr H PADDR
        alt attr J BOOL rep 0
endrecord
```

The type F will have the common part as defined by C, and three alternative parts; one contains an indefinite repetition. The indefinite repetition 'J' must be resolved whenever F is used either in a record descriptor or as a generator for allocation.

4.3.6. Record information

The record info string is used to give information to the S-Compiler on the use of certain records. Three classes of records are distinguished, any record descriptor will be in one of these classes:

- - All structured types which may occur as TYPE in value mode stack descriptors shall contain the record info "TYPE". This may be usefull when processing the protect construction.
- - A small set of structured types are used as prefixes to every dynamic quantity created by the run time system during execution; such types shall contain info "DYNAMIC". This information may be necessary in order to determine the size, since the target system may prohibit general use of the address space for such objects, e.g. dynamic reference should be to an even byte address).
  dsize is valid only for types in this set.
- - If a record descriptor does not contain (either directly or through a prefix) any such specification, the record will not be used for any of the two above mentioned purposes.
  It may not be necessary for the S-compiler to utilise the record information field; in that case the specifications may be ignored and no distinction should be made by the compiler.

4.4. Constant values

value
::= boolean_value ! character_value
::= integer_value ! size_value
::= real_value ! longreal_value
::= attribute_address ! object_address
::= general_address ! program_address
::= routine_address ! record_value

repetition_value
::= <boolean_value>+
::= <character_value>+ ! text_value
::= <integer_value>+ ! <size_value>+
::= <real_value>+ ! <longreal_value>+
::= <attribute_address>+ ! <object_address>+
::= <general_address>+ ! <program_address>+
::= <routine_address>+ ! <record_value>+

4.4.1. Arithmetic values

text_value
::= text long_string

long_string
::= (an ordinal with value N followed by N "data bytes".
Note that a text value cannot be empty.)

A text value is understood as a repetition of characters, e.g.
text "abcd" == c-char 'a' c-char 'b' c-char 'c' c-char 'd'.

boolean_value
::= true ! false

character_value
::= c-char byte

integer_value
::= c-int integer_literal:string

An integer literal is a string:

< <radix> R>? <sign>? <digit>+

where digit is one of the (ISO coded) decimal digits, and sign may be + or -. The letter R, if included, signals that the integer is specified with a radix preceding R. The only legal radices are 2, 4, 8 and 16. If the radix is 16, <digit> may also be one of the (ISO-coded) letters A-F, with the obvious meaning.

real_value
::= c-real real_literal:string

longreal_value
::= c-lreal real_literal:string

A real literal is a string:

<sign>? <digit>* < . <digit>+ >? < & <sign>? <digit>+ >?

Note that neither an integer literal nor a real literal may contain spaces.

size_value
::= c-size type ! nosize

If the type contains an indefinite repetition the size is measured as if this attribute is absent, i.e. only the part(s) of the type preceding the indefinite repetition is measured.

The size of the type is measured as the distance (see dist chapter 6) from the first object unit allocated to a record of the type to the first object unit following the record, i.e.

size = dist(first,next) .

4.4.2. Data address values

attribute_address
::= < c-dot attribute:tag >* c-aaddr attribute:tag
::= anone

The value of an attribute address is the OFFSET of the attribute. This may be computed relative to any surrounding record by means of the c-dot construction. anone is an empty OFFSET, refering to no attribute. The interpretation of the construction

c-dot T1 c-dot T2 ... c-aaddr LT is "T1.T2.. .LT".AADDR

object_address
::= c-oaddr global_or_const:tag
::= onone

The value is the object address of the global or constant quantity given. onone refers to no object unit.

general_address
::= < c-dot attr:tag >* c-gaddr global_or_const:tag
::= gnone

The value is the general address of the defined global or constant quantity designated by the lexically first tag. The general address of a sub-component of a structure component may be given by means of the c-dot construction; this is interpreted as for attribute addresses. gnone is the address of no atomic unit. The interpretation of the construction

c-dot T1 c-dot T2 c-gaddr T3 is "T3.T2.T1".GADDR

4.4.3. Instruction address values

program_address
::= c-paddr label:tag ! nowhere

The value is the program point designated by the label. nowhere designates no program point.

routine_address
::= c-raddr body:tag ! nobody

The value is (the entry point of) the routine specified; a peculiar routine cannot occur. nobody designates no routine body.

4.4.4. Structured values

record_value
::= c-record structured_type
<attribute_value>+ endrecord

attribute_value
::= attr attribute:tag type repetition_value

Strict type correspondence is required between an attribute, the given type and the repetition value. The attribute:tag defines which attribute is to be of a given value, i.e. the sequence need not be the same in the structured type and in the record value. With the exception of alternatives (see below), all attribute tags of the record descriptor must occur exactly once, otherwise: error.

An indefinite repetition is resolved by the number of values in the corresponding attribute repetiton value.

If the type contains alternative parts, a specific alternative is selected by naming one of its attributes. Once an alternative has been so selected, the S-program is in error if attributes from any other alternative occur, or if any attribute from this alternative is missing. If no alternative is selected then no alternative part is produced.

Example: (cf. 4.3)

```
c-record F attr E COMPLEX
        c-record COMPLEX
                attr IM REAL c-real "0.0"
                attr RE REAL c-real "0.0"
        endrecord (first element of repetition E)
        c-record COMPLEX
                attr RE REAL c-real "1.0"
                attr IM REAL c-real "-1.0"
        endrecord (second element of E)
        attr H PADDR c-paddr LABEL27
        attr D BOOL true
endrecord
```

Note that the attributes may be given in any order with the exception that the repetition elements must be given in sequence. Note also the specific selection of an alternative by naming H.

4.5. Tagged constants

constant_declaration
::= constant_specification ! constant_definition

constant_specification
::= constspec const:newtag quantity_descriptor

constant_definition
::= const const:spectag
quantity_descriptor repetition_value

A constant area is created to hold the specified value. A descriptor (identified by the given tag) will represent this area, i.e. the descriptor will be of the general form (REF, constant.TYPE, ...). In case it is necessary to refer to a tagged constant before it is possible to give its value, the constant specification is used to bind the tag to a value of the given type. This constant must be given a value later in the program through a constant definition, and the types must be the same.

A constant is always allocated an integral number of object units.

Examples:

```
constspec PI REAL
.
.
const PI REAL c-real "+3.1416"

const CZERO COMPLEX
        c-record COMPLEX
                attr RE REAL c-real "0.0"
                attr IM REAL c-real "0.0"
        endrecord
```

5. STACK INSTRUCTIONS AND ASSIGNMENT

stack_instruction
::= push obj:tag ! pushv obj:tag
::= pushc value ! dup ! pop ! empty
::= pushlen (see chapter 7)
::= popall byte

assign_instruction
::= assign ! update ! Rupdate

push obj:tag

* push(REF, obj.TYPE, obj.BASE, obj.OFFSET);

Obj must refer to a defined global, local or constant quantity, otherwise: error.
Observe that routine parameters and exit tags are local quantities.

A copy of the descriptor is pushed onto the stack.

pushv obj:tag

* push(REF, obj.TYPE, obj.BASE, obj.OFFSET);
* force TOS value;

This instruction has the same effect as a push obj:tag, followed by a fetch.

pushc constant:value

* push(VAL, constant.TYPE, "value");

A descriptor of the given value is pushed onto the stack.

Note that on some machines it is possible to use certain values as part of the target instruction (immediate operands), in which case the descriptor may hold the actual value. On other machines or with more complex values, it may be necessary to place the value in store. In either case the descriptor will be of mode VAL, thus the value will not have an associated address. The syntax of values is given in section 4.4.

dup

* push(TOS);
* force TOS value;

A duplicate of TOS is pushed onto the stack and forced into value mode.

pop

* pop;

Pop off TOS;
This instruction is illegal if TOS is a profile description.

popall N:byte

* perform pop n times;
* check stack empty;

Pop N items off the stack. The stack should then be empty, otherwise: error.

This instruction gives a short way of emptying the stack, together with the control of the number of elements that was on the stack. Profiles cannot be deleted from the stack by pop, only by deleting the complete stack through popall.

empty

* check stack empty;

This instruction is intended as a debugging aid, it is recommended that the condition is checked always.

assign (dyadic)

* force TOS value;
* check SOS ref; check types identical;
* pop; pop;

Code is generated to transfer the value described by TOS to the location designated by SOS. This implies that the stack elements must be evaluated, and that any code generation involving TOS or SOS, that has been deferred for optimisation purposes, must take place before the assignment code is generated. SOS and TOS are popped from the stack.

update (dyadic)

* force TOS value;
* check SOS ref; check types identical;
* force SOS value;
* pop;

Code is generated to transfer the value described by TOS to the location designated by SOS. TOS must be evaluated and any deferred code generation involving TOS must take place before the update code is generated. Note that only TOS is popped and the new TOS is modified to describe the value assigned.

rupdate (dyadic)

* check TOS ref;
* force SOS value; check types identical;
* pop;

This instruction ("reverse update") works almost like update with the sole exception that the roles of TOS and SOS are interchanged, i.e. the value transfer is from SOS to TOS.

6. ADDRESSING INSTRUCTIONS

addressing_instruction

::= fetch ! refer resolved_type ! deref

::= select attribute:tag ! selectv attribute:tag

::= remote attribute:tag ! remotev attribute:tag

::= index ! indexv

::= inco ! deco

::= dist ! dsize structured_type

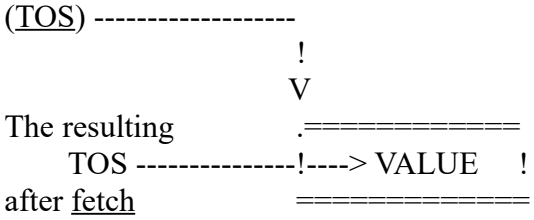
::= locate

The effect of the addressing instructions upon the stack will be illustrated by abstract diagrams. Note that a mode REF stack item is depicted as if it actually pointed into storage; this is of course not true during the compilation, where the address in most cases will be unknown.

fetch

* force TOS value;

TOS.MODE should be REF, otherwise fetch has no effect.
TOS is modified to describe the contents of the area previously described.



refer resolved_type	
*	force TOS value; check TOS type(GADDR);
*	TOS.MODE := REF; TOS.TYPE := type;
	TOS is modified to describe a quantity of the given type, at the address described by TOS.

(TOS) > GADDR VALUE

The resulting V

TOS >

after refer REF object

of

"type"

deref	
*	check TOS ref;
*	TOS.MODE := VAL; TOS.TYPE := GADDR;
	TOS is modified to describe the address of the area.

(TOS)

REF

The resulting V

TOS > GADDR VALUE >

after deref

select attr:tag	
*	check TOS ref;
*	TOS.TYPE := attr.TYPE;
*	"TOS.OFFSET := TOS.OFFSET ++ attr.OFFSET";
	(note that the BASE component of TOS is unchanged)

	The area described by TOS is considered to be holding a record of the type, say 'REC', in which the instruction argument attr is an attribute. TOS is modified to describe the designated component of that record. Note that no qualification check is implied, i.e. TOS.TYPE may be different from 'REC'.

BASE >

TOS.OFFSET

REF V

(TOS) >

attr.OFFSET

The resulting REF V

TOS >

after select attr

selectv attr:tag	
*	check TOS ref;
*	TOS.TYPE := attr.TYPE;
*	"TOS.OFFSET := TOS.OFFSET ++ attr.OFFSET";
*	force TOS value;
	The instruction has the same effect as a select attr:tag followed by a fetch.
remote attr:tag	
*	force TOS value; check TOS type(OADDR);
*	pop;
*	push(REF, attr.TYPE, "BASE = value(TOS), OFFSET = attr.OFFSET");
	This instruction uses one step of indirection. The value is considered to be the address of an object of the type 'REC' in which attr is an attribute. TOS is replaced by a descriptor of the designated component of that object. Note again that no qualification check is implied (neither could it be done).

(TOS) > OADDR VALUE >

attribute

OFFSET

The resulting REF V

TOS >

after remote attr

remotev attr:tag	
*	force TOS value; check TOS type(OADDR);
*	pop;
*	push(REF, attr.TYPE, "BASE = value(TOS), OFFSET = attr.OFFSET");
*	force TOS value;
	The instruction has the same effect as a remote attr:tag followed by a fetch.

index (dyadic)	
*	force TOS value; check TOS type(INT);
*	check SOS ref;
*	pop;
*	TOS.OFFSET := SOS.OFFSET ++ "SOS.SIZE * value(TOS)"
	SOS is considered to describe an element of a repetition, and the purpose of the instruction is to select one of the components of the repetition by indexing relative to the current position. The effect may perhaps best be understood by considering an infinite array A with elements of SOS.TYPE. The array is placed so that element A(0) is the quantity described by SOS. After index the stack top will describe A(N), where N is the value of TOS. No bounds checking should be performed.

indexv	
*	force TOS value; check TOS type(INT);
*	check SOS ref;
*	pop;

*	TOS.OFFSET := SOS.OFFSET ++ "SOS.SIZE * value(TOS)"
*	force TOS value;
	This instruction has the same effect as an index followed by a fetch.

inco, deco (dyadic)	
*	force TOS value; check TOS type(SIZE);
*	force SOS value; check SOS type(OADDR);
*	pop; pop;
*	push(VAL, OADDR, "value(SOS) +/- value(TOS)");
	The two top elements are replaced by a descriptor of the object address RESULT defined through the equation dist(RESULT,value(SOS)) = +/- value(TOS) where + corresponds to inco and - to deco.

indexv	
*	force TOS value; check TOS type(INT);
*	check SOS ref;
*	pop;
*	TOS.OFFSET := SOS.OFFSET ++ "SOS.SIZE * value(TOS)"
*	force TOS value;
	This instruction has the same effect as an index followed by a fetch.

dist (dyadic)	
*	force TOS value; check TOS type(OADDR);
*	force SOS value; check SOS type(OADDR);

*	pop; pop;
*	push(VAL, SIZE, "value(SOS) - value(TOS)");
	TOS and SOS are replaced by a description of the signed distance from TOS to SOS.

dsize structured_type	
*	force TOS value; check TOS type(INT);
*	pop;
*	push(VAL, SIZE, "size(type with mod. rep.count)");
	The structured type must be prefixed with a "DYNAMIC" type (see 4.3.6), and it must contain an indefinite repetition, otherwise: error. This instruction is a dynamic version of the c-size value function (see section 4.4.1); it may be used to compute the actual size of an object of a type containing an indefinite repetition. TOS describes a value, the actual repetition count to be used in the size calculation, i.e. the size is computed as if the type had been defined with this repetion count instead of zero. A description of the size thus obtained replaces TOS. Observe that if the repetition occurs as the last of several alternatives, the repeated alternative must be used to determine the size. This may give a smaller size than if c-size of the type was requested.

locate (dyadic)	
*	force TOS value; check TOS type(AADDR);
*	force SOS value; check SOS type(OADDR,GADDR);
*	pop; pop;
*	push(VAL, GADDR, "value(SOS).BASE, value(SOS).OFFSET++value(TOS)");
	SOS and TOS are replaced by a description of the general address value formed by "addition" of the two original addresses.

(SOS) >

(TOS)

The resulting V

TOS > : :

after locate

7. INTERMEDIATE RESULTS

protect_statement

::= save <program_element>* restore

protect_instruction

::= save <instruction>* restore

temp_control

::= t-inito ! t-geto ! t-seto

During the compilation the stack will regularly contain items, which describe partially evaluated expressions such as e.g. incomplete address calculations. The execution of the corresponding machine instructions will at run time give rise to intermediate results; of necessity these must be held in some form of anonymous storage, the temporary area. The actual implementation of this area should be highly target machine dependent, thus the machine registers may be used if a sufficiently large number of registers are available.

At save the intermediate results are copied from the temporary area into some object, the save-object, provided by the run time system, and at restore the temporary area is reestablished from this object.A garbage collection might be invoked from a routine (but never from the outermost program level), thus the values of type OADDR and GADDR in the save-objects must be accessible (to the run time system). (We will in this chapter call the OADDR part of such values pointers.)

For the purpose of explanation we assume that the S-compiler keeps track of two attributes of the temporary area, namely ALLOCATED which is the SIZE of the used part of the temporary area at any given program point, and MARKS which identify all intermediate values of type OADDR or GADDR. The former must be available during the processing of pushlen (q.v.), while some representation of the latter must be transferred to the save-object (at save).

The temp control instructions are used as follows: t-inito is called in preparation of a complete scan through the pointers of a save-object. During the scan t-geto will be called repeatedly, yielding the pointers successively. If the pointer is to be updated, one t-seto will follow the corresponding t-geto, so that the pointer just inspected will be updated.

7.1. Implementation directives

Since the contents of a save-object always will be accessed through one of the instructions to be defined below (t-inito, t-geto, t-seto) the internal structure is completely implementation- dependent. The reason is, that the access must be as fast as possible on any given architecture, since the garbage collector makes several scans over the existing save-objects during the mark and the move phases.

7.1.1. The save object

Apart from the actual temporaries saved, some additional information must be present in the save-objects. For the purpose of explanation we will call this additional attribute SAVE-MARKS. SAVE-MARKS is some representation of the MARKS attribute to the temporary area, which allows for sequential access to all pointer values saved. (E.g. a bit map indicating the positions of the pointers in the save object, or an address to a description of the save object, or a value giving the number of pointers, provided they are stored consecutively in the save object, from a fixed offset.)

All of the temporary area must be included in the save object, except those values which are constants. This includes non-pointers as well as pointers, and includes REF items as well as VAL items. All REF items, all GADDR items and all OADDR items must be delivered as pointers with t-geto unless they are known to be onone or gnone, or are known to point within global variables, local variables or constants. Pointers that are attributes of records must be returned unless they are in an alternate part of the record. Note that the requirements that REF pointers must be returned implies that in general the run time representation of a REF item must be the same as that of a GADDR.

7.1.2. Temp-control

The parameters to the instructions t-geto and t-seto are implicit, i.e. they refer to the save-object referenced by the most recent call (at run time) on t-inito, successive calls on t-geto scans through the pointers in the save object, and a call on t-seto refers to the pointer accessed by the most recent call on t-geto.

For the purpose of explanation we introduce two anonymous variables, SAVE-OBJECT and SAVE-INDEX. SAVE-OBJECT is set by t-inito and referenced by t-geto and t-seto. SAVE-INDEX is initialized by t-inito, updated by t-geto and referenced by t-seto. In an implementation some representation of SAVE-OBJECT and SAVE-INDEX could be kept in dedicated registers or in main storage. The use of the variables is explained in detail below.

7.2. The occurrence of the protect construction

The construction will occur in S-programs in the following context alone:

pushlen	(parameter to a run time)
asscall "profile:tag"	(system routine, which returns)
call "body:tag"	(the OADDR of a save-object)
save	

...	(some S-code sequence which will result in TOS describing an OADDR (mode REF), pointing to the object saved above)
restore	

The protect instruction is illegal within routine bodies.

The pushlen instruction will occur only in this context. Furthermore the object address returned from the run time system before restore will point to the object generated at the corresponding save. Observe that the object address is not necessarily the same since a garbage collection may have occurred between save and restore.

pushlen	
*	push(VAL, SIZE, "temporary area.LENGTH");
	An implicit eval is performed. The SIZE needed for the following save, that is the sum of the current value of ALLOCATED and the number of object units, which is needed for SAVE-MARKS and possibly other implementation-dependant information, is computed and the value is pushed onto the stack. For optimisation purposes, it is set to nosize in case ALLOCATED = nosize (i.e. if the temporary area is empty). In this case the accompaning save and corresponding restore will receive onone as parameter. An S-compiler may choose to skip code generation for the complete sequence pushlen, asscall, call, and save in the case ALLOCATED = nosize. In that case the processing of restore is changed, see below.

save	
*	force TOS value; check TOS type(OADDR);
*	pop;
*	remember stack;
*	purge stack;
	TOS describes the address of a save-object. The size of this object is as determined by the preceding pushlen. The complete state of the stack is remembered (together with the values of

	ALLOCATED and MARKS) and the compilation continues with an empty stack. Code is generated, which - if TOS.VALUE <> onone (see note below) - at run time will save the used part of the temporary area, and set the SAVE-MARKS attribute. TOS is popped.

restore	
*	check TOS ref; check TOS type(OADDR);
*	push(onone); perform assign;
*	check stack empty;
*	reestablish stack remembered at corresponding save;
	The stack remembered by the corresponding save is reestablished (together with the attributes ALLOCATED and MARKS). Code is generated, which - if TOS.VALUE <> onone (see note below) - at run time will copy the content of the specified save-object into the temporary area (the save-object will be the one generated at the corresponding save). After the copy has been taken, onone is assigned to what is referred by TOS. TOS is popped.

Notes:

- Although the test TOS.VALUE = onone is a run time condition it may be deduced by the S-compiler from the fact that onone will occur on the stack iff pushlen resulted in nosize being pushed (and the latter condition is determinable at compile time).
- The front end compilers do not generate the protect construction in case the compile time stack is empty.

t-inito	
*	force TOS value; check TOS type(OADDR);
*	pop;
	Code is generated to initialise a scan of the save-object described by TOS, i.e. SAVE-OBJECT is

	set to refer to the object, and SAVE-INDEX is initialized. TOS is popped.

t-geto	
*	push(VAL, OADDR, "value of current pointer");
	Code is generated, which in case SAVE-INDEX refers to the "last" pointer of the save object refered by SAVE-OBJECT or no pointer exists in the object, the value onone is returned to signal that the scan of the object should be terminated. Otherwise SAVE-INDEX is updated to describe the "next" pointer of the save object. In case the value of the "next" pointer is onone, the pointer is skipped, i.e. iterate this description, otherwise the value of the refered pointer is returned.

t-seto	
*	force TOS value; check TOS type(OADDR);
*	pop;
	Code is generated that inserts the value described by TOS into the pointer variable refered by SAVE-OBJECT and SAVE-INDEX. Note that t-seto does not update SAVE-INDEX. TOS is popped.

Important note:

- Only the OADDR part of a general address should be updated. This instruction is issued by the garbage collector during the storage compaction, and objects are always moved as a whole.

8. DYNAMIC QUANTITIES

access_instruction

::= setobj ! getobj

::= access oindex:byte attribute:tag

::= accessv oindex:byte attribute:tag

The addressing of dynamic quantities poses a problem as their descriptors are incomplete. Dynamic quantities are continually being created and destroyed during program execution, and the S-compiler is not in control of their creation and allocation in storage; this task is the responsibility of the run time system.

In order to complete the definition of descriptors of dynamic objects the S-compiler must provide a mechanism for associating object addresses (to be generated at run time) with natural numbers or object indices known at compile time. The access instructions described below manipulate such associations.

The scheme adopted should give complete freedom to the S-compiler in choosing an appropriate implementation strategy; this can have a considerable effect on run-time performance. The upper bound on the indices is implementation-defined, but it must be greater than 7.

A well-known implementation technique using a "display vector" being continually updated during program execution is for instance catered for in this proposal; as a matter of fact that is the reason why the numbers associated with the objects are called "object indices".

setobj (dyadic)	
*	force TOS value; check TOS type(INT);
*	force SOS value; check SOS type(OADDR);
*	pop; pop;
	The object addressed by the value described by SOS will at run time be associated with the value of TOS, destroying any previous association of TOS. TOS and SOS are popped from the stack.

getobj	
*	force TOS value; check TOS type(INT);
*	pop;
*	push(VAL, OADDR, "value ass. with value(TOS)");
	The object address previously associated with TOS value (through setobj) is retrieved, and a new descriptor is created to describe this object address. It replaces TOS.

access oindex:byte attr:tag	
*	push(REF, attribute.TYPE, "OADDR associated with oindex", attribute.OFFSET);
	This instruction may be approximately described as a getobj followed by a select. A descriptor of the attribute, regarded as a component of the object associated with oindex, will be pushed onto

	the stack.

accessv oindex:byte attr:tag	
*	push(REF, attribute.TYPE, "OADDR associated with oindex", attribute.OFFSET);
*	force TOS value;
	This instruction has the same effect as an access oindex:byte attr:tag followed by a fetch.

9. ARITHMETICAL AND BOOLEAN INSTRUCTIONS

arithmetic_instruction

::= add ! sub ! mult ! div ! rem ! neg

::= and ! or ! xor ! imp ! eqv ! not

::= compare relation

add, sub, mult, div (dyadic)	
*	force TOS value; check TOS type(INT,REAL,LREAL);
*	force SOS value; check SOS type(INT,REAL,LREAL);
*	check types equal;
*	pop; pop;
*	push(VAL, type, "value(SOS) op value(TOS)");
	SOS and TOS are replaced by a description of the value of the application of the operator. The type of the result is the same as the type of the operands. SOS is always the left operand, i.e. SOS op TOS. All arithmetic on subranges of INT should be performed in full integer arithmetic.

rem (dyadic)	
	Remainder, defined as "SOS - (SOS//TOS)*TOS". Syntax and semantics as for mult except that INT is the only legal type. Note that SIMULA demands "truncation towards zero" for integer division. Thus (except for a zero remainder) the result of rem has the same sign as the result of the division. In more formal

	terms: i div j = sign(i/j) * entier(abs(i/j)) i rem j = i - (i div j) * j where '/' represents the exact mathematical division within the space of real numbers.

neg

* force TOS value; check TOS type(INT,REAL,LREAL);
* value(TOS) := - value(TOS);

TOS is replaced by a description of the TOS value with its sign inverted.

Note: Arithmetic operations can result in an interrupt situation at run time. Handling of such situations are treated in more detail in the document "The Environment Interface" (4).

and, or, xor, imp, eqv (dyadic)

* force TOS value; check TOS type(BOOL);
* force SOS value; check SOS type(BOOL);
* pop; pop;
* push(VAL, BOOL, "value(SOS) op value(TOS)");

TOS and SOS are replaced by a description of the result of applying the operator. Note that SOS is the left operand.

(xor: exclusive or, imp: implies, eqv: equivalence)

not

* force TOS value; check TOS type(BOOL);
* value(TOS) := not value(TOS);

TOS is replaced by a description of the negated TOS value.

compare relation (dyadic)

* force TOS value; force SOS value;
* check relation;
* pop; pop;
* push(VAL, BOOL, "value(SOS) rel value(TOS)");

TOS and SOS replaced by a description of the boolean result of evaluating the relation. SOS is always the left operand, i.e. SOS rel TOS. Valid type-relation combinations are given in the table below.

relation
::= ?lt ! ?le ! ?eq ! ?ge ! ?gt ! ?ne

	?lt	?le	?eq	?ge	?gt	?ne
BOOL	+	+				
CHAR	+	+	+	+	+	+
INT	+	+	+	+	+	+
REAL	+	+	+	+	+	+
LREAL	+	+	+	+	+	+
SIZE	+	+	+	+	+	+
AADDR	+	+				
OADDR	+	+	+	+	+	+
GADDR	+	+				
PADDR	+	+				
RADDR	+	+				
resolved_structure	+	+				

Table of legal relational operations

(+ marks valid relation for the designated type)

SOS and TOS must be of the same resolved type (i.e. of the same type, possibly resolved with the same fixref-count), otherwise: error.

SOS is compared with TOS (i.e. SOS rel TOS), yielding a boolean value. TOS and SOS are then popped off the stack. In case a relation occurs in a compare statement, a descriptor of the boolean value is pushed, in if and skipif statements the value is incorporated in the conditional branch sequences generated.

Comparison between character values is done according to the ISO 646/2022 code (i.e. the corresponding integer values are compared).

Remark: ISO 646 is the 7-bit code, 2022 is the (full) 8-bit code. 2022 does not, however, define the actual encoding but just refers to 646 for the subset corresponding to the most significant bit being 0; the remainder is then defined through structural equivalence. It is necessary to conform to 2022, since CHAR is defined to ba 8 bits.

Assuming an integer representation of SIZE values, comparison is performed by comparing the numerical values of this representation. OADDR values are compared by comparing the corresponding machine addresses (regarded as ordinal numbers).

Comparison between quantities of structured types is performed component by component.

10. TYPE CONVERSION

convert_instruction

::= convert simple_type

TOS must be of simple type, otherwise: error.

The TYPE of TOS is changed to the type specified in the instruction, this may imply code generation. Not all conversions are valid, see the table below. An attempt to perform an invalid conversion is an error.

The conversion performed will in some cases be illegal because of the actual value; one example would be to try to convert a REAL to an INT, if the actual value of the REAL is outside the range of INT. Such errors should be checked for at run time, in the cases where they can occur. These conversions are marked ? in the table. The handling of these errors at run time are described in (4).

Conversion from a GADDR to OADDR (AADDR) means: take the object address (attribute address) part of the general address and return as result. An object address OADDR may be converted to a general address GADDR. In that case the object address is extended with an empty attribute address and the pair comprises the result.

REAL (LREAL) to INT conversion is performed after the rule

INT = entier(REAL + 0.5).

(Entier: the greatest integer not greater than the argument). The conversion should be done exactly.

Observe that this rule does not preserve symmetry around zero; e.g. +0.5 is converted to 1, while -0.5 is converted to 0.

L A O G P R

to: B C R R S A A A A A

O H I E E I D D D D D

O A N A A Z D D D D D

from: L R T L L E R R R R R

BOOL .

CHAR . +

INT ? . ? ?

REAL ? . ?

LREAL ? ? .

SIZE .

AADDR .

OADDR . +

GADDR + + .

PADDR .

RADDR .

Table of legal conversions

. - always a legal conversion, but a null operation

+ - always legal and exact

? - the legality depends on the actual value being converted. Loss of accuracy is not considered an error when converting from integer values to real values. In other cases execution time checks may have to be inserted in order to avoid loss of information due to truncation.

blank - always illegal.

11. TRANSFER OF CONTROL

Branches in the executing program may be specified in several ways. The labels that occur in the source program, correspond to the general labels treated in section 1. Some source language constructions such as loops will generate branches, but at the corresponding label it is known from where the branch came. Consequently a series of instructions handle such specific labels, this is treated in section 2. Sections 3 and 4 treat the skip and if statements respectively.

11.1. General labels

label_declaration

::= label_specification ! label_definition

label_specification

::= labelspec label:newtag

label_definition

::= label label:spectag

goto_instruction

::= goto

General labels permit unlimited transfer to program points outside routines, they correspond to labels in the source program. Inside routine bodies general labels may not be declared. The stack must be empty at label and after goto.

labelspec label:newtag

An incomplete label descriptor is created, with no associated program point. The program point must be defined later by a label definition.

label label:spectag

* check stack empty;

If the label has been specified, the corresponding descriptor is modified to describe the current program

point. Otherwise a new descriptor is created of the form (VAL, PADDR, "current program point").

goto

* force TOS value; check TOS type(PADDR);
* pop; check stack empty;

TOS is popped and instructions generated to perform the control transfer.

11.2. Specific labels

jump_instruction
::= forward_jump ! forward_destination
::= backward_jump ! backward_destination

forward_jump
::= switch switch:newtag size:number
::= fjumpif relation destination:newindex
::= fjump destination:newindex

forward_destination
::= sdest switch:tag which:number
::= fdest destination:index

backward_jump
::= bjump destination:index
::= bjumpif relation destination:index

backward_destination
::= bdest destination:newindex

Specific labels are generated as part of the translation process from the source program into its S-code representation. Each instance of such a label may only be referenced once, and this knowledge should simplify the label management in the S-compiler and make it possible to optimise the code generated.

Such program points are not identified by tags or descriptors; they are accessed by means of an "index". An index may be either undefined (syntactically "newindex"), it may refer to a particular program point (syntactically "index").

The constructions described in this section do not have to be properly nested, i.e. the destination of a forward jump may be located before or after the destination of a following forward jump etc. But such jump-destination sets must be fully enclosed either in one routine or in the main program.

Examples:

```
switch SW 3
...
?> sdest SW 0 ( two labels, because - )
> bdest LL1 ( - two jumps to here )
! ...
?!> sdest SW 1
! ...
bjump LL1
...
?> sdest SW 2

< fjump LF1
...
...
> bdest LB1
...
< fjump LF2
...
> fdest LF1
...
<- bjump LB1
...
> fdest LF2
```

(observe that the jump-destination sets do not need
to be properly nested)

switch switch:newtag size:number

* force TOS value; check TOS type(INT);
* pop;
* check stack empty;

The switch:newtag will be associated with a descriptor which identifies an ordered set of "size" destinations "D(0:size-1)", each of which must later be located by an sdest instruction. The value described by TOS is used to generate a jump to the required one, i.e. goto D(TOS). No range checking is implied, the necessary code will have been generated by the front-end compiler.

sdest switch:tag which:number

* check stack empty;

The tag must have been defined in a switch instruction, and the number must be within the range defined by the corresponding switch instruction, otherwise: error.

The destination "D(which)" of the switch instruction defining the tag is located at the current program point.

fjumpif relation destination:newindex (dyadic)

* force TOS value; force SOS value;
* check relation;
* pop; pop;

The destination must be undefined, and TOS and SOS must be of the same permissible resolved type with regard to the relation given, otherwise: error.

A conditional forward jump sequence will be generated, branching only if the relation (see chapter 9) evaluates true. The destination will refer to an undefined program point to be located later (by fdest).

fjump destination:newindex

* check stack empty;

The destination must be undefined,otherwise: error.

A jump to the (as yet unknown) program point is generated, and the destination becomes defined.

fdest destination:index

* check stack empty;

The destination must have been defined by a fjump or fjumpif instruction, otherwise: error.

The current program point becomes the destination of the jump-instruction and the destination becomes undefined.

bjumpif relation destination:index (dyadic)

* force TOS value; force SOS value;
* check relation;
* pop; pop;

The destination must be defined by a bdest instruction, and TOS and SOS must be of the same permissible resolved types with regard to relation, otherwise: error.

A conditional jump sequence will be generated, branching only if the relation evaluates true. The destination becomes undefined.

bjump destination:index

* check stack empty;

The destination must have been defined in a bdest instruction, otherwise: error.

A jump to the referenced program point is generated, and the destination becomes undefined.

bdest destination:newindex

* check stack empty;

The destination must be undefined, otherwise: error.

The destination is defined to refer to the current program point.

11.3. Skip statement

skip_statement

::= skipif relation <program_element>* endskip

skip_instruction

::= skipif relation <instruction>* endskip

The skip_statement is intended to be used where a transfer of control is to be generated without altering the state of the stack, commonly to report error conditions during expression evaluation. The skip instruction is the form the statement takes inside routine bodies.

skipif relation

* force TOS value; force SOS value;
* check relation;
* pop; pop;
* save skip-stack;

The generated code will compute the relation, and control is transferred to an "end-label" (to be defined later), if the relation is true. A copy of the complete state of the S- compiler`s stack is saved as the "skip-stack".

endskip

* check stack empty; restore skip-stack;

If it is possible for control to reach the current program point, a call on a suitable run time error routine must be inserted at the end of the generated skip-branch. This will be the interrupt handler described in (4).

The "end-label" is located at the current program point, and the "skip-stack" is restored as the current stack.

11.4. if-statement

if_statement

::= if relation <program_element>* else_part

else_part

::= else <program_element>* endif ! endif

if_instruction

::= if relation <instruction>* i else_part

i else_part

::= else <instruction>* endif ! endif

While the control transfer instructions covered in the preceding sections demand that the stack be empty at the destination, this demand is relaxed in the usage of the constructions defined here. In case the stack is non-empty at the point where different control branches merge, the contents must be meaningful, that is:

- the stack depth must be the same in both merged branches, and
- the types of corresponding elements must be the same.

Such merges take place only at endif.

To simplify the implementation of the if-construction we adopt the following restrictions:

The if-construction will end up with a mode value item on top of the stack while the rest of the stack remains the same. This item is produced in one of two different ways:

- The current TOS is modified in both if-branches.
- A new stack item of the same type is produced in each of the if-branches.

A force TOS value is performed in both if-branches.

The statement will generate target code to be executed conditionally, i.e. if the condition holds the if-part will be executed otherwise the elsepart (if any). It is comparable to very simple if-statements in other languages. The if instruction is the restricted form this construction takes inside routine bodies.

if relation

* force TOS value; force SOS value;

* check relation;

* pop; pop;

* remember stack as "if-stack";

The generated code will compute the value of the relation, and transfer control to an "else-label" (to be defined later) if the relation is false. A copy of the complete state of the S-compiler's stack is saved as the "if-stack".

else

* force TOS value;
* remember stack as "else-stack";
* reestablish stack saved as "if-stack";

An unconditional forward branch is generated to an "end-label" (to be defined later). A copy is made of the complete state of the stack and this is saved as the "else-stack", then the stack is restored to the state saved as the "if-stack". Finally the "else-label" (used by if) is located at the current program point.

endif

* force TOS value;
* merge current stack with "else-stack" if it exists, otherwise "if-stack";

The current stack and the saved stack are merged. The saved stack will be the "if-stack" if no else-part has been processed, otherwise it will be the "else-stack". The merge takes each corresponding pair of stack items and forces them to be identical by applying fetch operations when necessary - this process will generally involve inserting code sequences into the if-part and the else-part. It is an error if the two stacks do not contain the same number of elements or if any pair of stack items cannot be made identical. After the merge the saved stack is deleted.

If no else-part was processed the "else-label", otherwise the "end-label", is located at the current program point.

12. SEGMENTATION CONTROL

segment_instruction

::= bseg <program_element>* eseg

This instruction specifies that the enclosed program elements are out of sequence, i.e. the code generated must either be located elsewhere or it must be preceded by an unconditional jump instruction leading to the program point following eseg. The segment instruction is illegal within routine bodies.

The purpose of the segment instruction is to be able to generate e.g. code for a "thunk" in the natural S-

program context, without having to worry about whether to generate jumps around it etc. The intention is that the enclosed elements from the point of view of a sequential scan through the surrounding code should be completely invisible, i.e. the following piece of S-code:

sequence-1 bseg sequence-2 eseg sequence-3

will generate target code for sequence-1 and sequence-3 in direct control sequence while sequence-2 will be located somewhere that is unreachable except through an explicit transfer of control (goto or the like).

Note that all jump/destination sets must be fully enclosed within a segment.

An S-program will contain many such segment constructions. The S-compiler has complete freedom to decide where to locate the corresponding enclosed program segments.

The segment construction has no implied effect on the allocation of data or constants.

The bseg and eseg have similar effects on the stack as save and restore (chapter 7).

bseg

* remember stack;
* purge stack;

The current program point is remembered together with the complete state of the stack, the "bseg-stack". Some new segment is designated the current program point, together with a new, empty stack.

eseg

* check stack empty;
* reestablish stack remembered at corresponding bseg;

If it is possible for control to reach the current program point, a call on a suitable run time error routine must be inserted at the end of the generated program segment. This will be the interrupt_handler routine described in (4).

The "bseg-stack" is restored together with the saved program point.

13. ROUTINES

Routines in S-code correspond to subroutines in other languages but with certain restrictions:

- All parameters are passed to and from the routines by value.
- Routines are not recursive (but the routine may possibly contain calls of the routine itself. In that case the body of the routine will explicitly store and reload the necessary values around such recursive calls).
- Any routine must exit through its endroutine, it is not possible to exit via any explicit jump.
- The return address may be made available to the routine (by means of an exit definition), thus allowing the return address to be changed by the routine itself.

The definition of a routine creates two descriptors (of type VOID), describing the routine profile and the routine body. In certain cases to be described later no body occurs in the program.

The profile defines the parameter and exit descriptors for the routine, while the body defines the local descriptors as well as the instruction sequence to be executed when the routine is activated. Each routine body must have exactly one routine profile associated with it, whereas a "non-peculiar" profile may be associated with several routine bodies. It is not allowed for the same profile to be associated with more than one body in any dynamic sequence of routines calling routines, since this would imply re-use of the (static) allocation record defined by the profile.

The activation sequence for a routine is generated on the basis of the call instruction treated in section 4 below.

13.1. Routine profiles

routine_profile

::= profile profile:newtag <peculiar>?

<import_definition>* <export or exit>?

endprofile

peculiar

::= known body:newtag kid:string

::= system body:newtag sid:string

::= external body:newtag nature:string xid:string

::= interface pid:string

import_definition

::= import parm:newtag quantity_descriptor

export_or_exit

::= export parm:newtag resolved_type

::= exit return:newtag

The import (input) parameters and the export (return) parameter are transmitted "by value". Each import (export) definition will declare a quantity local to the routine body (bodies) later associated with the profile. The order in which the parameters are given in the profile will define the correspondence between the formal parameter locations and the assignment of the parameter value (asspar in the call instruction); the tag values (used in the associated bodies) will also be allocated in this sequence.

An import parameter defined as a repetition must correspond to an assrep in the call, the count specifies the maximum permissible number of values to be transferred. Note that the actual count is not transferred, it should be defined as a separate import parameter.

An exit definition identifies a descriptor for the area containing the return address of the routine. If no exit definition is given an an-onymous descriptor will be created for the same area. The exit des-criptor is of the basic form (REF, PADDR, ...) and exists for the use of endroutine (q.v.). If the descriptor is identified through exit it becomes accessible to the routine exactly as any other local quantity, and in that case it will be possible for the routine to change its return address. Such a routine cannot be called from other routines.

In general, only the profile:newtag and a possible body:newtag (in case the routine is peculiar) are visible outwith any routine body associated with the profile. Observe that routine parameters and exit tags are local quantities.

The parameter tags may, however, occur in the tag list of a module (see chapter 14).

Remark: The above sentence was included for the time being, because of special implementation strategy chosen by a subimplementor. The number of the external tags in the run-time system will be drastically reduced when these tags are removed from the tag list.

13.2. Peculiar routines

"Peculiar" provides information about routines whose bodies either will not be defined in any S-program, or in the case of known routines may be safely skipped by the S-compiler. Except for known no body may be associated with the profile; it uniquely defines one routine. The body tag so specified will be used to refer to the routine in subsequent call statements.

Peculiar routines are described in this way to permit the S-compiler to handle each one in the most convenient system- and routine- dependent manner. In particular it is common for system-provided routines to return the result in specific registers and not in store locations. In these cases the use of peculiar routines will ensure that the export descriptor, stacked following the call, describe the appropriate registers.

All peculiar routines are identified by a string (kid, sid, xid or pid) in the profile, since the actual tags associated with such a routine will vary from S-program to S-program. The string contains at most 6 characters with the case of any character being insignificant (e.g. "a" is equivalent to "A"). All id strings for peculiar routine profiles contain only alphanumeric characters, the first of which is a letter.

known body:newtag kid:string

A known routine has a body defined in S-code. The S-compiler may know the working of the routine and may thus replace the body with an optimised code sequence. It is intended to be used in cases where a standard S-code routine will be in danger of being inefficient in some implementations, or when the routine call can be replaced by an in-line code sequence at each call.

It should however be noted that it is necessary to simulate (in the S-compiler) a routine spec for the body tag when processing known profiles.

system body:newtag sid:string

System profiles provide the interface to the run time environment of the program, or they represent routines which are impossible to program in S-code (or potentially prohibitively inefficient in all implementations). Thus no body will be given. Such routines (e.g. date_and_time) are typically provided by the operating system on the target machine, and may require special intervention from the S-compiler, since the calling conventions and parameter passing mechanisms will be system-dependent.

external body:newtag nature:string xid:string

External routines are routines written in other languages. The exact nature of the routine is specified by the nature:string. External routines are implementation dependent.

interface pid:string

Interface profiles may occur in the head of the interface module only. The profile becomes visible from the run time environment through the identifying string.

Every profile associated to a routine address evaluation which is an actual parameter to a system routine should be specified as an interface profile. An interface profile cannot have more than ony body.

In fact, the interface specification is redundant, but can be used to simplify code production for routine address values.

Interface routines cannot have more than one body.

Note that when processing a known profile it is necessary to simulate a routine-spec for the body tag in the S-compiler.

13.3. Routine bodies

routine_specification

::= routinespec body:newtag profile:tag

routine_definition

::= routine body:spectag profile:tag

<local_quantity>* <instruction>* endroutine

local_quantity

::= local var:newtag quantity_descriptor

The body:newtag identifies the routine body and is used as an argument to call, while profile:tag connects the body to the relevant routine profile and is used as an argument to precall, asscall or repcall.

Routine specification is used in cases where it is necessary to call an as yet undefined routine, and in module heads.

A routine body will be compiled into a sequence of instructions to be executed when activated by a call statement referencing the body tag. No quantities or descriptors defined in the body are visible outwith that body; all such tags are destroyed at endroutine and made available for re-use. As a consequence the syntax restricts the use of certain constructs inside the routine body:

- constants and general labels cannot be defined inside the routine,
- external modules cannot be inserted into routine bodies (see 14.3),
- routines cannot be nested, i.e. routine specifications or definitions cannot occur in a body,
- all jump/destination sets must be fully enclosed within the body, except for destinations of goto jumps.
- protect statements are not allowed inside routines.
- if, skip, and segment constructions are restricted when used inside routine bodies, the restriction is syntactically specified as e.g. "if_statement" (invalid) versus "if_instruction" (valid). The constructions are simply restricted so as not to allow any of the above restrictions be violated.

local var:newtag quantity_descriptor

The local quantities constitute the local variables for the routine. Conceptually they are static quantities, i.e. the descriptors will be complete. An implementation may choose to make them dynamic, that is created each time the routine is activated, and destroyed at routine exit.

While the tags identifying the local variables not necessarily are generated in strict sequence, they will at endroutine comprise an interval.

routine body:spectag profile:tag

An implied bseg (see chapter 12) is performed.

The scope to the parameter tags defined in the associated profile is opened, and all defined destination indices and label tags will temporarily lose their meaning.

endroutine

All destination indices must be undefined, and the stack must be empty, otherwise: error.

An implicit eval is performed. Code is generated to transfer control to the program point described by the exit descriptor, the scope to the profile is closed, and an implicit delete is performed with the minimum local tag value defined in the routine as argument. The label tags and the destination indices defined before routine was evaluated regain their meaning. Observe, that since routines cannot be lexically nested, it is sufficient to use (the equivalence of) two destination index vectors. Finally an implied eseg is performed.

13.4. Routine activation

call_instruction

::= connect_profile <parameter eval>* connect_routine

connect_profile

::= precall profile:tag

::= asscall profile:tag

::= repcall n:byte profile:tag

connect_routine

::= call body:tag ! <instruction>+ call-tos

parameter eval

::= <instruction>+ asspar

::= <instruction>+ assrep n:byte

The call instruction causes generation of the activation sequence for a routine. First the profile is connected, this will provide information about the number and types of the parameters. Following this the parameter values are evaluated and transferred, and finally the actual routine to be activated is connected, either explicitly by giving the tag or implicitly by taking its RADDR value from TOS.

precall profile:tag

A copy of the profile descriptor is pushed onto the stack. This descriptor defines the number and types of the parameters required and where they are to be placed. Initially it identifies the first import parameter (if any). The descriptor will be modified during the further processing of the call instruction.

For each import parameter a parameter evaluation sequence follows, specifying instructions which result in a descriptor of the actual parameter value on top of the stack.

call body:tag

The profile descriptor must now be in TOS, and it must not describe any import parameter, otherwise: error.

The generated code will transfer any parameters from the temporary area to the parameter locations, and then perform a subroutine call to the body. The return address (the current program point) will be remembered in the (explicitly named or implicitly defined) exit area. How and when this remembering is done, is implementation dependent.

The profile descriptor is popped. If it indicates an export parameter, the corresponding descriptor is pushed onto the stack as (VAL, export.TYPE, ...).

asscall profile:tag

TOS must describe a quantity of the same type as the first import parameter to the given profile, otherwise: error.

asscall makes it possible to call a routine when the first import parameter is already on the stack. The evaluation of this construction will take place as if the TOS element first is moved off the stack into some anonymous location, and then a precall is simulated. Following this the former TOS is moved back to the stack on top of the profile descriptor, and finally an asspar is simulated.

repcall N:byte profile:tag

TOS must describe a quantity of the same type as the first import parameter to the given profile. This must be a repetition.

repcall makes it possible to call a routine when the first import parameter is already on the stack, and this is a repetition. The argument N must be less than or equal to the number of elements in the repetition, and the type of the 'N' uppermost stack descriptors must be identical to the type specified for the parameter. If these conditions are not satisfied, the S- program is in error.

The evaluation of this construction will take place as if the N uppermost elements of the stack are first moved into some anonymous location, and then a precall is simulated. Following this the former N stack elements are moved back to the stack on top of the profile descriptor, and finally an assrep is simulated.

asspar

At this stage SOS must be the connected descriptor (possibly modified by earlier parameter assignments) and it must describe an import parameter of the same type as TOS, otherwise error.

The generated code will move the value of TOS into the temporary area. The stack is popped and the profile descriptor is modified to describe the next parameter. After a parameter assign the profile descriptor will remain on top of the stack.

assrep N:byte

The profile descriptor in stack element number N+1 from the top must describe an import parameter, which must be a repetition. The argument N must be less than or equal to the number of elements in the repetition, and the types of the 'N' uppermost stack descriptors must be identical to the type specified for the parameter. If these conditions are not satisfied, it is an error.

The 'N' values described by the top elements in the stack are assigned to the elements of the repetition in the following order:

The top element is assigned to the N'th element of the repetition and then popped; then the new top element is assigned to the (N-1)'th repetition element and popped, and so on until the N stack elements in question have been assigned to the N first elements of the repetitive parameter.

These assignments imply generation of deferred code. TOS will now be the routine descriptor, which is modified to describe the next parameter.

This facility is intended to be used to pass a variable number of parameters of the same type to certain routines, in particular to routines associated with array accessing.

call-tos

TOS must be of type RADDR, and SOS must be the (modified) connected profile descriptor, which does not describe any import parameters, otherwise: error.

This is equivalent to "call TOS", i.e. the routine address popped off the stack identifies the entry point to a routine to be connected. It will normally not be possible for the S-compiler to check, whether TOS actually is associated with the profile. However, the value nobody should lead to a run time error.

13.5. Parameter transmission

Since OADDR and GADDR values are permitted as routine parameters, the transmission of the parameters cannot in the general case take place directly into the parameter locations. In order to make it possible for the garbage collector to identify such values, if the g.c. is activated during parameter transmission, the values must be transmitted as intermediate results, i.e. in the temporary area. All calls to the g.c. will be preceded by save, thus consistency will be ensured.

The actual transfer of the values from the temporaries to the parameter locations associated with the profile cannot take place before call.

14. SEPARATE COMPILATION

14..1. Module definition

module_definition

::= module module_id:string check_code:string

visible_existing

body <local_quantity>* <program_element>* endmodule

visible_existing

::= <visible>* tag_list ! existing

visible

::= record_descriptor ! routine_profile

::= routine_specification ! label_specification

::= constant_specification ! insert_statement

::= info_setting

tag_list

::= < tag internal:tag external:number >+

Rather than compiling a complete program, the S-compiler can be instructed to compile a part of a program (a module) which subsequently will be referenced by another module or a main program.

The module_id is a system-unique system-dependent identification for the module. It will be used to locate module descriptors referenced in insert statements, or the descriptor of the module itself if it is specified existing.

The check_code is an implementation-dependent code which can be used to determine the validity of the correspondence between a module object and the compiled version of the module it purports to describe. One possibility for this check_code is an accurate representation of the time and date of compilation.

An existing specification in the module head informs the S-compiler that the module has been compiled previously, and that the module head was unchanged (as well external as internal tags occurring in the head are the same). The visible attributes of the module will in this case be those

specified visible in the earlier compilation, but changes may have been made in the module body. Such changes do not however influence these attributes, and the S-compiler can obtain the module head from the module descriptor file. This means that the compilation of the module will not require recompilation of other modules referencing this module. The check_code given in the module head must then match that in the existing attribute file.

If the S-compiler encounters a module with existing specified, the check_code given in the module head must match that in the existing attribute file.

If the module is not specified existing, the list of descriptors specified in the module head specifies the externally visible attributes of the module. In order to be able to create unique tag identification of the visible elements, the tag list specifies a correspondence between the internal tags used in the module and an external number. Only tags specifically mentioned in the tag_list are transferred via the insert instruction given in another compilation unit; the tags defined through an insert instruction cannot themselves be made visible outwith the module being compiled.

The routine specifications refer to routines defined within the module body, neither routine profiles nor other definitions given in the head may be repeated in the body. If a profile tag occurs in the tag_list, all the parameter tags may also occur, since the profile tag may be associated with a routine body in another module. Label or constant specifications refer to quantities defined in the body.

Any local quantities and descriptors are visible only from within the body of the module; all such tags are destroyed at endmodule and made available for re-use. They are statically allocated and part of the module throughout an execution, by analogy with Algol's 'own'-variables. Their initial values are undefined.

14.2. The interface module

interface_module

::= global module module_id:string check_code:string

<global_interface>* tag_list

body < init global:tag type repetition_value >*

endmodule

global_interface

::= record_descriptor

::= constant_definition < system sid:string >?

::= global_definition < system sid:string >?

::= routine_profile

::= info_setting

global_definition

::= global internal:newtag quantity_descriptor

The interface module serves two main purposes:

- it specifies a set of statically allocated variables (the globals),
- it specifies the interface to the system environment.

An interface module is inserted as any other module with the restriction, that only one particular interface module can ever occur in the executing program, i.e. it must be checked that different modules do not insert different versions of the interface module.

A complete listing of the interface module may be found in (4).

MAXLEN

This is a predefined SIZE constant with an implementation dependent value; it is the c-size of the temporary area (page 35). Note that it is not the same as the ALLOCATED attribute of this area; ALLOCATED will always be less than or equal to the value of MAXLEN.

This constant is needed since it is necessary to have a special area available for a save object in case it is not possible to allocate one dynamically without garbage collection.

14.2.1. Global variables

The global definition specifies static allocation of a variable of the given type, therefore the corresponding descriptor will be complete. Each such variable is an object, and may thus be addressed either by an object or a general address.

A global variable may be initialised to a given value through the init statement. This initialisation must take place before execution of the program proper take place, e.g. in connection with main. It is of no concern whether this is done by preloading the values, or by executing code corresponding to the init statements. If a global variable is not so initialised, the initial value is undefined.

14.2.2. The environment interface

The S-code specification of the environment interface is given in the interface module in the form of system profiles, system globals, and system constants.

The system constants are assigned values during the generation of the S-code program. Values may also be assigned to system globals during this process, using init. In both cases the S-code program is only providing default values, which the S-compiler may chose to overwrite with values appropriate for the particular implementation.

All system routines are defined in this module; it is the responsibility of the S-compiler to insert the actual instruction sequences corresponding to each routine into the body.

14.3. The macro definition module

macro_definition_module

::= macro module module_id:string check_code:string

<macro_definition>* endmodule

The macro definition module contains the definitions of all macros used by the front end Simula compiler (the run time system S-code will never contain macro calls). In systems where this facility is implemented (see below) this module is implicitly inserted in the S-compiler when it processes S-programs generated by the Simula front end compiler. Only one macro definition module will occur in one particular release of the portable system.

Macros are introduced for two purposes:

- S-code compression, and
- code generation optimisation in the S-compiler.

The front end compiler will be able to either generate macro calls or generate the expansion of the call; this will be governed by a switch. Thus a particular S-compiler need not implement the macro facility at all, as long as the Simula front end compiler is installed with the macro expansion switch on.

The macro definition module will be accompanied by a listing of all macro definitions, giving the symbolic S-code to be substituted at each macro call together with the rationale for the particular macro, and a specification of the kind of the parameters and of the parameter insertion points. This will facilitate efficient code generation for the macro expansions.

14..3..1. Macro definitions

macro_definition

::= macro macro_name:byte macro_parcount:byte

known macro_id:string

<macro_body_element>*

endmacro

macro_body_element

::= mark macro_sequence:string

::= mpar macro_parnumber:byte

The macro definition determines the substitution schema for the macro call processing (see 14.3.2).

The macro sequences of the body should not (and cannot) be analysed when the definition is processed, and nested definitions are illegal. The macro_parcount is the number of actual parameters given when the macro is called, i.e. zero means no parameters, one means exactly one parameter in the call etc. The points at which some actual parameter is to be inserted when the macro is expanded are marked by the mpar-construction, and the parnumber refer to the place in the actual parameter list (thus parnumber cannot be zero).

The macro_id names the macro in a more reasonable manner than the pair (module_id, macro_name) does; it bears no other significance.

14..3..2. Macro expansion

macro_call

::= mcall macro_name:byte <actual_parameter:string>*

The effect of a macro call upon the S-compiler will be described as if the macro expansion should take place by textual replacement in the input stream. It is emphasised that this description is for the purpose of definition only, other and more efficient methods are of course legal as long as the described effect is achieved.

When a macro call is recognised the complete state of the S-compiler is saved and the S-compiler enters macro expansion mode. The corresponding macro definition is identified, and the call is scanned (the number of actual parameters is given by the definition). During the scan each actual parameter is assigned its parameter number, beginning at one. The macro call is now replaced "textually" after the following rules:

- All macro body elements are processed in the sequence they occur in the definition.
- If the element is a macro sequence, the string contents is inserted, i.e. the "length byte" of the string is removed.
- If the element is a parameter number, the contents of the corresponding actual parameter string is inserted in the same manner.

When the replacement defined above has taken place, the S-compiler leaves expansion mode, restores the state saved and continues processing with the first inserted byte.

14.4. Inclusion of a module

insert_statement

::= insert module_id:string check_code:string

external_id:string tagbase:newtag taglimit:newtag

::= sysinsert module_id:string check_code:string

external_id:string tagbase:newtag taglimit:newtag

This instruction causes the S-Compiler to include a module. The external_id is used to identify the module with respect to an operating system. If the single character ? is given as an external_id the S-Compiler should search the module definition library for a module identified by the module_id. If it cannot be found or if the check code does not match, the S-Code is in error.

sysinsert is used for system modules RTS, simob etc., while insert is used for user modules e.g. seperately compiled classes.

The visible objects of the module (as specified in the tag_list) are now brought into the current compilation unit by adding tagbase to the external numbers specified, checking that taglimit is not exceeded. Thus tags are allocated from the range (tagbase..taglimit).

Each insert instruction will have this effect, and the same module may be included several times in the same compilation unit. The consequence of that is to create several intervals of tags, refering the same elements in the included module.

14.5. Linkage of modules

The linkage of the executable code will generally be done in a manner standard to the target system. Some knowledge about the modules must, however, be communicated to the S-compiler: identification and type binding etc. of names external to the program being compiled. Such information is procured from a data base (the module definition library) maintained by the S-compiler itself. The naming conventions used, the structure of and access method to this data base are highly system-dependent.

15. INITIALISATION OF ALLOCATED AREAS

area_initialisation

::= zeroarea

::= initarea resolved_type ! dinitarea structured_type

For the purpose of giving dynamically allocated areas sensible initial values three instructions are defined. These instructions will always be used in the following manner:

- when an area has been allocated by the system environment, or when a possible garbage collection has returned free storage, the area(s) will be zeroed by zeroarea,
- when a particular area has been acquired (somehow) to be structured by some type, one of the instructions initarea or dinitarea is issued.

This usage pattern will be enforced by the front-end compiler and the run time system, i.e. when initarea is to be evaluated the S-compiler may assume that the area to be initialised has been zero-filled. Thus an implementation may choose to realise either zeroarea or the initarea-pair, or it may choose a

mixed strategy, zero filling the area (zeroarea implemented) and partly implementing (d)initarea for those components which do not have a zero representation. It should be obvious that the complete implementation of all will be redundant and will probably lead to considerable run time overhead.

Important note: Neither the prefix part nor alternative parts must ever be initialised.

zeroarea (dyadic)

* force TOS value; check TOS type(OADDR);
* force SOS value; check SOS type(OADDR);
* pop;

TOS and SOS must be OADDR, otherwise error.

The area between SOS and TOS (SOS included, TOS not) is to be zero-filled, and TOS is popped.

initarea resolved_type

* force TOS value; check TOS type(OADDR);

TOS.TYPE must be OADDR, otherwise: error.

The argument type is imposed upon the area, and the area is initialised according to the table below. Only the common part of an instance of a structure will be initialised, ignoring both the prefix and any alternate part(s). The structure is initialised component by component according to the table below.

dinitarea structured_type (dyadic)

* force TOS value; check TOS type(INT);
* force SOS value; check SOS type(OADDR);
* pop;

TOS.TYPE must be INT, SOS.TYPE must be OADDR, and the structured type must contain an indefinite repetition, otherwise: error.

The value of TOS is used to resolve the type, i.e fixing the number of elements in the indefinite repetition, following that the evaluation proceeds exactly as for initarea.

Area initialisation values

type: initialised to:

BOOL false

CHAR NUL (ISO repr. 0)

INT 0

REAL, LREAL 0.0

SIZE nosize

AADDR anone

OADDR onone

GADDR gnone

PADDR nowhere

RADDR nobody

structure - each attr. init. as above.

Note: if these values are represented as zero and if zeroarea is implemented, the instructions (d)initarea may safely be ignored!

16. OTHER INSTRUCTIONS

eval_instruction

::= eval

In certain cases the code specified by previuos program elements must be generated in order to ensure, that the generated program will behave correctly. As an example, consider the following SIMULA statement:

x.b := a + f;

where "f" is a function that, as a side effect, may change both "a" and "x".

However, the semantics of SIMULA demands strict left-to-right evaluation of statement components. The actual values to be used in the execution of the translated statement must be the values before the function f is executed. Thus it is necessary that the values of x and a used in the computation really are "fetched" into some temporary locations. A possible optimisation might instead remember that these values are stored in specific locations, and use these values directly from the locations "x" and "a". In that case the computed result would be wrong. In such a situation the front end compiler will issue this

instruction.

The instruction forces the S-compiler to perform any operations previously specified in the S-program, which may have been deferred for optimisation purposes. All values and references must be fully evaluated and stored in the temporary area when an eval is done, unless they are constants. Following the instruction the internal state of the S-compiler must match the state implied by a strict application of the S-code definition.

delete_statement

::= delete from:tag

* check stacks empty;

All tags defined with values greater than or equal to from:tag are made undefined, i.e. the corresponding descriptors may be released. The tags become available for reuse. The stack and all saved stacks must be empty, otherwise: error.

The S-code generators (e.g. the SIMULA front-end compiler) will ensure that the following rules are obeyed; since it may be very costly in terms of execution time for the S-compiler to check these conditions, we recommend the use of an internal switch to govern such consistency checks.

Let DEL be the set of tags to be deleted (i.e. the set of tags in the closed interval from...largest defined tag). The following conditions apply to elements of DEL:

a) If a record tag is in DEL, then

- all quantities having this record as type,
- all records having this record as prefix, and
- all attributes of this record

are in DEL.

b) If an attribute tag is in DEL, then

- the associated record tag is in DEL.

c) If a profile tag is in DEL, then

- all body tags associated with this profile
- all parameter tags of this profile

are in DEL.

d) If a parameter tag is in DEL, then

- the associated profile tag is in DEL.

These conditions may be summarised informally as follows: Once a tag is to be deleted, all tags which directly or indirectly refer to this tag are deleted in the same delete statement.

info_setting

::= decl line:number

::= line line:number

::= stmt line:number

::= info string

::= setswitch switch:byte setting:byte

The line, decl and stmt instructions are used to inform about a mapping between a source program and its S-Code. The argument of the instructions refer to the numbering of the lines of the program listing from the S-Code producer. decl informs that the code following is for a SIMULA declaration starting on the line with the supplied number. stmt informs that the code following is for a SIMULA statement starting on the line with the supplied number. line informs that the current point in the S-Code corresponds to the start of the source program line with the given number. The intention is that the Front End Compiler will produce decl and stmt instructions, while line instructions will occur in the code for the Run-Time System.

Eventually these instructions should be used to produce a mapping between the original source program and its corresponding machine-code. For this purpose none of them need occur in the final machine-code. The mapping may be represented by for instance a table.

The stmt instruction may be used for another purpose as well, see chapter 11 of The Environment Interface.

info offers the possibility to communicate to the S-compiler information, which must be interpreted in a system-dependent manner.

setswitch is intended to control various aspects of the working of the S-compiler such as the production of debugging information.

The compiler will maintain a set of switches which can be set to various values: SWITCH(switch) := setting. The meanings of the switches and their settings will be implementation-dependent with the following exceptions:

a. SWITCH(1) controls the listing of the S-code as it is processed by the compiler. Such listing should be done using the mnemonics defined in this document (underline excepted). SWITCH(2) controls the stack activity trace.
b. The setting to zero of a switch inhibits the actions controlled by the switch. This should be the default setting of all switches.

Appendix A

The syntax of the S-code language

(For each production is given reference to its definition)

S-program p. 13
::= program program_head:string
program_body endprogram

program_body p. 13
::= interface_module
::= macro_definition_module
::= <module_definition>*
::= main <local_quantity>* <program_element>*

program_element p. 13
::= instruction
::= label_declaration
::= routine_profile ! routine_definition
::= skip_statement ! if_statement
::= protect_statement ! insert_statement
::= delete_statement

instruction p. 13
::= constant_declaration
::= record_descriptor ! routine_specification
::= stack_instruction ! assign_instruction
::= addressing_instruction ! protect_instruction
::= temp_control ! access_instruction
::= arithmetic_instruction ! convert_instruction
::= jump_instruction ! goto_instruction
::= if_instruction ! skip_instruction
::= segment_instruction ! call_instruction
::= area_initialisation ! eval_instruction
::= info_setting ! macro_call

simple_type p. 15
::= BOOL ! CHAR

::= INT ! REAL ! LREAL ! SIZE
::= OADDR ! AADDR ! GADDR ! PADDR ! RADDR

type p. 15
::= structured_type ! simple_type

resolved_type p. 15
::= resolved_structure ! simple_type
::= INT range lower:number upper:number
::= SINT

quantity_descriptor p. 18
::= resolved_type < rep count:number >?

record_descriptor p. 19
::= record record_tag:newtag <record_info>?
<prefix_part>? common_part
<alternate_part>* endrecord

record_info p. 19
::= info "TYPE" ! info "DYNAMIC"

prefix_part p. 19
::= prefix resolved_structure

common_part p. 19
::= <attribute_definition>*

alternate_part p. 19
::= alt <attribute_definition>*

attribute_definition p. 19
::= attr attr:newtag quantity_descriptor

resolved_structure p. 19
::= structured_type < fixrep count:ordinal >?

structured_type p. 19
::= record_tag:tag

value p. 23
::= boolean_value ! character_value
::= integer_value ! size_value
::= real_value ! longreal_value
::= attribute_address ! object_address
::= general_address ! program_address
::= routine_address ! record_value

repetition_value p. 23
::= <boolean_value>+
::= <character_value>+ ! text_value
::= <integer_value>+ ! <size_value>+
::= <real_value>+ ! <longreal_value>+
::= <attribute_address>+ ! <object_address>+
::= <general_address>+ ! <program_address>+
::= <routine_address>+ ! <record_value>+

text_value p. 23
::= text long_string

boolean_value p. 23
::= true ! false

character_value p. 23
::= c-char byte

integer_value p. 23
::= c-int integer_literal:string

real_value p. 23
::= c-real real_literal:string

longreal_value p. 23
::= c-lreal real_literal:string

size_value p. 24
::= c-size type ! nosize

attribute_address p. 24
::= < c-dot attribute:tag >* c-aaddr attribute:tag
::= anone

object_address p. 24
::= c-oaddr global_or_const:tag
::= onone

general_address p. 24
::= < c-dot attr:tag >* c-gaddr global_or_const:tag
::= gnone

program_address p. 25
::= c-paddr label:tag ! nowhere

routine_address p. 25
::= c-raddr body:tag ! nobody

record_value p. 26
::= c-record structured_type
<attribute_value>+ endrecord

attribute_value p. 26
::= attr attribute:tag type repetition_value

constant_declaration p. 27
::= constant_specification ! constant_definition

constant_specification p. 27
::= constspec const:newtag quantity_descriptor

constant_definition p. 27
::= const const:spectag
quantity_descriptor repetition_value

stack_instruction p. 28
::= push obj:tag ! pushv obj:tag
::= pushc value ! pushlen ! dup ! pop ! empty
::= popall byte

assign_instruction p. 28
::= assign ! update ! rupdate

addressing_instruction p. 31
::= fetch ! refer resolved_type ! deref
::= select attribute:tag ! selectv attribute:tag
::= remote attribute:tag ! remotev attribute:tag
::= index ! indexv
::= inco ! deco
::= dist ! dsize structured_type ! locate

protect_statement p. 37
::= save <program_element>* restore

temp_control p. 37
::= t-inito ! t-geto ! t-seto

access_instruction p. 41
::= setobj ! getobj

::= access oindex:byte attribute:tag
::= accessv oindex:byte attribute:tag

arithmetic_instruction p. 43
::= add ! sub ! mult ! div ! rem ! neg
::= and ! or ! xor ! imp ! eqv ! not
::= compare relation

relation p. 45
::= ?lt ! ?le ! ?eq ! ?ge ! ?gt ! ?ne

convert_instruction p. 46
::= convert simple_type

label_declaration p. 48
::= label_specification ! label_definition

label_specification p. 48
::= labelspec label:newtag

label_definition p. 48
::= label label:spectag

goto_instruction p. 48
::= goto

jump_instruction p. 49
::= forward_jump ! forward_destination
::= backward_jump ! backward_destination

forward_jump p. 49
::= switch switch:newtag size:number
::= fjumpif relation destination:newindex
::= fjump destination:newindex

forward_destination p. 49
::= sdest switch:tag which:number
::= fdest destination:index

backward_jump p. 49
::= bjump destination:index
::= bjumpif relation destination:index

backward_destination p. 49
::= bdest destination:newindex

skip_statement p. 52
::= skipif relation <program_element>* endskip

skip_instruction p. 52
::= skipif relation <instruction>* endskip

if_statement p. 53
::= if relation <program_element>* else_part

else_part p. 53
::= else <program_element>* endif
::= endif

if_instruction p. 53
::= if relation <instruction>* i_else_part

i_else_part p. 53
::= else <instruction>* endif
::= endif

segment_instruction p. 55
::= bseg <program_element>* eseg

routine_profile p. 58
::= profile profile:newtag <peculiar>?
<import_definition>* <export_or_exit>?
endprofile

peculiar p. 58
::= known body:newtag kid:string
::= system body:newtag sid:string
::= external body:newtag nature:string xid:string
::= interface pid:string

import_definition p. 58
::= import parm:newtag quantity_descriptor

export_or_exit p. 58
::= export parm:newtag export_type
::= exit return:newtag

export_type p. 58
::= resolved_type ! range

routine_specification p. 61
::= routinespec body:newtag profile:tag

routine_definition p. 61
::= routine body:spectag profile:tag
<local_quantity>* <instruction>*
endroutine

local_quantity p. 61
::= local var:newtag quantity_descriptor

call_instruction p. 63

::= connect_profile <parameter eval>*
connect_routine

connect_profile p. 63
::= precall profile:tag
::= asscall profile:tag
::= repcall n:byte profile:tag

connect_routine p. 63
::= call body:tag ! <instruction>+ call-tos

parameter eval p. 63
::= <instruction>+ asspar
::= <instruction>+ assrep n:byte

module_definition p. 67
::= module module_id:string check_code:string
visible_existing
body <local_quantity>* <program_element>* endmodule

visible_existing
::= <visible>* tag_list ! existing

visible p. 67
::= record_descriptor ! routine_profile
::= routine_specification ! label_specification
::= constant_specification ! insert_statement
::= info_setting

tag_list p. 67
::= < tag internal:tag external:number >+

interface_module p. 68
::= global module module_id:string checkcode:string

<global interface>* tag_list
body < init global:tag type repetition_value >*
endmodule

global_interface p. 68
::= record_descriptor
::= constant_definition < system sid:string >?
::= global_definition < system sid:string >?
::= routine_profile
::= info_setting

global_definition p. 68
::= global internal:newtag quantity_descriptor

macro_definition_module p. 70
::= macro module module_id:string checkcode:string
<macro_definition>* endmodule

macro_definition p. 71
::= macro macro_name:byte macro_parcount:byte
known macro_id:string
<macro_body_element>*
endmacro

macro_body_element p. 71
::= mark macro_sequence:string
::= mpar macro_parnumber:byte

macro_call p. 71
::= mcall macro_name:byte <actual_parameter:string>*

insert_statement p. 72
::= insert module_id:string check_code:string
external_id:string tagbase:newtag

taglimit:newtag

::= sysinsert module_id:string check_code:string

external_id:string tagbase:newtag

taglimit:newtag

area_initialisation p. 73

::= zeroarea

::= initarea resolved_type

::= dinitarea structured_type

eval_instruction p. 75

::= eval

delete_statement p. 75

::= delete from:tag

info_setting p. 76

::= info string

::= decl line:number

::= line line:number

::= stmt line:number

::= setswitch switch:byte setting:byte

byte

::= an (8-bit) unsigned integer value in the range 0..255.

number

::= a two-byte value greater than or equal to zero. A number (or any other multi-byte structure) will always be transmitted with the most significant byte first. Let the bytes be <B1><B2> in that sequence; the value will be 256 * B1 + B2 .

ordinal

::= a number with value greater than zero.

tag

::= An ordinal (the "tag-value") associated with a descriptor. See section 2.4.

::= The number zero followed by an ordinal (the "tag value") and an identifying string. This second form is intended for debugging purposes, and is used to associate an identification to the tag.

newtag

::= A tag-value with no association, see section 2.4.

spectag

::= a tag-value which, if not undefined, must have been given an association in a specification (labelspec, constspec, routinespec), see section 2.4.

index

::= a byte within an implementation-defined range which identifies an internal label. An important side effect of its occurrence is that it looses its meaning. Thus an internal label can be the destination of exactly one jump-instruction. See section 11.2.

newindex

::= an index-byte which is undefined; it becomes defined as an internal label through its occurrence. See section 11.2.

string

::= a byte with the value N followed by N "data bytes". The character count N must be greater than zero, thus a string cannot be empty (neither can a long_string, q.v.).

long_string

::= an ordinal with the value N followed by N "data bytes".

Appendix B

Encoding of basic symbols

0 - not used	1 record	2 not used	3 prefix
4 attr	5 not used	6 rep	7 alt
8 fixrep	9 endrecord		
10 c-record	11 text	12 c-char	13 c-int
14 c-size	15 c-real	16 c-lreal	17 c-aaddr

18 c-oaddr	19 c-gaddr	20 c-paddr	21 c-dot
22 c-raddr	23 nobody	24 anone	25 onone
26 gnone	27 nowhere	28 true	29 false
30 profile	31 known	32 system	33 external
34 import	35 export	36 exit	37 endprofile
38 routinespec	39 routine	40 local	41 endroutine
42 module	43 existing	44 tag	45 body
46 endmodule			
47 labelspec	48 label		
49 range			
50 global	51 init		
52 constspec	53 const		
54 delete			
55 fdest	56 bdest		
57 save	58 restore		
59 bseg	60 eseg		
61 skipif	62 endskip		
63 if	64 else	65 endif	66 not used
67 precall	68 asspar	69 assrep	70 call
71 fetch	72 refer	73 deref	74 select
75 remote	76 locate	77 index	78 inco
79 deco			
80 push	81 pushc	82 pushlen	83 dup
84 pop	85 empty		
86 setobj	87 getobj	88 access	
89 fjump	90 bjump	91 fjumpif	92 bjumpif
93 switch	94 goto		
95 t-inito	96 t-geto	97 t-seto	
98 add	99 sub	100 mult	101 div
102 rem	103 neg	104 and	105 or
106 xor	107 imp	108 eqv	109 not
110 dist			
111 assign	112 update		
113 convert	114 sysinsert		
115 insert			

116 zeroarea	117 initarea		
118 compare			
119 ?lt	120 ?le	121 ?eq	122 ?ge
123 ?gt	124 ?ne		
125 eval	126 info	127 line	128 setswitch
129 not used			
130 program	131 main	132 endprogram	
133 dsize	134 sdest	135 rupdate	136 asscall
137 call-tos	138 dinitarea	139 nosize	140 popall
141 repcall	142 interface		
143 macro	144 mark	145 mpar	
146 endmacro	147 mcall		
148 pushv	149 selectv	150 remotev	151 indexv
152 accessv			
153 decl	154 stmt		
155 - 255 - reserved.			

Predefined tags:

0 - not used	1 BOOL	2 CHAR	3 INT
4 SINT	5 REAL	6 LREAL	7 AADDR
8 OADDR	9 GADDR	10 PADDR	11 RADDR
12 SIZE			
13-31 reserved, i.e. first program-defined tag will be 32.			

Appendix C

Statutes for SG/PSS

STATUTES

for the

STANDARDS GROUP

for the Portabls SIMULA System.

Article 1. Definitions

The Portable SIMULA System (PSS) consists of a language dependent part and a target dependent part. The interface between these two parts is at any time defined in the documents:

S-PORT: Definition of S-code

S-PORT: The Environment Interface

The two documents comprise the PSS interface definition.

An S-compiler system is a system which is able to translate and execute programs represented according to the PSS interface definition.

Article 2. Objectives

The Standards Group for the Portable SIMULA System (SG/PSS) is an organisation which at all times shall:

- be the final arbiter in the interpretation of the PSS interface definition and be a center for custody of this formal definition.
- provide a forum for discussion and exchange of information relating to the PSS interface definition and its support.
- standardise the PSS interface definition and modify the definition when this is found necessary.

Article 3. Membership

Membership is open to organisations and firms responsible for the maintenance and support of an S-compiler system in active use. Any organisation may apply and be voted a member of the SG/PSS.

The Norwegian Computing Center (NCC), Oslo, Norway and the Edinburgh Regional Computing Center (ERCC), Edinburgh, Scotland are ex officio members of the SG/PSS. The NCC will also act as the secretariat of the SG/PSS.

The SG/PSS can offer membership to individuals, in recognition of their contribution to the SG/PSS work.

Once granted, an SG/PSS membership lasts until:

- it is resigned by the member, or
- it is revoked by the SG/PSS because the conditions under which it was granted cease to exist or the member acts against the objectives of the SG/PSS.

There is no membership fee for the SG/PSS. Members must cover their own expences.

Article 4. Representation, Voting and Meetings

Each member shall appoint one person to be his/her representative in the SG/PSS. The duration of appointment is determined by the members.

The SG/PSS shall meet once every year for an Annual Meeting. This meeting will, in addition to possible administrative matters, handle proposals related to the PSS interface definition. The Annual Meeting shall also elect one of the member's representatives as Chairman for the Standards Group.

Decisions can only be taken regarding matters on the agenda presented to the members at least 3 weeks before the meeting, unless all members present agree otherwise.

In addition to the Annual Meeting SG/PSS may have extraordinary meetings. Such meetings are held when the Chairman finds it necessary, or when this is approved by a majority of the members.

To constitute a quorum, all members of the SG/PSS shall be notified of the meeting and a majority of the members representatives shall be present or give their votes by mail, including the NCC and ERCC representatives.

Decisions by SG/PSS are made by a majority vote among the representatives taking part in the vote. Changes to the statutes for the SG/PSS or a decision to dissolve the SG/PSS require 4/5 majority. Any decision requires the consent of the NCC.

The SG/PSS meetings are open to non-representatives or non-members. Observers have no voting rights, and must apply to the Chairman of the SG/PSS for each meeting they wish to attend.

Article 5. Effectuation and revisions

These statutes were adopted at the Foundation Meeting of the Standards Group for the Portable SIMULA System in Edinburgh the 3rd March 1981 and come into effect immediately.

Formal rules of the SG/PSS operation

1. The main task of the SG/PSS is the maintenance of the PSS interface definition. Its work consists of:
    a. a clarification of obscure parts of the definition.
    b. removal of eventual conflicts in the definition.
    c. alteration of the definition when necessary.

2. The following types of changes in the definition can be directly considered by SG/PSS:
    a. obvious oversights that have occurred in the text of the definition.
    b. removal of language restrictions that are proved obsolete for consistency and implementation.
    c. trivial extensions to the existing concepts that are felt relevant for continued use of the definition in changing environments.
    d. any other changes as long as none of the members are against it.

3. Any other changes than those mentioned in point 2 above can also be considered. These changes must, however, be submitted to the secretariat two (2) months prior to the meeting. Submitted proposals will be distributed to the members without delay, to facilitate inclusion of relevant comments in the material presented with the meeting agenda at least three (3) weeks before said meeting.

4. All proposals for changes in the PSS interface definition conforming with the above rules must be formulated in writing in a concise manner and submitted to the secretariat. Anyone may submit such a proposal.

5. The Chairman of the SG/PSS is responsible for confirming receipt of each proposal, registering it and scheduling its processing at one of the meetings of the SG/PSS. Alternatively the Chairman may point out any inadequacies in a proposal to its submitter.

The proposal will be announced at the subsequent SG/PSS meeting which may approve or revoke the Chairman's decision related to this proposal.

Complete material related to a proposal will be submitted to the members when the proposal is processed at the next meeting or by specific request.

6. In its final form every proposal will be an updating text to one of the documents comprising the PSS interface definition. It will further indicate the original submitter, date of submission and its motivation. Alternative forms of the proposal and reasons for their rejection are a valuable part of the document. An example of a suitable form is attached to these rules.

The logical consistency of the text, its clarity and conciseness are of utmost importance. To this end the SG/PSS or its Chairman may allocate one particular member to bring the proposal into the required shape if this is deemed necessary.

7. It is the responsability of the Chairman to notify the submitter of a proposal about the result of its processing by the SG/PSS if this is not otherwise obvious. It is also his/her responsibility to minimize the time taken over each proposal.

8. The final text of the proposals will be available from the secratariat as a supplement to the PSS interface definition until they are incorporated into the definition at the next revision of the documents. To facilitate this process, at its approval a proposal will be assigned a number reflecting which of the documents it refers to and its chronological order.

Proposal for changing the Portable SIMULA System interface definition.

Document affected:

Submitter:

Date:

Title of proposal:

Affected section:		
Proposal:		
Motivation:		
SG/PSS decision on the above proposal		
For:	Against:	Abstained:

Index of Keywords