SPECIFICATION OF

S/SL: SYNTAX/SEMANTIC LANGUAGE

J.R. Cordy and R.C. Holt
December 1979
(Revised March 1980)

ABSTRACT

This document defines "new" S/SL, the second generation of the Syntax/Semantic Language. S/SL is a programming language developed at the Computer Systems Research Group, University of Toronto as a tool for constructing compilers. It has been used to implement scanners, parsers, semantic analyzers, storage allocators and machine code generators. S/SL has been used to implement compilers for Euclid, PT Pascal and Speckle, a PL/1 subset.

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and by Bell-Northern Research Limited.

INTRODUCTION

S/SL is a procedure-based variable-free programming language in which the program logic is stated using a small number of simple control constructs. It accesses data in terms of a set of operations organized into data-management modules called mechanisms. The interface to these mechanisms is defined in S/SL but their implementation is hidden from the S/SL program.

S/SL has one input stream and one output stream, each of which is strictly sequential. These streams are organized into "tokens" each of which is read and written as a unit. An auxiliary output stream for error diagnostics is also provided.

IDENTIFIERS, STRINGS AND INTEGERS

An S/SL identifier may consist of any string of up to 50 letters, digits and underscores (_) beginning with a letter. Upper and lower case letters are considered identical in S/SL, hence aa, aA, Aa and AA all represent the same identifier. **input**, **output**, **error**, **type**, **mechanism**, **rules**, **do**, **od**, **if**, **fi**, **end** and their various upper case forms are keywords of S/SL and must not be used as identifiers in an S/SL program.

An S/SL string is any sequence of characters not including a quote surrounded by quotes (').

Integers may be signed or unsigned and must lie within a range defined by the implementation. For example, this range could be -32767 to 32767 on the PDP-11.

Identifiers, keywords, strings and integers must not cross line boundaries. Identifiers, keywords and integers must not contain embedded blanks.

COMMENTS

A comment consists of the character "%" (which is not in a string) and the characters to the right of it on a source line.

CHARACTER SET

Since not all of the special characters used in S/SL are available on all machines, the following alternatives to special characters are allowed.

"!"	for	"|"
"**do**"	for	"{"
"**od**"	for	"}"
"**if**"	for	"["
"**fi**"	for	"]"

SOURCE PROGRAM FORMAT

S/SL programs are free format; that is, the identifiers, keywords, strings, integers and special characters which make up an S/SL program may be separated by any number of blanks, tab characters, form feeds and source line boundaries.

NOTATION

The following sections define the syntax of S/SL. Throughout the following, {item} means zero or more of the item, and [item] means the item is optional. The abbreviation "id" is used for identifier.

PROGRAMS

An S/SL program consists of a set of definitions followed by a set of rules.

A <u>program</u> is:

```
[inputDefinition]
[outputDefinition]
[inputOutputDefinition]
[errorDefinition]
{typeOrMechanismDefinition}
rules
     {rule}
end
```

INPUT AND OUTPUT DEFINITIONS

An <u>inputDefinition</u> is:

```
input ":"
    {tokenDefinition} ";"
```

An <u>outputDefinition</u> is:

```
output ":"
    {tokenDefinition} ";"
```

An <u>inputOutputDefinition</u> is:

```
input output ":"
    {tokenDefinition} ";"
```

A <u>tokenDefinition</u> is:

```
id [string] ["=" tokenValue]
```

The inputDefinition section defines the input tokens to the S/SL program. The outputDefinition section defines the output tokens of the program. The inputOutputDefinition section defines those tokens which are both input tokens and output tokens of the program. Tokens already defined in the inputDefinition or outputDefinition sections must not be redefined in the inputOutputDefinition section.

The optional string which may be given in a tokenDefinition is a synonym for the token identifier and can be used in place of the identifier anywhere in the S/SL program.

Each input and output token is assigned an integer value for use in the implementation of the S/SL program. This value may be optionally specified in each tokenDefinition. The tokenValue may be specified as an integer or as the value of any previously defined identifier or string. If omitted, the value assigned to the token is the value associated with the previous token in the class plus one. The default value associated with the first input token and the first output token is zero. The default value associated with the first input-output token is the maximum of the last token defined in the inputDefinition section and the last token defined in the outputDefinition section. In this way the default input-output token values are unique with respect to both input tokens and output tokens.

ERROR SIGNALS

An errorDefinition is:

```
          error ":"
              {errorSignalDefinition} ";"
```

An errorSignalDefinition is:

```
          id ["=" errorValue]
```

Each errorSignalDefinition defines an error signal which can be signalled by the S/SL program. An integer error code value is associated with each errorId for use in the implementation of the S/SL program. This value may be optionally specified in each errorSignalDefinition. The errorValue may be specified as an integer or as the value of any previously defined identifier or string. The default value associated with an error signal is the value associated with the previous error signal plus one. The default value for the first error signal is 10 (errors 0 to 9 are reserved for S/SL system use).

TYPE AND MECHANISM DEFINITIONS

Type and mechanism definitions may be grouped and intermixed to reflect the association of types and the operation definitions which use them.

A typeOrMechanismDefinition is one of:

```
a.  typeDefinition
b.  mechanismDefinition
```

TYPES

A typeDefinition is:

```
type id ":"
    {valueDefinition} ";"
```

A valueDefinition is:

```
id ["=" value]
```

Each typeDefinition defines a type of values for use as the parameter or result type of a semantic operation or as the result type of a rule.

Each valueDefinition defines a valueId in the type. An integer value is associated with each valueId for use in the implementation of the S/SL program. This value may be optionally specified in each valueDefinition. The value may be specified as an integer or as the value of any previously defined identifier or string. The default value assigned to a value identifier is the value associated with the previous value identifier plus one. The default value associated with the first valueDefinition in a type is zero.

MECHANISMS

A mechanismDefinition is:

```
mechanism id ":"
    {operationDefinition} ";"
```

Each mechanismDefinition defines the set of semantic operations associated with a semantic mechanism. The mechanismId itself is unused in the S/SL program. However, operation identifiers associated with a mechanism are by convention expected to begin with the mechanism identifier.

An operationDefinition is one of:

```
a.  id
b.  id "(" typeId ")"
c.  id ">>" typeId
d.  id "(" typeId ")" ">>" typeId
```

Each operationDefinition defines a semantic operation associated with the mechanism.

Form (a) defines an update semantic operation, which causes an update to the semantic data structure.

Form (b) defines a parameterized update operation, which uses the parameter value to update the semantic data structure. The typeId gives the type of the parameter and can be any previously defined type.

Form (c) defines a choice semantic operation, which returns a result value obtained from the current state of the semantic mechanism, which is used as the selector in a semantic choice. The typeId gives the type of the result and can be any previously defined type.

Form (d) defines a parameterized choice operation. The first typeId gives the parameter type, the second the result type. Each can be any previously defined type.

Choice operations (forms (c) and (d) above) may be invoked only as the selector in a semantic choice.

RULES

A _rule_ is one of:

```
a.  id ":"
        {action} ";"

b.  id ">>" typeId ":"
        {action} ";"
```

The rules define the subroutines and functions of the S/SL program. Rules may call one another recursively. A rule need not be defined before it is used. Execution of the program begins with the first rule.

Form (a) defines a _procedure_ rule which can be invoked using a call action.

Form (b) defines a _choice_ rule which returns a result value of the specified type. The typeId can be any previously defined type. Choice rules may only be invoked as the selector in a rule choice.

ACTIONS

An _action_ is one of the following:

```
a.  inputToken
b.  "." outputToken
```

```
c.  "#" errorId
d.  "{"
        {action}
    "}"
e.  ">"
f.  "["
        {"|" inputToken {"," inputToken} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
g.  "@" procedureRuleId
h.  ">>"
i.  "[" "@" choiceRuleId
        {"|" valueId {"," valueId} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
j.  ">>" valueId
k.  updateOpId
l.  parameterizedUpdateOpId "(" valueId ")"
m.  "[" choiceOpId
        {"|" valueId {"," valueId} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
n.  "[" parameterizedChoiceOpId "(" valueId ")"
        {"|" valueId {"," valueId} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
```

Form (a) is an input action. The next input token is to be accepted from the input stream. If it is not the one specified, a syntax error is flagged. The inputToken may be an inputTokenId, an inputOutputTokenId, an inputTokenString, an inputOutputTokenString, or a question mark (?). The question mark is a special token which matches any input token.

Form (b) denotes emission of an output token to the output stream. The outputToken may be an outputTokenId, an inputOutputTokenId, an outputTokenString or an inputOutputTokenString.

Form (c) denotes the emission of the specified error signal to the error stream.

Form (d) is a cycle or loop. Execution of the actions inside the cycle is repeated until one of its cycle exits (form (e)) or a return (forms (h) and (j)) is encountered. A cycle exit causes execution to continue following the nearest enclosing cycle. The cycle exit action is not allowed outside of a cycle.

Form (f) is an input token choice. The next token in the input stream is examined and execution continues with the first action in the alternative labelled with that input token. The matched input token is accepted from the input stream.

Each inputToken label can be an inputTokenId, an inputOutputTokenId, an inputTokenString or an inputOutputTokenString. A label can not be repeated nor appear on more than one alternative.

The alternative labelled with an "*" is the otherwise alternative. If the next input token does not match any of the alternative labels of the choice, execution continues with the first action in the otherwise alternative. If the otherwise alternative is taken, the input token is not accepted from the input stream, but remains as the next input token. After execution of the last action in an alternative of the choice, execution continues following the choice.

If the next input token does not match any of the alternative labels and no otherwise alternative is present, a syntax error is flagged. For parsers written in S/SL, the default error handling strategy is to repeat the choice after modifying the input stream such that the next input token matches the first alternative. For compiler phases other than parsers, continued execution is undefined (the implementation aborts).

Form (g) is a call to a procedure rule. Execution continues at the first action in the specified rule. When execution of the called rule is completed, either by executing the last action in the rule or by encountering a return action (form (h)), execution is resumed following the call.

Form (h) is a return action. It causes a return from the procedure rule in which it appears. A procedure rule may return explicitly by executing a return action or implicitly by reaching the end of the rule. A procedure rule must not contain a valued return (form (j)).

Form (i) is a rule choice. The specified choice rule is called and returns a value by executing a valued return action (form (j)). The returned value is used to make a choice similar to an input token choice (form (f) above). Execution continues with the first action of the alternative whose label matches the returned value. If none of the alternative labels matches the value, the otherwise alternative is taken. Following execution of the last action in the chosen alternative, execution continues following the choice.

Each alternative label in a rule choice must be a value of the result type of the choice rule. A label can not be repeated nor appear on more than one alternative.

Form (j) is a valued return action. The specified value is returned as the result of the choice rule in which the action appears. The value must be of the result type of the choice rule. A choice rule may return only by executing a valued return action. A choice rule must not return implicitly by reaching the end of the rule. It must not contain a non-valued return (form (h)).

Form (k) is the invocation of an update semantic operation. Similarly, form (l) is the invocation of a parameterized update operation. The parameter value, which must be of the operation's parameter type, is supplied to the invocation of the operation.

Form (m) is a semantic choice. The specified choice semantic operation is invoked and the returned value used to make a choice similar to an input token choice (form (f) above). Execution continues with the first action of the alternative whose label matches the returned value. If none of the alternative labels matches the value, the otherwise alternative is taken. Following execution of the last action in the chosen alternative, execution continues following the choice. Similarly, form (n) is a parameterized semantic choice. The parameter value, which must be of the operation's parameter type, is provided to the invocation of the choice operation.

Each alternative label in a semantic choice must be a value of the result type of the choice operation. A label can not be repeated nor appear on more than one alternative.

If the returned value in a rule choice or semantic choice does not match any of the alternative labels and no otherwise alternative is present, continued execution is undefined (the implementation aborts).

THE SYNTAX OF S/SL

A <u>program</u> is:

```
[inputDefinition]
[outputDefinition]
[inputOutputDefinition]
[errorDefinition]
{typeOrMechanismDefinition}
rules
  {rule}
end
```

An <u>inputDefinition</u> is:

```
input ":"
    {tokenDefinition} ";"
```

An <u>outputDefinition</u> is:

```
output ":"
    {tokenDefinition} ";"
```

An inputOutputDefinition is:

```
input output ":"
    {tokenDefinition} ";"
```

A tokenDefinition is:

```
id [string] ["=" tokenValue]
```

An errorDefinition is:

```
error ":"
    {errorSignalDefinition} ";"
```

An errorSignalDefinition is:

```
id ["=" errorValue]
```

A typeOrMechanismDefinition is one of:

```
a.  typeDefinition
b.  mechanismDefinition
```

A typeDefinition is:

```
type id ":"
    {valueDefinition} ";"
```

A valueDefinition is:

```
id ["=" value]
```

A mechanismDefinition is:

```
mechanism id ":"
    {operationDefinition} ";"
```

A rule is one of:

```
a.  id ":"
        {action} ";"

b.  id ">>" typeId ":"
        {action} ";"
```

An <u>action</u> is one of the following:

```
a.  inputToken
b.  "." outputToken
c.  "#" errorId
d.  "{"
        {action}
    "}"
e.  ">"
f.  "["
        {"|" inputToken {"," inputToken} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
g.  "@" procedureRuleId
h.  ">>"
i.  "[" "@" choiceRuleId
        {"|" valueId {"," valueId} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
j.  ">>" valueId
k.  updateOpId
l.  parameterizedUpdateOpId "(" valueId ")"
m.  "[" choiceOpId
        {"|" valueId {"," valueId} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
n.  "[" parameterizedChoiceOpId "(" valueId ")"
        {"|" valueId {"," valueId} ":"
            {action} }
        ["|" "*" ":"
            {action} ]
    "]"
```