S/SL: SYNTAX/SEMANTIC LANGUAGE

INTRODUCTION AND SPECIFICATION

Richard C. Holt
James R. Cordy
David B. Wortman

Technical Report CSRG-118
September 1980

Computer Systems Research Group
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the

University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

FOREWORD

This report on S/SL represents several years of research at CSRG on compiler development techniques. S/SL has been tried under considerable stress, in the development of the Toronto Euclid compiler, and has been found to be a very effective software tool.

An accompanying technical report by Rosselet, CSRG-119, presents a Pascal subset (PT) compiler developed using S/SL. Persons interested in using S/SL for production work are encouraged to study Rosselet's report and his compiler. His compiler is written in a highly readable style in the PT subset of standard Pascal, using S/SL as the technique to drive each of three passes.

This report consists of two separate papers. The first is an introduction to the S/SL language and technique. The second is a detailed specification of the S/SL language.

R.C. Holt

AN INTRODUCTION TO

S/SL: SYNTAX/SEMANTIC LANGUAGE

R.C. Holt, J.R. Cordy and D.B. Wortman
December 1979
(Revised June 1980)

Computer Systems Research Group
University of Toronto

Copyright (C) 1979, 1980 by the University of Toronto.
This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and by Bell-Northern Research Limited.

ABSTRACT

S/SL (Syntax/Semantic Language) is a language that was developed for implementing compilers. A subset called SL (Syntax Language) has the same recognition power as LR(k) parsers. Complete S/SL includes invocation of semantic operations implemented in another language such as Pascal.

S/SL implies a topdown programming methodology. First, a data-free algorithm is developed in S/SL. The algorithm invokes operations on "semantic mechanisms". A semantic mechanism is an abstract object, specified, from the point of view of the S/SL, only by the effect of operations upon the object. Later, the mechanisms are implemented apart from the S/SL program. The separation of the algorithm from the data, and the division of data into mechanisms simplifies the effort needed to understand and maintain the resulting software.

S/SL has been used to construct compilers for Speckle (a PL/1 subset), PT (a Pascal subset) and Toronto Euclid. It has been used to implement scanners, parsers, semantic analyzers and code generators.

S/SL programs are implemented by translating them to tables of integers. A "table walker" program executes the S/SL program by interpreting this table. The translation of S/SL programs to tables is performed by a program called the S/SL processor. This processor serves a function analogous to an LR(k) parser generator.

The implementation of S/SL is quite simple and highly portable. It is available in a very small subset of Pascal that can easily be transliterated into other high level languages.

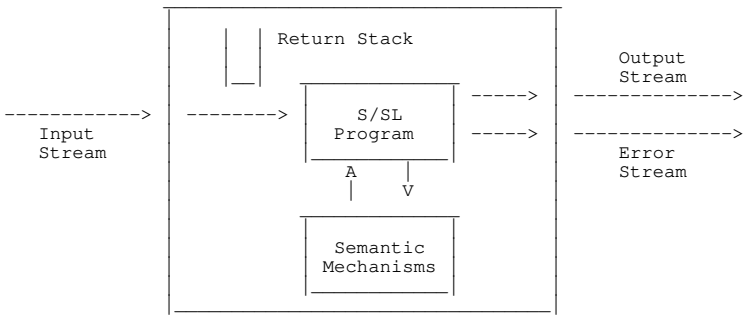
INTRODUCTION

S/SL is a programming language developed at the University of Toronto as a tool for constructing compilers. It is a very modest language, incorporating only the following features: sequences, repetitions and selections of actions (statements); input, matching and output of tokens; output of error signals; subprograms (called rules); and invocation of semantic operations implemented in a base language such as Pascal. S/SL is a language without data or assignment; it is a pure control language [Cordy 1980]. Data can be manipulated only via semantic operations.

This paper is organized as follows. We give a computational model for S/SL and a summary of features of S/SL. Then, example S/SL programs are given to illustrate the use of S/SL in writing scanners, parsers, semantic analyzers and code generators. We discuss the methodology of software development used with S/SL. Next, we give the relation of S/SL to the theory of formal languages and automata. This is followed by a discussion of the implementation of S/SL. Finally, we give observations concerning the use of S/SL.

I. A COMPUTATIONAL MODEL

S/SL assumes the computational model illustrated in the following diagram. There is an input stream that consists of tokens. Each token is a member of a finite set, such as the set of ASCII characters or the set of lexical constructs in Pascal (including identifiers, integers, keywords and operators). The S/SL program reads (accepts) tokens one-by-one from the input stream and emits tokens to an output stream. It can also emit error signals to an error stream. Error signals are analogous to tokens, but we do not call them tokens to avoid confusion with the input and output streams. In most applications of S/SL, each emitted error signal is transformed into an error message.



Computational Model for S/SL

The S/SL program transduces (translates) its input stream into an output stream. For example, a parser written in S/SL reads a stream of tokens produced by a scanner and generates an output stream to be consumed by the semantic analysis pass of the compiler. Likewise, a scanner written in S/SL reads a stream of characters (its input tokens) and emits a stream of tokens to be consumed by the parser.

An S/SL program is a set of possibly recursive rules (subprograms). To support this recursion, the implementation uses an implicit stack of return addresses, to allow a called rule to return to the appropriate calling point. The return stack is of little interest except that in terms of automata theory it corresponds to the stack of a pushdown automaton.

Besides controlling the input and output streams, the S/SL program can manipulate data using semantic operations that are organized into modules called semantic mechanisms. The interface to each semantic mechanism is defined in S/SL, but the implementation is hidden. The implementation is done separately in a base language such as Pascal. The S/SL program invokes semantic operations to inspect or manipulate data, but has no other access to data. It has no direct data handling capabilities such as assignment or comparison.

The symbol table is the most important semantic mechanism in a typical compiler. In building a semantic analysis pass, one would define this semantic mechanism by specifying operations such as the following:

Enter a symbol into the symbol table
Look up a symbol in the symbol table
Start a new scope in the symbol table
Finish a scope in the symbol table

The S/SL programmer need not be directly concerned with the implementation of the operations. In writing his S/SL program he needs to know only their meaning (specification). This is analogous to specifying an abstract data type and using it without being concerned with its implementation.

In certain instances a semantic mechanism (or at least its data) may be preserved beyond the termination of a particular S/SL execution. For example, one pass of the Toronto Euclid compiler [Holt 1978, CSRG 1980] creates a symbol/type table that is used by successive passes.

A parser pass of a compiler may not require semantic mechanisms, because S/SL without semantic mechanisms is powerful enough to do syntax checking.

II. FEATURES OF S/SL

First we will describe the features of SL, which is S/SL without semantic mechanisms. Each SL program consists of a list of executable rules. Each rule consists of a name, an optional return type and a sequence of actions (statements). Each rule has one of these two forms:

```
name: actions;

name >> type: actions;
```

A rule with the first form is called a procedure rule; a rule with the second form is called a choice rule. These two forms are analogous to procedures and functions in Pascal. Execution begins with the first rule of the S/SL program, which must be a procedure rule. A rule returns when it reaches its end (;) or it encounters a return action (written >>). A choice rule returns a value in its specified type (a range), which is tested in a choice action (see below).

Summary of SL Actions

There are only eight different actions (statements) in SL. These are now described.

1. The input (or match) action. The appearance of an input token in a rule signifies that the next input should be read and must match the specified token. For example, the following specifies that the next item in the input stream is to be read and it must be an integer token:

```
integer
```

The appearance of a question mark in a rule signifies that the next token (whatever it is) is to be read. The ? matches any token.

2. The emit action. The appearance of a dot (a period) followed by an output token signifies that the token is to be emitted to the output stream, for example:

```
.add
```

causes an add token to be emitted to the stream.

3. The error action. The appearance of the symbol # followed by an error signal signifies that the error signal is to be emitted to a special error stream, for example:


```
#missingSemicolon
```

The signal called missingSemicolon is output to the error stream; presumably this stream is used to print error messages.

4. The cycle action. Each cycle is of the form:

```
{
   actions
}
```

The enclosed actions are repeated until a return (>>) or one of the cycle's exits (>) is encountered.

5. The exit construct. The appearance of the symbol > signifies exit from the most tightly enclosing cycle.

6. The choice action. The choice action is of the form:

```
[ selector
   | labels:
     actions
   | labels:
     actions
     ...
   | *:
     actions
]
```

The selector is optional. If it is omitted, we have an input choice, which tries to match the next input token to one of the labels. The selector can also be of the form @identifier where "identifier" is the name of a choice rule; this gives us a rule choice. In a rule choice, the specified rule is called and then the choice tries to match the returned value to one of the labels. There is also a semantic choice, in which the selector is the name of a semantic operation; semantic choices are in S/SL, but not in SL. The actions associated with the matched label are executed; if no label is matched, the final alternative, labelled by the star (*), is executed.

The otherwise clause:

```
| *:
  actions
```

is optional. If omitted, the selector must match one of the choice's labels (otherwise there is an error). In an input choice, the matching of the input token to a label has the side-effect of reading another token; if the token is not matched and the otherwise clause is selected, then reading

is not done.

Each alternative in a choice can have several labels, separated by commas; for example, this alternative has as labels the tokens plus and minus:

```
| '+', '-':
   integer
```

Each label in a given choice construct must be of the same type as the selector, and each label of the choice must be a distinct value. For input choices, the labels must be input tokens. For rule choices, the labels must be of the type returned by the rule, for example

```
[ @OptionalExpression
   | true:
      actions
   | false:
      actions
]
```

where the range of OptionalExpression is Boolean.

When writing a parser it is common to use an input choice that accepts a particular limited set of tokens, with no otherwise clause. We do this in spite of the fact that a syntax error may cause the next input token to fall outside this set. When the next token is not acceptable, we rely on error handling logic to either abort the SL program or to repair the input stream to be acceptable by the input choice. The usual repair strategy is to take the first choice in case of such an error, and to modify the input stream accordingly.

7. The call action. The appearance of the symbol @ followed by the name of a procedure rule signifies that the rule is to be called. For example, here is a call to the Expression rule:

```
@Expression
```

8. The return action. The symbol >> in a rule signifies return before reaching the end of the rule. In choice rules, the >> must be followed by a value in the rule's specified return type, and implicit return via the final semicolon is not allowed. In procedure rules, the return >> cannot be followed by a value. Usually >> is not used in procedures because return is implicit at the end of the rule. Here is a choice rule, OptionalExpression, that returns true if the next token is the end-of-file; otherwise it parses an expression and returns false.


```
OptionalExpression >> Boolean:
   [
      | eof:
        >> true
      | *:
        @Expression
        >> false
   ];
```

This completes a summary of the actions in SL. The actions in S/SL are the same with the addition of calls to semantic operations.

Definitions in SL

We need to make certain definitions that are used by SL rules. In particular we need to specify the set of input tokens, the set of output tokens, the set of error signals and the set of values returned by each choice rule (and by each semantic operation). Each of these sets is defined by enumerating the names of their members; this is similar to defining enumerated types in Pascal. For example:

```
input:              % Input tokens
   integer
   plus   '+'
   minus  '-'
   assign ':='
   ...etc...;
output:             % Output tokens
   Int
   Add
   Subt
   ...etc...;
error:              % Error signals
   missingSemicolon
   ...etc...;
type Boolean:       % User defined set of values
   false
   true;
```

Here we have also shown the convention for comments in S/SL programs, namely, anything to the right of a % on a line is ignored.

Each token and error signal must be given a name, such as integer or Subt. The input and output tokens can also be given a string name, for example, the plus token also has the name '+'. Both names can be used in the S/SL program.

In the case of '+' the name "plus" seems to be useless, as it is defined but need not be used in the S/SL. The reason that identifiers such as plus are required is that they are used in the implementation of S/SL. For example, if the implementation is done using Pascal, the "plus" will be declared as a Pascal named constant whose value is the token's number.

The S/SL programmer can specify the internal value of each token, in order to be compatible with an external interface, for example,

```
plus '+' = 21
```

This assigns the internal token number 21 to "plus".

Sometimes it is convenient to emit the same tokens that are read. To allow this, a special class called "input output" is allowed, for example:

```
input output:           % Used for both input and output
   integer
   plus  '+'
   minus '-'
   ..etc...;
```

Any token listed under "input output" is included in both the input and output types.

Semantic Mechanisms

SL is sufficient for implementing parsers for languages having LR(k) grammars, such as Pascal and Algol-60. However, SL by itself is not sufficient for implementing more complex phases of a compiler, such as semantic analysis. It does not provide for data structures, such as symbol tables, which are required by these phases. SL as augmented by semantic operations becomes S/SL. The S/SL programmer groups the operations that access a particular data structure; these operations together with the data structure are called a semantic mechanism.

As a simple example of a semantic mechanism, let us consider a stack of counters. As is shown in the next section, this mechanism can be used for checking the number of actual parameters of a Pascal procedure call. The operations that modify and update the stack are defined in S/SL, but the implementation is carried out later in another language. Here is the definition in S/SL of a stack of counters:

```
mechanism Count:
  CountPush(number)        % New counter gets specified value
  CountPushSymbolDimension % New counter gets value from symbol
                           %    table
  CountPop                 % Delete counter
  CountIncrement           % Add 1 to top counter
  CountDecrement           % Subtract 1 from top counter
  CountChoose >> number;   % Inspect top counter
```

Following the keyword "mechanism" is the name of the mechanism (Count). This name has no significance except for documentation. After the colon comes the list of semantic operations for the mechanism.

By convention, the name of each operation on a semantic mechanism begins with the mechanism's name. For example, BufferSave is an operation on the Buffer mechanism, while

CountPop is an operation on the Count mechanism.

We have defined six semantic operations for the count stack. the first five are update operations; they are defined without a return type (without >>). Their purpose is to modify the semantic mechanism. The last operation, CountChoose, is a choice operation. A choice operation returns a value that is used in an S/SL choice action.

By convention a choice operation returns information about its semantic mechanism but does not modify it. Also by convention, operations on one semantic mechanism may inspect but not modify other semantic mechanisms. These conventions depend on programmer discipline and are not enforced automatically.

The first operation, CountPush, puts a new value on the count stack; for example, CountPush(zero) pushes zero onto the stack. Since CountPush takes a parameter, it is called a parameterized operation. A parameter of a parameterized operation must be a constant in a previously defined set of values in the S/SL program. Essentially a parameterized operation is a class of nonparameterized operations, one for each constant in the type.

The last semantic operation, CountChoose, is called to find the top value on the count stack. The notation ">> number" means it returns a value from the type named "number". A choice operation can be used only in S/SL choice actions, for example, see line 12 of the following example rule.

An Example Using the Count Stack

We now give an example S/SL rule that uses the count stack. This example rule handles a list of actual parameters for a procedure call in a language such as Pascal.

```
 1   ActualParameterList:
 2      CountPushSymbolDimension   % Number of formal parameters
 3      {
 4         @HandleActualParameter
 5         CountDecrement
 6         [
 7            | ')':
 8              >
 9            | ',':
10         ]
11      }
12      [ CountChoose
13         | zero:
14         | *:
```



```
15            #WrongNumberParameters
16      ]
17      CountPop;
```

Suppose the Pascal procedure call is

```
P(exp1,exp2);
```

The procedure's name P is accepted by the S/SL program and becomes the current symbol of interest. Then the left parenthesis is accepted and finally the ActualParameterList rule is called. Line 2 finds the declared number of parameters of P and pushes this number onto the count stack. Then the loop (lines 3 through 11) processes the list of actuals, decrementing the count for each actual. HandleActualParameter will accept "exp1" the first time through the loop and "exp2" the second time. Then lines 12 through 16 print an error message if the wrong number of actual parameters was supplied. Line 17 pops the counter that was pushed on line 2.

In Pascal an actual parameter may contain function calls, so the ActualParameterList rule may be re-entered recursively. Each recursion needs a new counter, and the count stack is used to store these counters.

This example has illustrated how to define and use semantic operations. Each operation performs an update or a choice and may be parameterized. We have given an example of a parameterless choice operation (CountChoose) but not a parameterized choice operation, which is defined using this form:

```
name(parameterType) >> returnType
```

A parameterized choice operation is similar to a parameterless choice operation except it receives a constant as a parameter.

We have shown how to define the names of semantic operations as well as their parameter and return types, but we have not shown how to give their implementations. Before going into the details of these, we will give several more example S/SL programs.

III. EXAMPLES OF S/SL USE

The following examples illustrate the use of S/SL in implementing a compiler. We start with scanning and proceed eventually to code generation for a PDP-11.

Scanning

The purpose of our example scanner is to collect the characters of a source program into syntax tokens. Leading blanks are skipped and the characters of the syntax token are collected by a semantic operation named BufferSave. We will assume that an input filter for our scanner has mapped the characters A through Z to a super character called "letter" and the characters 0 through 9 to "digit".

This example is a complete S/SL program that can be submitted to an S/SL processor. The first part consists of definitions. The second part, between the words "rules" and "end", gives the S/SL rules.

```
     % A scanner for identifiers, integers, ';', '+' and '-'

     input:
        letter
        digit
        blank
        illegalChar;

     output:
        identifier
        integer;

     input output:
        semicolon  ';'
        plus       '+'
        minus      '-';

     error:
        badChar;

     mechanism Buffer:
        BufferSave;  % Save last accepted character

     rules

     Scanner:
        @SkipNoise
        [
           | letter:
             BufferSave
```



```
             {[
                | letter,digit:
                  BufferSave
                | *:
                  .identifier
                  >
             ]}
           | digit:
             BufferSave
             {[
                | digit:
                  BufferSave
                | *:
                  .integer
                  >
             ]}
           | ';':
             .semicolon
           | '+':
             .plus
           | '-':
             .minus
        ];

     SkipNoise:
        {[
           | blank:
           | illegalChar:
             #badChar
           | *:
             >
        ]};

     end
```

This scanner is a simplified version of the one used in the PT Pascal compiler [Rosselet 1980]. This example uses the notation {[and]}. This is not a new construct, but simply a choice action nested directly inside a cycle.

Parsing

It is straightforward to write a parser for a language such as Pascal in S/SL. The Pascal report [Jensen 1974] contains a specification of the grammar of Pascal in the form of syntax charts. These charts are easily transliterated to S/SL, to produce a grammar for Pascal in S/SL. We call this an algorithmic grammar, because it can be directly executed to accept a string in the specified language. We will give three examples of parsing: recognizing a statement, handling an "if" statement and producing postfix for expressions.

Recognizing statements. A Pascal statement can be recognized by the following rule.

```
     Statement:
        [
           | identifier:
             @AssignmentOrCallStatement
           | 'if':
             @IfStatement
           | 'case':
             @CaseStatement
           | 'while':
             @WhileStatement
           | 'repeat':
             @RepeatStatement
           | 'for':
             @ForStatement
           | 'with':
             @WithStatement
           | 'begin':
             @BeginStatement
           | 'goto':
             @GotoStatement
           | *:
             % null statement
        ];
```

Each individual statement such as 'if' is handled by its own rule. Unfortunately for the parser, there is a local ambiguity in Pascal in that a statement beginning with an identifier can be either an assignment or a procedure call, so one rule must handle both.

Handling "if" statements. As an example of a rules for individual statements, we will give a rule to handle "if".

```
     IfStatement:    % Just accepted 'if' token
        @Expression 'then' @Statement
        [
           | 'else':
             @Statement
           | *:
        ];
```

This rule recursively calls the Statement rule to handle 'then' and 'else' clauses. The "dangling else" problem is easily solved, as the choice construct immediately accepts the adjacent else clause if present. We have not shown output operations or calls to semantic operations; these can be inserted to make this example rule useful in a compiler.

Expressions and postfix. We now show how expressions can be parsed. To keep the example simple, we restrict our

attention to expressions that consist of identifiers, the binary operators +, -, * and /, and nesting via parentheses. We assume that expressions are evaluated left to right except that * and / have higher precedence than + and - and parenthesized subexpressions are evaluated before use. The example S/SL will transduce infix to postfix, for example, the input stream

```
                          A + B * C
```

is output as

```
                          A B C multiply add
```

Here are rules that handle expressions:

```
     Expression:
        @Factor
        {[
           | '+':
             @Factor .add
           | '-':
             @Factor .subtract
           | *:
             >
        ]};

     Factor:
        @Primary
        {[
           | '*':
             @Primary .multiply
           | '/':
             @Primary .divide
           | *:
             >
        ]};

     Primary:
        [
           | '(':
             @Expression ')'
           | identifier:
             .identifier EmitIdentifierText
        ];
```

The Expresson rule calls the Factor rule to handle all high precedence operations and subexpressions before handling addition and subtraction. Similarly, Factor calls Primary before handling multiplication and division. Primary calls Expression recursively to handle nested expressions. Since each identifier token has a value (such as "A" or "B"), we have used the semantic operation EmitIdentifierText to place

this value in the output stream.

Semantic Analysis

We take as a typical semantic analysis task the problem of checking types in expressions. We can do this using a semantic mechanism called the type stack. It mimics the action of a runtime stack used in evaluating expressions. Each entry in the type stack gives the type of the corresponding operand in the runtime stack. The definition in S/SL of this semantic mechanism would be similar to the definition we gave for the stack of counters. We will assume that the input to our semantic analysis pass has its expressions in syntactically correct postfix. This rule is used to accept expressions:

```
PostfixExpression:
   {
      @Primaries @Operators
      [
         | exprEnd:
           >
         | *:
      ]
   };
```

We have assumed that the token exprEnd marks the end of each expression. To simplify our example, we consider only these operators: addition, equality and intersection (and). As in Pascal, we assume that only numbers can be added, only Booleans can be intersected, numbers can be compared to numbers only and Booleans to Booleans only. We assume that all numbers are integers.

As each primary is accepted, its type is pushed onto the type stack. As each operator is accepted, the types of its operands are checked and their types on the type stack are replaced by the operator's result type.

```
Primaries:
   {[
      | constant: TypePushConstant
      ...other primaries...
      | *:
        >
   ]};

Operators:
   {[
      | add:
        @CheckInteger @CheckInteger TypePush(int)
      | and:
        @CheckBoolean @CheckBoolean TypePush(bool)
      | equal:
        @CheckEquality TypePush(bool)
      | *:
        >
   ]};

CheckInteger:
   [ TypeChoose
      | int:
      | *:
        #integerRequired
   ]
   TypePop;

CheckBoolean:
   [ TypeChoose
      | bool:
      | *:
        #booleanRequired
   ]
   TypePop;

CheckEquality:
   [ TypeChoose
      | int:
        TypePop @CheckInteger
      | bool:
        TypePop @CheckBoolean
   ];
```

We could easily enhance these rules to produce pseudo-machine code (P-code) while they are checking types. We could incorporate this type checking into the pass that does syntax checking (the parser).

Code Generation

The code generator pass of the Toronto Euclid compiler accepts expressions in postfix and generates PDP-11 assembly language. It does extensive local optimization, to take advantage of the PDP-11 order code. To illustrate, we show a simplified version of the rule that generates code to add the right operand to the left. We will assume that previous analysis by S/SL in this pass has discovered that the source statement is of the form

```
x := x + y
```

and now y is the right operand and x is the left. A semantic mechanism called the symbol stack holds these operands, with the right operand as its top element and the left

operand as its second element. The rule pops the top (right) element from the symbol stack, leaving the left (new top) element to represent the result.

```
     AddRightToLeft:
        [ SymbolIsManifestValue  % Is right operand a constant?
            | yes:
              [ SymbolChooseManifestValue
                  | one:
                    SymbolPop GenerateSingle(inc)
                  | zero:
                    SymbolPop % Generate nothing
                  | minusOne:
                    SymbolPop GenerateSingle(dec)
                  | *:
                    GenerateDouble(add) SymbolPop
              ]
            | no:
              [ SymbolIsLeftSameRight  % Adding x to x?
                  | yes:
                    SymbolPop GenerateSingle(asl)
                  | no:
                    GenerateDouble(add) SymbolPop
              ]
        ];
```

This S/SL rule selectively generates the following PDP-11 code:

Right Operand	PDP-11 code generated
1	inc left
0	(no code)
-1	dec left
same as left operand	asl left
otherwise	add right,left

The parameterized semantic operation GenerateSingle emits a PDP-11 single operand instruction such as inc (increment), dec (decrement) or asl (arithmetic shift left). Similarly, GenerateDouble generates double operand instructions such as add.

Readability and Special Characters

The preceding set of examples is intended to demonstrate the power, convenience and expressiveness of S/SL. In practice, S/SL has been found to be quite readable and maintainable. A question that arises is why special characters rather than keywords are used to denote control constructs. People who are used to programming in Pascal-like languages are surprised to find, for example, that a choice action is written as [...|...|...] instead of case...of...end. While

no objective explanation is possible, the following observations are put forth.

Before developing S/SL, the authors used syntax and semantic charts [Barnard 1975, Cordy 1976, Cordy 1979]; these charts were hand translated into an assembly-like notation that used keywords. It was observed that this latter notation was considerably bulkier and clumsier than the charts. This led to experimentation with various notations, especially those used for regular expressions.

It was discovered that essentially all the readability of charts could be maintained using S/SL, and besides, S/SL can be directly processed by a computer. This degree of readability depends on (1) consistent indentation of choices and cycles so these constructs are visually obvious and (2) sufficient exposure of the reader to the S/SL notation, so that he immediately associates [...] with selection and {...} with repetition.

In hindsight, it appears that special characters are suitable for S/SL because it is such a small language. The fact is, S/SL contains only eight actions, plus calls to semantic operations. It has no arithmetic or addressing operators. Thus it is natural to use a concise notation (special characters) to represent the few existing features. Conversely, it is not necessary to introduce the relatively bulky framework implied by keywords such as "case", "loop", and "exit". For analogous reasons, notations such as BNF and regular expressions use special characters such as "*" and "|" rather than bulkier symbols such as "repeated" and "or". (See also Hoare's defense of his use of special characters in Communicating Sequential Processes [Hoare 1978].)

Unfortunately, some computer systems do not support special characters such as { and [. As a concession to portability, our S/SL implementation allows do and od as substitutes for { and }, if and fi for [and], and ! for |. We have chosen very short keywords, such as do and if, to preserve most of the concise readability of S/SL.

IV. PROGRAMMING METHODOLOGY

The programming methodology associated with S/SL is perhaps as important as the notation itself. Briefly, this methodology consists of breaking the problem solution into two distinct parts: the abstract algorithm (implemented in S/SL) and the abstract data (written in a base language such as Pascal). The abstract data is further divided into largely independent semantic mechanisms. Each semantic mechanism is an abstract machine that can carry out a well-defined set of instructions (its semantic operations).

Since the abstract algorithm is written in a different language from that used to implement the semantic mechanisms, it is inevitable that we maintain the distinction between the two. By comparison, if we did not use S/SL and we wrote both in a language such as Pascal, these divisions would be easily overlooked, especially during maintenance.

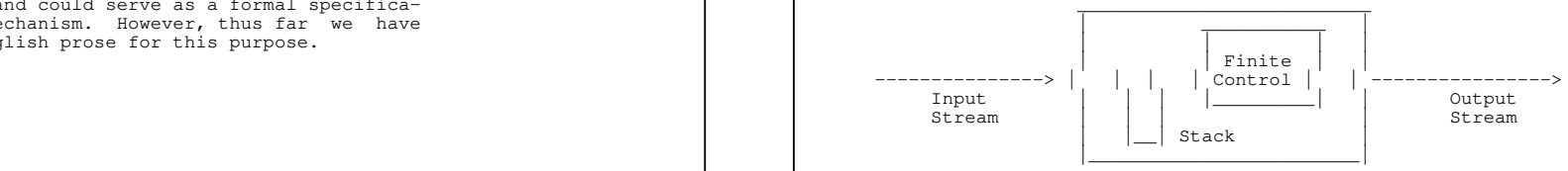
The definition in S/SL of the name of each semantic mechanism along with its operations serves as a concise, readable summary of the underlying data of the program. The programmer is expected to include comments with these definitions that give the meaning (specification) of each operation. These comments could use a notation such as that of abstract data types, and could serve as a formal specification of the semantic mechanism. However, thus far we have been content to use English prose for this purpose.

V. RELATION TO FORMAL LANGUAGES AND AUTOMATA THEORY

We will relate SL to the following theoretic models: pushdown automata, BNF, finite automata and regular expressions.

Those lacking an affection for theory can consider that SL (and S/SL) is a programming language particularly suited to certain tasks. But the more theoretically inclined may choose to consider that SL is a notation for defining deterministic pushdown automata. One of our purposes here is to show that SL is equivalent in formal descriptive power to the LR(k) technique. (See Aho and Ullman [1977] for theoretic background).

A push down automaton (PDA) is a machine that reads a stream of tokens (an input tape). The PDA has an internal configuration that consists of two parts: (1) one of a finite number of control states together with (2) a stack. Each entry on the stack has one of a finite number of values. The PDA can read the next input, change its control state and push or pop its stack depending on the values of the next input, the present control state and the present top of stack.



Pushdown Automaton

If the PDA has (at most) one possible change of configuration for each given input, control state and stack top combination, it is called a deterministic pushdown automaton (DPDA). If there can be more than one such possible changes then the PDA is called nondeterministic; it is a NPDA.

The significance of the DPDA and NPDA models are the following:

(1) Most practical parsers are similar to the DPDA. They are not similar to the NPDA, which would require a heavy overhead to keep track of various possible sequences of configurations (various possible parsing sequences).

(2) The NPDA is equivalent to BNF. This means a language can be described by BNF if and only if it can be parsed by a NPDA.

In brief, the DPDA is a model of practical parsers while the NPDA is a model of parsers for arbitrary BNF grammars.

If we limit ourselves to those languages which can be accepted by a DPDA, then we have deterministic languages. It turns out that this is the same class of languages that can be described by LR(k) grammars. LR(k) is the largest subset of BNF grammars for which deterministic parsers can be automatically generated. A subset of the LR(k) grammars, called LALR(1), seems to be the best present basis for parsers generated from BNF grammars.

What is the connection between SL and DPDA? The answer is that each SL program defines a DPDA. The control state is given by the present point of execution in an SL rule. The stack gives the return points of the presently active SL rules. It is easy to show that an SL program can do no more than a DPDA, because the SL program is effectively a special-purpose DPDA. We can also show that any DPDA can be simulated by an SL program, so we get this result:

Theorem. A language is LR(k) iff it is accepted by an SL program.

The simulation of a DPDA by an SL program depends on using SL choice rules and is not entirely obvious [Lomet 1973, Wu et al 1980].

Persons familiar with LR(k) parsers may be surprised that SL can recognize any language described by an LR(k) grammar. They might argue that the k in LR(k) implies k tokens of look-ahead and SL clearly uses only one symbol of look-ahead. The flaw in this argument is exposed by Knuth's proof [Hopcroft 1969] that any LR(k) language (not grammar) is also an LR(0) language, given an end-marker. So, any LR(k) language can be recognized with no look-ahead at all. This does not mean that every LR(k) grammar is also an LR(0) grammar; rather, it means that any LR(k) grammar can be rewritten to be an LR(0) grammar, given an end-marker.

Our theorem means that if a parser for a particular language can be developed using LR(k) methods, then a parser can be developed using SL, and vice versa. This is a theoretic result and implies nothing about a host of important practical issues. It says nothing about the relative difficulty of writing appropriate BNF versus SL, and nothing about the relative efficiencies of the resulting parsers. It does not imply anything about error handling. Neither does it imply anything about methods of transduction or

convenience of attaching semantic operations to the parsing process.

Let us return to the DPDA model and consider its stack more closely. If the stack is eliminated then we get a finite automaton (FA). A finite automaton is equivalent to regular expressions. If we limit the stack depth to any finite maximum, then the DPDA has only the power of a FA. In terms of SL, this means that any non-recursive SL program necessarily recognizes a regular language, because without recursion the stack can only get so deep. Our example scanner clearly fits this pattern.

Printed by mea from mea

VI. IMPLEMENTING S/SL

Up to now we have discussed S/SL without worrying about its implementation. We have been content to consider that S/SL is a well-defined abstraction supported perhaps by some special computer. This idea of an abstraction supported by underlying software is one of the most important tools available for structuring programs and is used constantly in software engineering.

In this section we will face the implementation problem. The implementation is interesting because it is simple and efficient. Once the implementation of S/SL is understood, it becomes clear how to implement semantic operations.

There are a number of possible ways to implement S/SL. We could transliterate S/SL programs to Pascal, producing a sequence of procedure calls that implement the S/SL operations for input, output, etc. The result would be a "recursive descent" compiler. While the result would be correct, it would be considerably larger than necessary, and this is not the method we favor.

We could translate S/SL directly to machine language for, say, the PDP-11. This too would work, and the machine language would be very fast. But it is relatively difficult to write code generators, and we would need to write one for each computer that is to support S/SL.

Rather than generate code for an existing computer architecture, we will invent an "S/SL machine" which is designed to make S/SL implementation easy, efficient and portable.

An S/SL Machine

Since S/SL is such a small language, our machine will be very simple. To support the SL subset of S/SL it will need only these 12 instructions:

```
 1 jumpForward label
 2 jumpBack    label
 3 input       token
 4 inputAny
 5 emit        token
 6 error       signal
 7 inputChoice table
 8 call        label
 9 return
10 setResult   value
11 choice      table
12 endChoice
```


Instructions 1 and 2 transfer control to the given label; instruction 1 jumps forward to its label while instruction 2 jumps backward to its label. Instruction 3 checks that its operand matches the next input and then reads another input. Instruction 4 (inputAny) implements the "?" action by reading an input without checking for a match. Instructions 5 and 6 implement the output (.) and error (#) actions, causing output tokens and error signals to be emitted to the appropriate streams.

Instruction 7, inputChoice, implements the input choice action. Its operand locates a table of this form:

```
n  (number of choices)
token  label
token  label
...
token  label
default
```

First comes a number n giving the number of alternatives. Then come n token/label table entries. The table is searched from top to bottom, trying to match the present input token. If a match is found, then control is transferred to the label corresponding to the token. If no match is found in the n entries, then control is given to the default instruction following the table. If the S/SL choice has an otherwise alternative (*) then the default is the code for otherwise.

If there is no otherwise alternative then the default is reached only when there is a syntax error in the input stream. The default in this case is an input instruction whose token is the first label of the choice, followed by a jumpBack instruction which transfers to the first alternative. This default forces a mismatch in the input instruction; the mismatch is handled by the strategy for handling syntax errors in input instructions. This default is simple and effective for most error situations, but can be specialized if desired to improve the handling of particular errors.

Instruction 8 calls an S/SL rule; the rule is located by the call's label. Instruction 9 returns from a rule to the instruction just beyond the call. A stack is used to hold the return addresses of rules that have been called but have not yet returned.

Instructions 10, 11 and 12 are used to implement calls to choice rules. The call to a choice rule is translated to:

```
call   label
choice table
```

Printed by mea from mea

The call causes the choice rule to execute; the rule leaves its return value in a variable called "result". The choice instruction searches its table for "result", just as inputChoice searches its table for the current token value. A choice rule always returns by executing ">> value" which is translated to:

```
     setResult  value
     return
```

This assigns the value to the "result" variable so it can be used by the "choice" instruction.

The choice table is followed by a default action. If the S/SL choice had an otherwise alternative (*) then the default is the code for otherwise. But if there was no otherwise alternative, then the default is the endChoice instruction. This instruction is reached only when no labels of alternatives can be matched and there is no otherwise alternative; so endChoice aborts the S/SL program.

Our twelve instructions are sufficient to implement all of S/SL except for semantic operations. We will support each semantic operation by inventing a new instruction to implement that particular operation. Before discussing these new instructions, we will give an example S/SL program translated into S/SL machine instructions.

Translating S/SL to Machine Instructions

The mapping from an S/SL program to instructions for our S/SL machine is straightforward as we will now show. Here is our SkipNoise S/SL rule translated to a sequence of instructions.

```
S/SL Source              Assembly Language Location: Machine Code

SkipNoise:
   {                       L1: inputChoice        51: 7
     [                           Table            52: 7
       |blank:               L2: jumpForward      53: 1
                                 L4               54:12
       |illegalChar:        L3: error            55: 6
         #badChar                  badChar         56:10
                               jumpForward        57: 1
                                 L4               58: 8
                             Table: 2             59: 2
                               blank              60: 2
                                 L2               61: 8
                               illegalChar        62: 3
                                 L3               63: 8
```



```
       |*:                       jumpForward      64: 1
         >                           L5           65: 3
     ]                         L4: jumpBack       66: 2
   }                                 L1           67:16
   ;                         L5: return           68: 8
```

The numbers representing the S/SL program in machine language are given on the right. We have arbitrarily started the code for SkipNoise at location 51. The first instruction is an inputChoice (7) so location 51 contains 7. Location 52 holds the relative location (7) of the choice table; this 7 is added to 52 to find the table (at location 59). The next instruction is "jumpForward L4" which goes to the end of the input choice. This appears in locations 53 and 54 as 1 (jumpForward) and 12; 12 is a relative address (12 added to 54 gives 66 which is L4's location). We have used relative addressing throughout to make the encoding of target labels more compact.

It is straightforward to design a microprocessor [Lazar 1980] or to write a Pascal program to execute this S/SL machine language. We are not planning to build such a microprocessor in the near future, but we have written the Pascal program, which we will now describe.

An S/SL Table Walker

We call the sequence of numbers representing an S/SL program an S/SL table. These numbers can be stored in a Pascal array. A Pascal program which accesses this table and simulates an S/SL machine is called a table walker. It "walks" through the table executing instructions and thus carrying out the actions of the S/SL program.

We assume that the following procedures have been written in Pascal.

AcceptInputToken – this reads the next token in the input stream into the "token" variable, which is global to the table walker.

EmitOutputToken – this emits the token that is its parameter to the output stream.

SignalError – this generates the error message specified by its parameter. For certain values of its parameter (for "fatal" errors), SignalError sets "processing" to false, thereby causing the table walker to terminate.

HandleSyntaxError – takes some appropriate error handling action; it is called when the present input token is

```
   syntactically  illegal.  Its parameter gives the expected
   input token.

We have put a small letter "o" as the first letter  of  each
instruction  name,  for example "return" becomes oReturn, to
avoid clashes with other names in the  Pascal  program.   We
have defined oJumpForward to be 1, oJumpBack to be 2, oInput
to be 3, and so on.

   We have factored out the logic that  searches  tables  in
choice  actions.   This  logic,  contained  in the procedure
named Choose, is used by input  choices,  rule  choices  and
semantic  choices.   This  version of the table walker omits
the implementation of semantic operations. It is  sufficient
for  implementing  a  compiler  pass without semantic opera-
tions, such as a parser.  The next section shows how  seman-
tic operations are supported.  Here is the table walker:



tablePointer := 1;   {Locates instruction to execute}
returnTop := 0;      {Locates top of return Stack}
processing := true;  {Is set to false to stop the table walker}

while processing do
   begin
      operation := sslTable[tablePointer];
      tablePointer := tablePointer + 1;

      case operation of

         oJumpForward:
            tablePointer := tablePointer + sslTable[tablePointer];

         oJumpBack:
            tablePointer := tablePointer - sslTable[tablePointer];

         oInput:
            begin
               if token = sslTable[tablePointer] then
                  AcceptInputToken
               else
                  HandleSyntaxError(sslTable[tablePointer]);
               tablePointer := tablePointer + 1
            end;

         oInputAny:
            AcceptInputToken;

         oEmit:
            begin
               EmitOutputToken(sslTable[tablePointer]);
               tablePointer := tablePointer + 1
```



```
            end;

         oError:  ...similar to oEmit...

         oInputChoice:
            begin
               choiceTable := tablePointer +
                  sslTable[tablePointer];
               result := token;
               Choose;
               if choiceMatch then
                  AcceptInputToken
            end;

         oCall:
            if returnTop < returnSize then
               begin
                  returnTop := returnTop + 1;
                  returnStack[returnTop] := tablePointer + 1;
                  tablePointer := sslTable[tablePointer]
               end
            else  ...abort, setting processing to false...

         oReturn:
            if returnTop > 0 then
               begin
                  tablePointer := returnStack[returnTop];
                  returnTop := returnTop - 1
               end
            else
               processing := false;   {Return from main rule}

         oSetResult:
            begin
               result := sslTable[tablePointer];
               tablePointer := tablePointer + 1
            end;

         oChoice:
            begin
               choiceTable := tablePointer +
                  sslTable[tablePointer];
               Choose
            end;

         oEndChoice:  ...abort, setting processing to false...

         ...alternatives to implement semantic operations...

      end   { case }
   end { while }

As  can  be  seen by reading this program, it is a simulator
```

for an S/SL machine.

The instructions jumpForward and jumpBack are implemented as jumps relative to the current value of tablePointer. Similarly, in the inputChoice and choice instructions, the choice tables are located relative to the current tablePointer. This use of relative addressing together with separate forward and backward jumps makes it relatively easy to compact the sslTable to use byte entries. For this compaction to be practical, we would need to encode the label in a call instruction into two bytes, but we will not go into these optimizations here.

Supporting Semantic Operations

Our table walker supports SL but not S/SL. To support S/SL it must be enhanced with new instructions. We add a new alternative to the table walker's case statement for each semantic operation. For example, the CountPop semantic operation can be implemented by

```
oCountPop:
   countTop := countTop - 1;
```

where countTop is a variable pointing to the top of the count stack.

We also invent the following special instruction to help implement parameterized semantic operations:

```
13 setParameter value
```

This instruction (number 13) assigns its value to a variable called "parameter". Before invoking a parameterized semantic operation, such as CountPush, a setParameter instruction is executed to give the appropriate value to "parameter". For example, PushCount(zero) in S/SL is translated into the instructions:

```
setParameter zero
CountPush
```

We translate a semantic choice operation into an instruction to invoke the operation followed by a choice instruction. For example, CountChoose in S/SL is translated to the instructions:

```
CountChoose
choice     table
```

The CountChoose instruction assigns to the variable named "result", and the choice instruction searches its table for

this value.

VII. EXPERIENCE AND OBSERVATIONS

Experience using S/SL

S/SL has been used in implementing three compilers: Speckle (a PL/1 subset [Miceli 1977], Toronto Euclid [Holt 1978], and PT (a Pascal subset) [Rosselet 1980]; it has also been used to implement an S/SL processor. Previous to S/SL, syntax and semantic charts [Cordy 1979] had been used by the authors, most notably for implementing SP/k (a PL/1 subset) [Holt 1977]. These charts are highly readable and have an efficient implementation. Their primary disadvantage is that they are not machine readable and, like flowcharts, are difficult to maintain.

When designing the Euclid compiler, we contemplated using charts, but decided to use S/SL instead. This turns out to have been a fortuitous decision, as we now believe that the job would not have been possible using charts. What we had not predicted was the sheer bulk of programs in S/SL needed to compile a language like Euclid. Five passes of the Euclid compiler (parser, table builder, type conformance checking, storage allocation and code generation) were written in S/SL, with a total of about 24,000 lines of S/SL code. It would probably not have been possible to keep track of the equivalent volume of hand-drawn charts.

The technique of software development used in the Euclid compiler was based on S/SL pass skeletons in the following manner. First a parser was developed for the Euclid source language; this parser produced an output stream, which we will call I-code (intermediate code). I-code is essentially the complete syntax-checked Euclid program, encoded as a sequence of tokens. About the only interesting transformation from the source is that most operators have been moved into postfix positions.

Once the parser was implemented, an S/SL program was written to accept an I-code stream and reproduce the same stream as output. This seemingly useless program served two key purposes. First, it formally specified the stream coming out of the parser. Second, it served as a skeleton for each of the following passes of the compiler. Each of these passes makes only minor modifications on the stream; most information is relayed to the following passes via the symbol/type table. Each pass was constructed by modifying the skeleton S/SL program, most commonly by adding calls to semantic operations. As a result, all of these passes are similar in structure, and the whole compiler is relatively easy to understand.

As each pass executes, its S/SL parses the pass's input stream. This parsing provides an important check on the

interface between passes, verifying that the input I-code stream is syntactically correct.

In general the experience of writing several compilers using S/SL has been a happy one. The S/SL programs have been relatively easy to write and maintain. The computer time spent executing code written in S/SL has been small compared to time spent in other activities, such as input/output. The tables produced by S/SL together with the table walker are observed to be quite small, better than or comparable to other techniques such as writing in a high level language or using LR(k).

Using S/SL for Non-Compiler Applications

Although S/SL evolved as a tool for constructing compilers, it appears to be useful for a much larger class of problems. The concept of a pure control language and abstract data (semantic mechanisms) are not intrinsically tied to compiler writing. In non-compiler applications, these concepts become the center of focus while stream-oriented features (input and output of tokens) become more peripheral, although still widely useful.

In the future we expect to experiment with S/SL as a software specification language. As such an S/SL program will serve as the top level, executable design for a system, which is to be implemented later by programming its semantic mechanisms. For large systems, these implementations may in turn be designed using S/SL, the result being several levels of S/SL implementing increasing levels of detail of the system.

A Synthesis of Ideas

The S/SL language is a synthesis of several programming concepts. The following related notations have been directly influential.

(a) BNF and regular expressions.
(b) Syntax charts and semantic charts.
(c) Recursive descent compilers (written in Pascal-like languages).
(d) Separable transition diagrams [Conway 1963].
(e) Table-driven coding, such as the scheme used in the PL/C compiler [Wilcox 1971].
(f) Data encapsulation techniques, such as those in Simula and Euclid.

What seems encouraging about S/SL is that it captures such a large fraction of the power of these notations, while itself

remaining so simple.

There are many possibilities for expansion. One could make S/SL into a Pascal-like language by introducing declarations, expressions and assignments. The result might be particularly attractive as an interactive language, due to the conciseness of programs. Another possibility would be to allow several input streams and output streams. If we also allow several processes then the language becomes similar in form and concept to Hoare's cooperating sequential processes.

Generally we have resisted the urge to expand S/SL, on the principle that "small is beautiful". Its simplicity results in readability, a great deal of flexibility and an easy, efficient implementation.

ACKNOWLEDGMENTS

The development of S/SL was aided and influenced by a number of people; Prof. Rudy Marty was especially helpful. We are happy to acknowledge the support from the Natural Sciences and Engineering Research Council of Canada and from Bell-Northern Research Limited.

REFERENCES

Aho, A.V. and Ullman, J.D. [1977] Principles of Compiler Design, Addison-Wesley Publishing Company, 1977.

Barnard, D.T. [1975] Automatic Generation of Syntax-Repairing and Paragraphing Parsers, Computer Systems Research Group, University of Toronto, Technical Report CSRG-52, March 1975.

Conway, M.E. [1963] "Design of a separable transition diagram compiler", Comm. ACM Vol.6, No.7, 396-408.

Cordy, J.R. [1976] A Diagrammatic Approach to Programming Language Semantics, Computer Systems Research Group, University of Toronto, Technical Report CSRG-67, 1976.

Cordy, J.R., Holt, R.C., and Wortman, D.B. [1979] "Semantic Charts: A Diagrammatic Approach to Semantic Processing", Proceedings of SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices Vol.14, No.8 (August 1979), pp.39-49.

Cordy, J.R. and Holt, R.C. [1980] Specification of S/SL: Syntax/Semantic Language, Computer Systems Research Group, University of Toronto.

CSRG [1980] Toronto Euclid Compiler, Computer Systems Research Group, University of Toronto, January 1980.

Hoare, C.A.R. [1978] "Communicating Sequential Processes", Comm. ACM, Vol.21, No.8, (August 1978), pp.666-677.

Holt, R.C., Wortman, D.B., Barnard, D.T., Cordy, J.R. [1977] "SP/k: A System for Teaching Computer Programming", Comm. ACM, Vol.20, No.5 (May 1977), pp.301-309.

Holt, R.C., Wortman, D.B., Cordy, J.R., Crowe, D.R. [1978] "The Euclid Language: A Progress Report", Proceedings of the ACM National Conference, December 1978.

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

Mass, 1969.

Jensen, K. and Wirth, N. [1974] PASCAL User manual and report, Springer-Verlag, Berlin, 1974.

Lazar, Dov S. [1980] The Design of a Hardware Compiler: A Compiling Dedicated Machine, M.Sc. thesis, Department of Computer Science, University of Toronto, 1980.

Lomet, D.B. [1973] "A Formalization of Transition Diagram Systems", Journal ACM, Vol.20, No.2, (April 1973) pp.235-257.

Wu, T., McKenzie, P. and Spinney, B. [1980] S/SL Parses the LR(k) Languages, Computer Systems Research Group, University of Toronto, September 1980.

Miceli, J. [1977] Some experiences with a One-Man Language, Computer Systems Research Group, University of Toronto, Technical Note 13, December 1977.

Rosselet, J.A. [1980] PT: A Pascal Subset, M.Sc. Thesis, University of Toronto, January 1980.

Wilcox, T.R. [1971] Generating Machine Language for High-Level Programming Languages, Ph.D. Thesis, Cornell University, Ithaca, N.Y., 1971.

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.

Copyright (C) 1979, 1980 by the University of Toronto.
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 lower 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 program is:

```
[inputDefinition]                    [output-
Definition]                  [inputOutputDefinition]
[errorDefinition]                      {type-
OrMechanismDefinition}                  RULES
  {rule}                  END
```

INPUT AND OUTPUT DEFINITIONS

An inputDefinition is:

```
INPUT  ":"                        {tokenDefi-
nition} ";"
```

An outputDefinition is:

```
OUTPUT ":"                        {tokenDefi-
nition} ";"
```

An inputOutputDefinition is:


```
INPUT OUTPUT ":"                        {tok-
enDefinition} ";"
```

A tokenDefinition 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  ":"                      {errorSig-
nalDefinition} ";"
```

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 {"," val-
ueId}    ":"                                    {action}   }
                          ["|"  "*"  ":"
{action}   ]                          "]"                   j.
">>"      valueId                        k.        updateOpId
                l.  parameterizedUpdateOpId "(" valueId ")"
                m.    "["   choiceOpId
{"|"    valueId    {","   valueId}   ":"
{action}   }                                 ["|"    "*"    ":"
                              {action}  ]
"]"                   n.   "["  parameterizedChoiceOpId  "("
valueId  ")"                          {"|" valueId {"," val-
ueId}   ":"                                    {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 program is:

                [inputDefinition]                 [output-
Definition]                          [inputOutputDefinition]
                [errorDefinition]                    {type-
OrMechanismDefinition}                                RULES
                  {rule}                  END

An inputDefinition is:

                INPUT ":"                       {tokenDefi-
nition} ";"

An outputDefinition is:

                OUTPUT ":"                      {tokenDefi-
nition} ";"

An inputOutputDefinition is:

                INPUT OUTPUT ":"                      {tok-
enDefinition} ";"

A tokenDefinition is:

                id [string] ["=" tokenValue]

An errorDefinition is:

                ERROR  ":"                       {errorSig-
nalDefinition} ";"

An errorSignalDefinition is:

                id ["=" errorValue]

A typeOrMechanismDefinition is one of:

                a.    typeDefinition                    b.
mechanismDefinition
```



```
A typeDefinition is:

                TYPE id ":"                      {valueDef-
inition} ";"

A valueDefinition is:

                id ["=" value]

A mechanismDefinition is:

                MECHANISM    id    ":"
{operationDefinition} ";"

A rule is one of:

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

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

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 {"," val-
ueId}   ":"                                  {action}    }
                          ["|"  "*"  ":"
{action}  ]                       "]"                    j.
">>"       valueId                      k.       updateOpId
                l.  parameterizedUpdateOpId "(" valueId ")"
```

```
                m.     "["   choiceOpId
{"|"   valueId   {","   valueId}    ":"
{action}    }                                  ["|"   "*"   ":"
                                {action}  ]
"]"                  n.   "["  parameterizedChoiceOpId  "("
valueId ")"                          {"|" valueId {","  val-
ueId}    ":"                                    {action}   }
                          ["|"  "*"  ":"
{action} ]                       "]"
```