An Introduction to S/SL: Syntax/Semantic Language

R. C. HOLT, J. R. CORDY, and D. B. WORTMAN
University of Toronto

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 do LR(k) parsers. Complete S/SL includes invocation of semantic operations implemented in another language such as PASCAL.

S/SL implies a top-down 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 reduce the effort needed to understand and maintain the resulting software.

S/SL has been used to construct compilers for SPECKLE (a PL/I subset), PT (a PASCAL subset), Toronto EUCLID, and Concurrent EUCLID. It has been used to implement scanners, parsers, semantic analyzers, and code generators.

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

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

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*design languages*; *S/SL*; D.3.4 [**Programming Languages**]: Processors

General Terms: Languages

Additional Key Words and Phrases: LR(k)

1. INTRODUCTION

S/SL is a programming language developed at the University of Toronto as a tool for constructing compilers. It is a modest language, incorporating only the following features: (1) sequences, repetitions, and selections of actions (statements), (2) input, matching, and output of tokens, (3) output of error signals, (4) subroutines (called rules), and (5) invocation of semantic operations implemented

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and by Bell-Northern Research Limited.
Authors' address: Computer Systems Research Group, University of Toronto, Toronto, Canada M5S 1A1.

in a base language such as PASCAL. S/SL is a language without data or assignment; it is a pure control language [8]. Data can be manipulated only via semantic operations.

We give a computational model for S/SL and a summary of the 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 relate our experience in implementing production compilers using S/SL, and we discuss the methodology of software development implied by S/SL.

This is followed by a discussion of the implementation of S/SL. We describe the relationship of S/SL to the theory of formal languages and automata. Finally, we compare S/SL and LR(k) as software tools.

2. A COMPUTATIONAL MODEL

S/SL assumes the computational model illustrated in Figure 1. 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.

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 in order 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.

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.

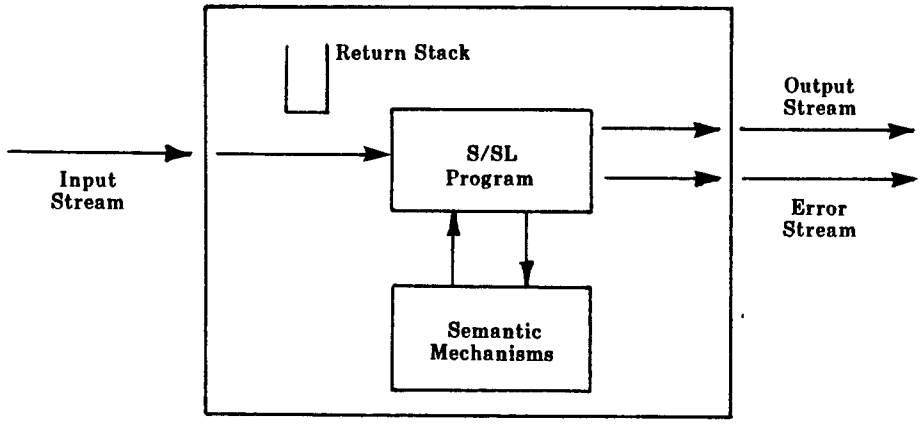

Fig. 1. Computational model for S/SL.

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 [10] 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.

3. FEATURES OF S/SL

This section summarizes the S/SL language; for a precise definition of the language, see [8].

First, we describe 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 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).

3.1 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 single 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* Action. 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, labeled by the asterisk (*), is executed.

The otherwise clause,

```
| *:
  actions
```

is optional. If it is 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 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.

3.2 Definitions in S/SL

We need to make certain definitions that are used by S/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 '+'. Either name 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.

3.3 Semantic Mechanisms

SL is sufficient for implementing parsers for languages having LR(k) grammars, such as PASCAL and ALGOL 60. However, SL 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 for 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 their implementation is carried out 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 operatio 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.

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 rule.

3.4 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(*exp*1, *exp*2);

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 accepts "*exp*1" the first time through the loop and "*exp*2" 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 reentered 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 that 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 implementation details, we give several more example S/SL programs.

4. 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. Persons with a detailed knowledge of compilers may wish to skip some of these examples.

4.1 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 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 [17]. This example uses the notation {[and]}. This is not a new construct but simply a choice action nested directly inside a cycle.

4.2 Parsing

It is straightforward to write a parser for a language such as PASCAL in S/SL. The PASCAL report [12] 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 now give three examples of parsing: recognizing a statement, handling an **if** statement, and producing postfix for expressions.

(1) *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 recall; so one rule must handle both.

(2) *Handling* **if** *Statements.* As an example of rules for individual statements, we 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 resolved, 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.

(3) *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 operators are left associative, that * and / have higher precedence than + and −, and that 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 Expression 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.

We have shown how to handle left-associative operators, such as addition and multiplication. Right-associative operators, such as exponentiation, are easily handled using recursive S/SL rules.

4.3 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 run-time stack used in evaluating expressions. Each entry in the type stack gives the type of the corresponding operand in the run-time stack. The definition in S/SL of this semantic mechanism would be similar to the definition we gave for the stack of counters. We 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.

Since the postfix stream was created by a previous pass, we can assume that it is well formed. Thus, the PostfixExpression rule does not check that each operator has its required number of operands.

To simplify our example, we consider only these operators: addition, equality, and intersection. 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 can be compared 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)
    | intersect:
      @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 pseudomachine code (P-code) while they are checking types. This is done in the PT PASCAL compiler [17]. We could incorporate this type checking into the pass that does syntax checking (the parser).

4.4 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 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.

4.5 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 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 [2, 4, 6]; 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; 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 bulky 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 [7].)

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 short keywords, such as do and if, to preserve most of the concise readability of S/SL.

5. EXPERIENCE USING S/SL TO DEVELOP COMPILERS

S/SL has been used in implementing a number of production compilers: SPECKLE (a PL/I subset) [16], Toronto EUCLID [10, 20, 21], PT (a PASCAL subset) [17], and Concurrent EUCLID [5]. It has also been used to implement the S/SL processor.

5.1 The SP/*k* Compiler

Previous to S/SL, syntax and semantic charts were used by the authors, most notably in implementing the SP/*k* subset of PL/I [9]. SP/*k* was designed to be a

high-speed, highly diagnostic compiler. Syntax charts were developed in preference to LR(k)-style parsing so that syntax error-handling could be done easily and diagnostically [2]. Semantic charts were developed to gain the convenience and efficiency of syntax charts in semantic analysis [4, 6]. Our implementation of syntax and semantic charts via tables and a table walker is essentially the same as the implementation of S/SL. The SP/k compiler, which uses this scheme for its parser and semantic analysis passes, is extremely fast, compiling over 13,000 lines per minute on an IBM 370 model 165-II. This compilation rate is significantly better than high-speed, hand-written compilers such as PL/C and IBM's PL/I Checkout compiler. See [9] for detailed comparisons of these rates; note that one of the reasons that the SP/k compiler is faster than PL/C and Checkout is that it handles a limited subset of PL/I.

The disadvantage of charts is that they are not machine readable and, like flowcharts, are difficult to maintain. These shortcomings led to the development of S/SL.

5.2 The Toronto EUCLID Compiler

When designing the Toronto 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 Toronto EUCLID compiler was based on an S/SL skeleton for compiler passes in the following manner. First, a parser was developed for the source language; this parser produced an output stream, which we 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 expression 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 precisely 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 program parses the pass's input stream. This parsing provides an important check on the interface between passes, verifying that the previous pass produced a reasonable I-code stream.

The use of S/SL contributed in two critical ways to the successful completion of the self-compiling Toronto EUCLID compiler. First, it helped us organize and

simplify the challenging job of compiling a language of the complexity of Toronto EUCLID. Second, since algorithms written in S/SL are space efficient (generally much more so than those using a high-level language), we were able to fit the compiler into the target minicomputer (Digital PDP-11/50).

The Toronto EUCLID compiler is not particularly fast when run on the PDP-11/50 because the symbol table is disk resident and swapping of the compiler (by UNIX™) is slow.

5.3 The Concurrent EUCLID Compiler

Once Toronto EUCLID was implemented, it was used in developing the prototype of a UNIX-compatible operating system. This effort lead to the design of a new language, Concurrent EUCLID, which omits various features of Toronto EUCLID and adds processes and monitors [5]. The compiler was developed to be highly portable (especially to 16-bit microprocessors), to compile fast, and to generate excellent code.

Generally, this compiler is structured like its predecessor, Toronto EUCLID, but the symbol table is core resident, and there is one less pass. The new compiler uses S/SL to implement its scanner (this was not done for Toronto EUCLID).

The new compiler uses S/SL more extensively in its code generation pass. The first code generator for Concurrent EUCLID produced excellent code for the Motorola 68000 [18]. The S/SL for this pass is about 6000 lines long. It is approximately 60 percent independent of the target machine architecture and has been used as the basis of pluggable code generators for the IBM 360, the Motorola 6809, the PDP-11, and the Digital Equipment VAX.

In the new compiler, S/SL has again supported a clean organization and compactly encoded algorithms. In addition, it has been key to developing retargetable code generators.

The Concurrent EUCLID compiler is small (about 50K bytes for the largest pass) and fast (about 8 minutes for a compilation of a 5000-line pass of itself on the PDP-11/50).

5.4 PT and Student Projects

S/SL has been used for three years in undergraduate courses in compiler-writing at the University of Toronto. The students are given a multipass, self-compiling compiler and asked to modify or extend it. This was originally done with SPECKLE (PL/I subset) and then with PT (PASCAL subset).

Students seem to find it easy to use S/SL. After an introduction to compilation concepts, S/SL is presented to them in a one-hour lecture. Then the students are given an S/SL manual, and they begin writing S/SL programs.

The PT compiler is written to be readable and to be used as a model of compiler structure. It uses S/SL to drive each of its three passes: a scanner–parser pass, a semantic analysis pass, and a code generator pass. The last pass generates good PDP-11 code, and students have replaced it by code generators for other machines.

Persons interested in using S/SL for production work should study the PT compiler; see Rosselet's report [17], which includes the complete S/SL for the

PT compiler. Running on the PDP-11/50, the PT compiler is reasonably fast, compiling one of its own 2500-line passes in 4 minutes.

We can summarize our experience with S/SL as follows. It has proved itself as an effective tool for developing production-quality software. It has been used for relatively simple tasks, such as scanning and parsing, and for complex tasks, such as semantic analysis and code generation.

6. 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 are 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 langauge such as PASCAL, these divisions would be easily overlooked, especially during maintenance.

The definition in S/SL of the name of each 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.

6.1 Developing a Code Generator

As an example of the use of this methodology, we take the development of the Motorola 68000 code generator pass of the Concurrent EUCLID compiler [18].

This software consists of two major parts: (1) the pass's abstract algorithm implemented in S/SL and (2) the abstract data (semantic mechanisms) written in the base language (EUCLID). There are also supporting routines, including the S/SL table walker and an I/O package, but these had previously been implemented to support other compiler passes.

This code generator was developed in two major steps. In the first, the skeletal S/SL (about 1200 lines), which was used by other compiler passes, was expanded to become the pass's abstract algorithm (about 6000 lines). This S/SL algorithm was a complete code generator in that it specified the acceptance of all input tokens, all updating and inspection of semantic mechanisms, and all emission of Motorola 68000 machine constructions. What remained to be done was to implement the semantic mechanisms.

In the second step, these mechanisms were written in EUCLID. This was a straightforward task, as the mechanisms were nearly independent and had been precisely specified as a part of the development of the abstract algorithm in S/SL.

The division of this complex piece of software into two major parts (S/SL and abstract data) and the data into separate mechanisms greatly facilitated development. It has simplified the task of replacing major parts to produce code generators for other target architectures.

6.2 Using S/SL for Noncompiler Applications

Although S/SL evolved as a tool for constructing compilers, it appears to be useful for a larger class of problems. The concepts of an abstract algorithm and abstract data are not intrinsically tied to compiler writing. In noncompiler 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.

7. 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 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 into 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 into mechine language for, say, the PDP-11. This too would work, and the machine language would be 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 invent an "S/SL machine" that is designed to make S/SL implementation easy, efficient, and portable.

7.1 An S/SL Machine

Since S/SL is such a small language, our machine can be simple. To support the SL subset of S/SL, it needs 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 than 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 alternatives)
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 the otherwise alternative.

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 that 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
```

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. Failure of a rule choice implies a compiler failure; so endChoice aborts the S/SL program.

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

7.2 Translating S/SL into Machine Instructions

The mapping from an S/SL program to instructions for our S/SL machine is straightforward, as we now show. Here is our SkipNoise S/SL rule translated into 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 just beyond 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 compact.

It is straightforward to design a microprocessor [14] 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 now describe.

7.3 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 that 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	reads the next token in the input stream into the "token" variable, which is global to the table walker;
EmitOutputToken	emits the token that is its parameter to the output stream;
SignalError	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 (e.g., *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 algorithm that searches tables in choice actions. This algorithm, 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 operations, such as a parser. The next section shows how semantic operations are supported. Here is the table walker:

```
tablePointer := 1;      {Locates instruction to execute}
returnTop := 0;         {Locates top of returnStack}
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 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 do not go into these optimizations here.

7.4 Supporting Semantic Operations

The table walker as described so far 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 into the instructions

```
CountChoose
choice table
```

The CountChoose instruction assigns to the variable named "result," and the choice instruction searches its table for this value. This is analogous to the implementation of calls to choice rules.

8. RELATION TO FORMAL LANGUAGES AND AUTOMATA THEORY

We now relate S/SL to the following theoretical models: Turing machines, pushdown automata, BNF, finite automata, and regular expressions. (See Hopcroft and Ullman [11] for theoretical background.)

An S/SL program can be made to simulate an arbitrary Turing machine by the following construction. The S/SL program simply copies its input stream, token

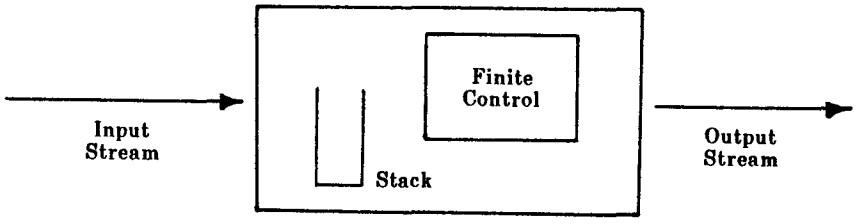

Fig. 2. Pushdown automaton.

by token, into an appropriate semantic mechanism, which proceeds to behave like the Turing machine. The implication is that, in recognition power, S/SL is as powerful as any possible computing device. The question of how powerful S/SL is becomes interesting when we disallow semantic mechanisms, restricting ourselves to the SL subset. We now explore this question.

A pushdown automaton (PDA) (see Figure 2) 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 next input, the present control state, and the present top of stack. 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 change, then the PDA is called *nondeterministic*; it is a NPDA.

The significance of the DPDA and NPDA models is as follows:

(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 that 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 (context-free) grammars for which left-to-right 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. For a formal proof of this theorem see [15]. This

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.

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 lookahead while SL clearly uses only one symbol of lookahead. The flaw in this argument is exposed by Knuth's proof [11] 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 lookahead 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.

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). Finite automata are 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 nonrecursive SL program necessarily recognizes a regular language, because without recursion the stack can only get so deep. Our example scanner clearly fits this pattern.

9. LR(k) VERSUS SL AS SOFTWARE TOOLS

LR(k) and the SL subset of S/SL are equivalent in terms of formal recognition power. This is a theoretical result and implies little about their relative effectiveness in producing software. Any comparison of LR(k) and S/SL as software tools is difficult, as they are fundamentally different in several ways. To allow an easier comparison with LR(k), we narrow our attention to parsing and limit S/SL to the SL subset.

9.1 Notable Differences

The notation for LR(k) is BNF (or context-free grammars). BNF is nondeterministic and allows ambiguities. By contrast, the SL notation is algorithmic and deterministic; so an SL program cannot be ambiguous.

An LR(k) processor performs a significant computation to produce a parser from a BNF grammar [1]. This computation is nonlinear in time, becoming increasingly slow with larger grammars. For grammars of more than a few hundred productions, LR(k) processors typically abort due to excessive memory requirements or running time.

By contrast, an S/SL processor is essentially an assembler. Our present S/SL processor generates tables for an 8000-line S/SL program in ten minutes on a PDP-11/50.

9.2 Ease of Producing a Parser

We now consider the relative ease of producing a parser using LR(k) or SL, given the description of a high-level language to be compiled. The steps taken for both techniques are basically the same:

(1) determine the precise syntax of the language (this might be given in the language description in English, BNF, extended BNF, syntax charts, or some other notation);

(2) specify this syntax using BNF (or SL);
(3) submit the result to an LR(k) (or S/SL) processor, which produces corresponding parse tables.

In an ideal situation (from the LR(k) point of view), the language description contains a BNF grammar; so steps (1) and (2) are trivial. And, ideally, in step (3) an LR(k) processor produces a parser from this grammar without significant human intervention.

In practice, things do not usually work out this way. Many language descriptions, including those for FORTRAN and COBOL, do not contain BNF. BNF grammars that appear in language descriptions are typically ambiguous and rarely meet the restriction that LR(k) imposes on BNF. Not all BNF grammars, not even all unambiguous grammars, are acceptable to LR(k), and there is no simple, local test the grammar writer can apply to see if his grammar meets the LR(k) restriction. He must repeatedly hand modify the grammar until it is finally acceptable to the LR(k) processor. Thus, step (3) can be tedious. Johnson [13] discusses this difficulty in his presentation of the YACC LR(k) system: "In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here.... The services of a local guru may be appropriate."

Now consider the analogous process using S/SL. In step (2) the SL programmer writes an SL program, which is comparable to a recursive descent parser written in PASCAL. In complex parsing situations, he must manually determine "lookahead" tokens; by contrast, an LR(k) processor determines these automatically. Fortunately, the set of lookahead tokens is obvious in most situations. Once the parser's SL is complete, it is submitted to the S/SL processor (step (3)). There are no restrictions on SL analogous to those LR(k) imposes on BNF. Hence, when using SL, step (3) is straightforward.

As we approach the ideal LR(k) situation of a language specification containing an LR(k) BNF grammar, the LR(k) approach offers the following advantages over SL. The automatically generated parser exactly matches the specified syntax (assuming a correct LR(k) processor), and the problem of determining lookahead tokens is solved automatically.

In brief, LR(k) parsers are usually developed by writing and repeatedly manipulating a nondeterministic (BNF) grammar. An SL parser is programmed as an algorithm, much the same way a parser is programmed recursively in PASCAL. Neither the LR(k) nor the SL technique is particularly difficult to use. For a language of the complexity of PASCAL, using either technique an expert can produce a parser in a day.

9.3 Parser Quality

We now compare LR(k) and S/SL parsers on the basis of size, speed, and error handling. Based on informal studies, it appears that S/SL tables are significantly more compact. One comparison of S/SL-like tables for a PASCAL-like language to LR(k) YACC tables [13] showed a 2 : 1 S/SL advantage. It appears that a new version of YACC produces more compact tables than did earlier versions; other persons may wish to do further comparisons of table sizes for SL versus LR(k).

Comparison of parsing times is difficult, and no relative timings are available. Parser speeds for both techniques are highly dependent on details of the BNF grammar or S/SL program.

The implementer of a SL parser can collect execution statistics for the parser to obtain a profile giving a count of the executions of each action (statement) of the SL program. The high-frequency sections can be hand tuned for faster execution. (This method of optimization is exactly the same as is used for programs in languages like PASCAL.) By contrast, such tuning is difficult using LR(k) because important parsing actions such as "lookaheads" are not obvious from the BNF. It is difficult to predict from the BNF what optimizations a particular LR(k) processor may attempt.

One thing that is clear is that both the LR(k) and SL techniques can produce fast parsers. Our experience has been that time spent parsing is rarely as much as 15 percent of overall compilation time. Efforts to speed up compilation are usually best spent on parts of the compiler other than the parser.

9.4 Handling Syntax Errors

One of the reasons for developing chart-driven parsers and then S/SL parsers was to obtain better syntax error-handling than was available using LR(k). It has been our experience that top-down techniques like SL make good error-handling easy, because the context of the error is readily available and because the parse tables correlate directly with the SL source. By contrast, with LR(k) several potential reductions may be in progress simultaneously, and the LR(k) parse tables do not correlate directly with the BNF grammar.

We now summarize our comparison of LR(k) versus S/SL as software tools. Point-by-point comparison of the two techniques is difficult because they are so dissimilar. S/SL is a more general software tool and competes directly with LR(k) only in the area of parsing. Even there the comparison is difficult, and few figures are available. It is the authors' belief that, compared to LR(k) parsers, SL parsers provide easier development, smaller tables, comparable speed, and better error-handling. Other persons may wish to investigate this comparison further.

10. A SYNTHESIS OF IDEAS

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

(1) BNF and regular expressions;
(2) syntax charts and semantic charts;
(3) recursive descent compilers (written in PASCAL-like languages);
(4) separable transition diagrams [3];
(5) table-driven coding, such as the scheme used in the PL/C compiler [19];
(6) 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 Communicating Sequential Processes [7].

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. Suggestions by referees of this paper helped the clarity of several sections.

REFERENCES

1. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
2. BARNARD, D.T. Automatic generation of syntax-repairing and paragraphing parsers. Tech. Rep. CSRG-52, Computer Systems Research Group, Univ. of Toronto, Canada, Mar. 1975.
3. CONWAY, M.E. Design of a separable transition-diagram compiler. *Commun. ACM 6,* 7 (July 1963), 396–408.
4. CORDY, J.R. A diagrammatic approach to programming language semantics. Tech. Rep. CSRG-67, Computer Systems Research Group, Univ. of Toronto, Canada, 1976.
5. CORDY, J.R., AND HOLT, R.C. Specification of Concurrent Euclid. Tech. Rep. CSRG-115, Computer Systems Research Group, Univ. of Toronto, Canada, July 1980.
6. CORDY, J.R., HOLT, R.C., AND WORTMAN, D.B. Semantic charts: A diagrammatic approach to semantic processing. In Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colo., Aug. 6–10, 1979. *SIGPLAN Notices* (ACM) *14,* 8 (Aug. 1979), 39–49.
7. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM 21,* 8 (Aug. 1978), 666–677.
8. HOLT, R.C., CORDY, J.R., AND WORTMAN, D.B. S/SL: Syntax/Semantic Language introduction and specification. Tech. Rep. CSRG-118, Computer Systems Research Group, Univ. of Toronoto, Canada, Sept. 1980.
9. HOLT, R.C., WORTMAN, D.B., BARNARD, D.T., AND CORDY, J.R. SP/k: A system for teaching computer programming. *Commun. ACM 20,* 5 (May 1977), 301–309.
10. HOLT, R.C., WORTMAN, D.B., CORDY, J.R., AND CROWE, D.R. The Euclid language: A progress report. In ACM 78 Proceedings, 1978 Annual Conference, Washington, D.C., Dec. 4–6, 1978, vol. 1, pp. 111–115.
11. HOPCROFT, J.E., AND ULLMAN, J.D. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.
12. JENSEN, K., AND WIRTH, N. *PASCAL User Manual and Report.* Springer-Verlag, New York, 1974.
13. JOHNSON, S.C. YACC: Yet Another Compiler-Compiler. In UNIX Programmer's Manual, 7th ed. Bell Laboratories, Murray Hill, N.J., Jan. 1979.
14. LAZAR, D.S. The design of a hardware compiler: A compiling dedicated machine. M.Sc. thesis, Dep. of Computer Science, Univ. of Toronto, Canada, 1980.
15. MCKENZIE, P., SPINNEY, B., AND WU, T. S/SL parses the LR(k) languages. Tech. Note CSRG-21, Computer Systems Research Group, Univ. of Toronto, Canada, Sept. 1980.
16. MICELI, J. Some experiences with a one-man language. Tech. Note 13, Computer Systems Research Group, Univ. of Toronto, Canada, Dec. 1977.
17. ROSSELET, J.A. PT: A Pascal subset. Rep. CSRG-119, Computer Systems Research Group, Univ. of Toronto, Canada, Sept. 1980.
18. SPINNEY, B. A technique for portable production quality coders. M.Sc. thesis, Univ. of Toronto, Canada, 1981.
19. WILCOX, T.R. Generating machine language for high-level programming languages. Ph.D. dissertation, Cornell Univ., Ithaca, N.Y., 1971.

20. WORTMAN, D.B., AND CORDY, J.R. Early experiences with Euclid. In Proceedings of the 5th International Conference on Software Engineering, San Diego, March 1981.
21. WORTMAN, D.B., HOLT, R.C., CORDY, J.R., CROWE, D.R., AND GRIGGS, I.H. Euclid—A language for compilng quality software. In Proceedings, 1981 National Computer Conference, Chicago, Ill., May 4–7, 1981. AFIPS Conference Proceedings, vol. 50. AFIPS Press, Arlington, Va., 1981, pp. 257–263.

Received December 1979; revised August 1981; accepted October 1981