

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 124

The Snocone Programming Language

Andrew Koenig

19 August 1986

The Snocone Programming Language

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

SNOBOL4 is a well-known programming language with convenient semantics and clumsy syntax. Following the pattern of Ratfor and EFL, we have added “syntactic sugar” to SNOBOL4, with an eye toward making it easier to use. We call the result *Snocone*.

We describe the Snocone language in enough detail that people not familiar with SNOBOL4 can learn to use it, although previous familiarity with SNOBOL4 will help.

This paper originally appeared in the proceedings of the USENIX conference in Portland, Oregon in June 1985.

19 August 1986

The Snocone Programming Language

Andrew Koenig

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

The semantics of the SNOBOL4 programming language¹ include such unusual and useful features as dynamic typing, a general-purpose garbage collector, excellent character string facilities, associative arrays, and strong run-time diagnostics. Griswold,² Gimpel,³ and others have documented many ways of doing things in SNOBOL4 that can only be accomplished with much more difficulty in other languages.

Unfortunately, the control structures of SNOBOL4 are archaic, and the fact that blank is an operator tends to encourage certain types of errors that are hard to detect. Other aspects of the language make it a nuisance to construct large programs out of small parts.

To ameliorate some of these problems, we have designed and implemented a new language that provides some syntactic sugar for SNOBOL4. The obvious name for such a language is *Snocone*.

The design of the Snocone language was inspired by Ratfor⁴ and EFL.⁵ Like EFL, and unlike Ratfor, Snocone is a self-contained programming language, rather than a proper superset of SNOBOL4. Like Ratfor, and unlike EFL, the Snocone translator makes no attempt to produce SNOBOL4 output that is easy for humans to understand.

Hanson⁶ has written a similar, but simpler preprocessor for SNOBOL4. His preprocessor is like Ratfor in that it does not change the statements in the basic language but rather adds new syntax.

Griswold⁷ has written a preprocessor for a language similar to Snocone, the syntax of which is based on Icon⁸ rather than on C or EFL. His preprocessor is written in C, and uses Yacc for its parsing.

In contrast, Snocone has a syntax roughly based on C and is written in Snocone.

What's nice about SNOBOL4

Variables in SNOBOL4 are dynamically typed: the type of any variable is the type of the value most recently assigned to it. Thus, declarations are unnecessary, and, in fact, SNOBOL4 has no declarations as such (except that procedures can have local variables).

All operators and built-in functions check their argument types; each argument is converted automatically to an appropriate type. A run-time diagnostic message results if a conversion is impossible. For instance, if an addend is a string, SNOBOL4 will try to convert it to an integer or real number, depending on the form of its value. If the value is inappropriate to convert to a numeric type, the program will halt.

Like Lisp, and unlike most other languages, SNOBOL4 does not have any explicit mechanism for returning memory to the system. Rather, the implementation must detect when memory is no longer needed and make that memory available for re-use. This makes the language harder to implement, but easier to use. One nice by-product of this sort of memory allocator is that SNOBOL4 programs tend to be free of arbitrary size limitations.

One SNOBOL4 datatype is the *pattern*, which describes an (arbitrarily complex) set of character strings. Briefly, the language incorporates a general top-down backtracking parser. This parser is so easy to use that it is the usual way of dealing with strings in SNOBOL4.

Another useful datatype is the *table*. A table is like a one-dimensional array, except that its subscripts are not limited to being integers. For instance, a compiler symbol table might be maintained as a SNOBOL4 table, with the subscript being the identifier and the value being some structure which holds the desired information about that identifier.

An interesting and unusual semantic idea in SNOBOL4 is that of *statement failure*. Many operations can easily encounter conditions that preclude their execution. For example, an attempt may be made to access a non-existent array element, read beyond the last record of a file, search a string for a pattern that it does not contain, or even make a comparison that gives an unexpected result. When this happens, the operation *fails*. Usually, the failure of an operation implies the failure of the statement of which it is a part. While statement failure is not an error condition, it can be tested. In fact, conditional transfer on the success or failure of a statement is virtually the only kind of control structure in SNOBOL4.

A good implementation of SNOBOL4 is Macrospitbol, by Dewar and McCann.⁹ This is a portable threaded-code interpreter, which is fast and robust enough to be used for “production” purposes. For example, on a VAX-11/780 it translates about 150 statements per second and executes about 5,000.

SNOBOL4 implementations in general have excellent run-time diagnostic facilities. The language definition includes ways of tracing programs for debugging, and of trapping almost any kind of run-time error.

What’s not nice about SNOBOL4

Here is a simple SNOBOL4 program to add the integers between 1 and 1000:

```
sum = 0
term = 1
loopsum = sum + term
term = term + 1
LE(term,1000)      :s(loop)
OUTPUT = "The sum is " sum
END
```

The first two lines of this program assign values to variables. The name *loop* in the third line is a label; it is recognized as such because it begins the line without any white space ahead of it.

The fifth line calls a built-in function to compare the value of the variable *term* to 1000. SNOBOL4 has no comparison operators: all comparisons are done by *predicate* functions that either return the null string if the condition being tested is true or fail if the condition is false. The `:s(loop)` at the end of the line is a conditional **go to**: it causes control to transfer to the label *loop* if the statement succeeds, and to the following statement if it fails.

In contrast, the program looks like this in Snocone:

```
sum = 0
term = 1
do {
    sum = sum + term
    term = term + 1
} while (term <= 1000)
OUTPUT = "The sum is " && sum
```

In the past fifteen years, the trend in programming language design has been overwhelmingly toward programming languages with control structures that make it reasonable to write programs with at most a very few **go to** statements. SNOBOL4 exhibits almost the exact opposite of this trend: essentially the only control structure in SNOBOL4 is a form of conditional **go to** statement. There is no block structure, and all labels are global. Thus, writing a SNOBOL4 program requires a constant effort to invent new label names, and to choose names that have at least some chance of telling the reader whether their use is local or global.

The programmer’s job is not made any easier by the peculiar way SNOBOL4 handles subroutines. SNOBOL4 subroutines are defined at run time by a built-in function called `DEFINE`. Its argument is a *prototype* of the subroutine to be defined – a character string that describes how the subroutine is to be used. The subroutine’s entry point is the label with the same name. Because the statement bearing this label is not special in any other way, it must be placed where it will not be executed in the ordinary course of events. The call to `DEFINE`, on the other hand, must be executed before the subroutine it defines can be used.

This leads SNOBOL4 programmers into contortions. To keep the DEFINE call near the subroutine body, one must invent yet another label:

```
        DEFINE("square(x)")      : (square.end)
square  square = x * x          : (RETURN)
square.end
```

Alternatively, one can put all the DEFINE calls in one place, at the cost of moving the body of each subroutine arbitrarily far from where its prototype appears.

The Snocone programmer has an easier job:

```
procedure square (x) {
    return x * x
}
```

Motivation

Snocone's purpose, then, is to make it easy for a programmer used to a block-structured language like C to write programs that have the freedom and semantic flexibility of SNOBOL4. To this end, we have changed the SNOBOL4 language in several ways.

First, we introduced an explicit concatenation operator. SNOBOL4 uses blank for concatenation, so

```
y = f(x)
```

is a function call, but

```
y = f (x)
```

assigns to `y` the concatenation of the values of the variables `f` and `x`. We chose `&&` to represent concatenation because SNOBOL4 programs often use concatenation for the same purpose that C programs use `&&`.

Second, we allow much the same freedom with spaces that C does, with one exception: newline ends a statement. Statements may be continued onto multiple lines in much the same way as in EFL programs: a line is continued if it ends with an operator or some kind of open bracket.

Third, we have added procedure and structure declarations. These things are accomplished in SNOBOL4 by calling built-in subroutines, and the context of the calls is often obscure.

Finally, we have introduced control structures similar to those in C and other block-structured languages: the `if`, `while`, `for`, and `do` statements.

LANGUAGE DESCRIPTION

Lexical Conventions

Blanks (spaces and tabs, but not newlines) may not appear within a token, but may freely separate tokens. Comments begin with a `#` character and end at the end of the line.

Constants may be integers, reals, or strings. There are no signed numeric constants. Integers range from 0 to $2^{31} - 1$. Real constants are short-precision only, and must contain either a decimal point or an `E` (or `e`). String constants may be delimited by single or double quotes with the same meaning. Characters in quotes receive no special interpretation. A single quote may appear in strings delimited by double quotes, and vice versa.

Identifiers may be of any length. All characters in an identifier are significant, but their case is presently not significant. Case may become significant in the future. An identifier is a non-empty sequence of letters and digits, beginning with a letter. Underscore (`_`) is a letter. The same identifier may be used independently to represent a procedure, label, or variable.

Statement Separation

Snocone delimits statements much like the Shell¹⁰ (the command interpreter in the UNIX® operating system)¹¹ or EFL. A statement is ended either by a semicolon or newline, except that if the last token on a line is an operator or open parenthesis or bracket, the next line is automatically considered as part of the current statement. Newlines may also separate clauses of compound statements, so

```
if (a < 0)
    a = 0
```

is acceptable and means the same as

```
if (a < 0) a = 0
```

or, spreading things as much as possible:

```
if (
a <
0)
a =
0
```

File Inclusion

The contents of an arbitrary file can be incorporated into the program by writing an *include line* in one of the following four forms:

```
#include "file"
#include 'file'
#include <file>
#include {file}
```

Spaces may appear between # and include. The contents of the named file are substituted for the include line.

The exact behavior of an include line depends on the delimiters surrounding the file name. If quotes are used (single or double), the file is sought in the current directory. If brackets are used (angle brackets or curly braces), the file is sought in an installation-dependent system library (*/usr/lib/snocone* on our systems). If double quotes or angle brackets are used, the file is included unconditionally, even if the same file is included several times. If single quotes or curly braces are used, the file will only be included once, even if it is named in several include lines.

This latter behavior is useful in the following sort of situation. Suppose that procedure *proc1* is defined in file *proc1.h*, and procedure *proc2* is defined in file *proc2.h*. A program that calls both *proc1* and *proc2* might then contain:

```
#include "proc1.h"
#include "proc2.h"
```

Now, suppose that *proc1* is changed to call *proc2*. If *proc1.h* is made to contain

```
#include "proc2.h"
```

then our hypothetical sample program will wind up with two copies of *proc2.h* and will therefore run into trouble with duplicate procedure definitions. If, however, *proc1.h* contains:

```
#include 'proc2.h'
```

and the main program contains:

```
#include 'proc1.h'
#include 'proc2.h'
```

then *proc2* will be defined only once.

Expression Evaluation, Success, and Failure

Evaluating an expression has one of three outcomes: a value, failure, or error.

Errors normally result in a diagnostic message and termination of the program, and arise from the usual sorts of things: overflow, underflow, division by zero, impossible conversions, running out of memory, and so on.

Failure, on the other hand, is not an error condition. An expression that fails is merely one that does not yield a value. Examples of expressions that fail include attempts to refer to non-existent array elements, attempts to read beyond end of file, and even comparisons that yield unexpected results. For instance, the expression

```
a < b
```

yields a null string if *a* is indeed less than *b*, and fails otherwise.

Most operators fail if any of their operands fails. The few exceptions will be noted below.

An expression that always either yields a null string or fails is sometimes called a *predicate*. Testing such expressions for success or failure in *if*, *while*, and *do* statements is the primary way of affecting the flow of control in Snocone.

Data Types, Declarations, and Scope

Variables in Snocone are dynamically typed: a variable has the type of the value most recently assigned to it. Except for a few predefined variables (described under *pattern matching*), all variables have the null string as their initial values. All variables are global, but procedures can nominate variables whose values will automatically be saved at entry and restored at exit. The only declarations define structures:

```
struct cons {car, cdr}
```

In effect, this declaration defines three procedures named *cons*, *car*, and *cdr*. The value of *cons(a,b)* is a newly-created *cons* object with *a* and *b* as the values of its *car* and *cdr* fields, respectively. The fields, in turn, are accessed by similarly-named functions. For instance, the *cdr* field of *cons* structure *x* is accessed as *cdr(x)*. Thus this program:

```
struct cons {car, cdr}
a = cons (3, cons (4, 5))
OUTPUT = car (cdr (a))
```

prints 4. The procedures corresponding to field names may be used as the target of an assignment:

```
car (a) = "Hello"
```

Snocone offers other data types than those described above. While numeric constants cannot be negative, variables and expressions suffer from no such restriction. Strings may be of any length up to an implementation-defined upper bound, usually many thousands of characters. All variables have the null string as their initial value.

Aggregate values come in two types: arrays and tables. Each type is created by a built-in procedure with the same name. Thus:

```
a = ARRAY (20)
```

creates a 20-element array, initializes each element to the null string, and assigns it to *a*. The second argument to the array procedure is an initializing value:

```
a = ARRAY (20, -1)
```

makes a an array of 20 elements, each with value *-1*. To get multi-dimensional arrays, express the dimensions as a string:

```
a = ARRAY ('4,5', -1)
```

One can also give explicit lower bounds:

```
a = ARRAY ('-10:10')
```

The default lower bound is 1.

Array elements are referenced by Algol-like subscripts:

```
a[i,j] = a[i,j] + b[i,k] * c[k,j]
```

Each array element behaves as a variable, and can therefore have a value of any data type, independently of any other element.

A table is like a one-dimensional array whose subscripts are not restricted to integers:

```
t = TABLE (20)
```

The argument to the TABLE procedure is an estimate of the maximum number of elements that will actually be stored in the table. If substantially more elements than this are stored, access will begin to slow down. On the other hand, giving too large an initial value wastes space. Once a table has been created, any value can be used for a subscript. `t[a]` and `t[b]` will refer to the same element if and only if `a` and `b` are identical. The precise meaning of "identical" is given with the description of the `::` and `:::` operators; suffice it to say that `3` and `"3"` are not identical, and that after executing

```
a = b
```

`a` and `b` are identical regardless of what values they had before.

Binary Operators

The following operators are grouped in order of decreasing priority. Unless otherwise stated, they are left-associative.

- . \$ Pattern value assignment (see Patterns)
- ^ Exponentiation (right-associative). The right argument must be an integer. If both arguments are integers, the right argument must be non-negative
- * / % Multiplication, division, and remainder. As in C, and as not in SNOBOL4, multiplication and division have the same precedence.
- + - Addition and subtraction.
- == != < > <= >= ::= :!=: :<: :>: :<=: :>=: :: :!:
- Comparison predicates. Each of these operators returns a null string if the indicated relation holds, and fails if not. The first six do numeric comparisons: an error results if either operand cannot be converted to a number. The next six do string comparisons: an error results if either operand cannot be converted to a string. The last two test if the two operands are identical. Values of different data types are never identical. Strings and numbers are identical if their data types and values match. Other values are identical if they refer to the same object.
- && Concatenation. The left operand is evaluated first; if its evaluation fails, the && operator fails. Otherwise, the right operand is evaluated, and && fails if the right operand fails. If either operand is the null string, the result is the other operand, even if that operand is not a string. Otherwise, both operands are converted to strings and the result is their concatenation. An error results if either operand cannot be converted to a string. Note that if the operands of && are predicates, && can be used as a kind of logical conjunction.
- || Logical disjunction. The left operand is evaluated first; if it succeeds, its value is the value of ||. Otherwise, the value is that of the right operand. If both operands fail, || fails.
- | Pattern alternation. See *pattern matching* for details.
- = Assignment. This operator is right-associative.
- ? Pattern match operator. The left operand is converted to a string and searched for the first substring that matches the pattern given by the right operand. If no such substring is found, the operator fails. If the left operand is a variable, ? may be used on the left side of an assignment.

Unary Operators

Unary operators bind more tightly than all binary operators.

- + - Unary plus and minus. The result is always numeric, so unary plus is sometimes used for type conversion. Because unary operators bind so tightly, expressions that look like negative constants behave that way for all practical purposes.
- . Name operator. The operand must be a variable; the result is essentially a pointer to the variable, similarly to the unary & operator in C. The result of applying the DATATYPE function to the result of the . operator is the string "NAME".
- \$ Indirection. The operand is converted to a name; the result is the object thus named. \$ always yields an lvalue.
- ? Query. If its operand fails, ? fails. If its operand succeeds, ? yields a null string. Useful for evaluating an expression solely for its side effects.
- ~ Logical negation. If its operand fails, ~ yields a null string. If the operand succeeds, ~ fails.
- & Keyword value. The operand must be the name of one of a restricted set of variables. The lvalue result is a system variable, whose value affects the execution of the program in some way. System variables are discussed separately.
- @ Pattern cursor assignment. See *pattern matching* for details.
- * Deferred evaluation. Returns a value of type EXPRESSION that contains all the information necessary to evaluate the operand. The operand is not actually evaluated at this time, but can be evaluated later, either by the EVAL procedure or implicitly during pattern matching.

Statements

Elements in brackets are optional. If the description of a statement is split over more than one line, the statement itself may be split analogously. Snocone does not have a null statement.

expression

The expression is evaluated for its side effects. The result, if any, is discarded.

```
if (expression)
    statement1
[ else
    statement2 ]
```

The parenthesized *expression* is evaluated. If it succeeds, *statement1* is executed, otherwise *statement2* is executed.

```
while (expression)
    statement
```

Behaves similarly to C: the *expression* is evaluated, and if it succeeds, the *statement* is executed and control passes back to the beginning of the while statement. Unlike C, Snocone has no *break* or *continue* statements.

```
do
    statement
while (expression)
```

Behaves similarly to C: the *statement* is executed, then the *expression* is evaluated,

and if the *expression* succeeds, control passes back to the beginning of the do statement.

```
for (expression1, expression2, expression3)  
  statement
```

Equivalent to:

```
  expression1  
  while (expression3) {  
    statement  
    expression2  
  }
```

```
{  
  statement list  
}
```

The statements in the list, which may contain zero or more statements, are executed in sequence.

label: *statement*

All labels are global (because all SNOBOL4 labels are global), even across procedure boundaries, so they must be chosen with care. A useful convention is to begin a label inside a procedure body with the name of the procedure and an underscore.

go to *label*

The space between `go` and `to` is optional. It is wise not to jump from a point inside one procedure into another. Program execution may be terminated by jumping to the reserved label `END`. Labels `RETURN`, `FRETURN`, `NRETURN`, `ABORT`, and `CONTINUE` are also reserved.

return [*expression*]

The current procedure returns to its caller. If the *expression* is given, the procedure yields that value. If no *expression* is given, the value returned is that of the variable with the same name as the procedure; if that variable was not assigned in the procedure, the null string is returned.

freturn

The current procedure returns and fails.

nreturn [*expression*]

The current procedure returns $\$(*expression*)$ as an lvalue. If the *expression* is not given, the indirection is applied to the variable with the same name as the

procedure. An error results if this variable was not given a value inside the procedure.

Procedures

Here is an example of a procedure declaration:

```
procedure gcd (m, n) {  
    while (m != n) {  
        if (m > n)  
            m = m % n  
        else  
            n = n % m  
    }  
    return m  
}
```

Arguments are passed by value, but note that the value passed for an aggregate argument (array, table, or structure) is really a pointer to the aggregate itself.

If a procedure is called with too few arguments, extra null strings are supplied as necessary. If called with too many arguments, the extras are quietly ignored.

Local variables can be nominated for a procedure:

```
procedure f(x) y, z {...
```

All variables are global, but name scoping is dynamic. One way to look at it is to imagine that when a procedure is entered, all the variables defined in that procedure are saved and set to null. Those variables are then restored when the procedure returns.

Thus, the following example prints 5 and then 1:

```
a = 1  
f()  
g()  
  
procedure f() a {  
    a = 5  
    g()  
}  
  
procedure g() {  
    OUTPUT = a  
}
```

The following example also prints 5 and then 1:

```
a = 1
b = .a
f()
g()

procedure f() a {
    a = 5
    g()
}

procedure g() {
    OUTPUT = $b
}
```

Local variables can only be associated with procedures. Procedures are recursive.

Input-Output

All I/O is done through “associated variables”. A variable may be input-associated or output-associated (or both). Whenever a value is assigned to an output-associated variable, that value is automatically written in the file associated with that variable. Whenever a value is requested for an input-associated variable, a line is read from the file associated with that variable, and the contents of the line are used for the value. When the end of an input file is reached, any attempts to access variables associated with that file will fail.

Initially, the variable `OUTPUT` is output-associated with the standard output file, the variable `INPUT` is input-associated with the standard input file, and the variable `TERMINAL` is both input- and output-associated with the user’s terminal.

Thus, the following program copies its standard input to its standard output, a line at a time:

```
while (OUTPUT = INPUT) {}
```

New associations are formed by the `INPUT` and `OUTPUT` procedures:

```
INPUT (name, channel, file)
OUTPUT (name, channel, file)
```

In both cases, *name* is the name of the variable to be associated (the name of the variable *x* is `.x`), and *file* is the file to be used. *Channel* is a string that you will use to identify subsequent operations on that file. Internally, there is a one-to-one correspondence between channel names and file descriptors.

The file argument must not be given for other than the first call to `INPUT` or `OUTPUT` on a given channel, as a channel can only be connected to a single file.

Other procedures dealing with input-output are:

```
SET (channel, offset, whence)
```

This function repositions the file specified by its first argument in a system-dependent manner. In implementations running under the UNIX system, the behavior is similar to the *lseek* system call.

```
REWIND (channel)
```

Repositions the named channel to the beginning of the file.

ENDFILE (*channel*)

Indicates that you are done using the given channel. All variables associated through that channel are disassociated, output buffers are flushed (if any), and the file is closed.

DETACH (*name*)

Disassociates the named variable. Does not close the file: a later call to INPUT or OUTPUT can reassociate it.

Pattern Matching

A *pattern* is a data structure that describes a class of strings. Patterns are used by the ? operator, which determines if the string given as its left operand contains a substring described by the pattern given as its right operand. The part of the Snocone system that does this is called the *scanner*. The input to the scanner is the string to be searched, called the *subject*, and the pattern sought.

The scanner tries to find a substring of the subject that is matched by the pattern. It first looks at substrings starting at the first character of the subject. If it doesn't find one, it tries the ones starting at the second subject character, and so on. If the scanner cannot find an appropriate substring, the pattern match fails.

If the &ANCHOR system variable is nonzero, the scanner only looks at substrings that start at the first character of the subject. This is said to be an *anchored* pattern match. The &ANCHOR variable is zero at the start of program execution.

The important part of pattern matching is therefore determining whether the subject contains a substring that starts at a given character and matches a given pattern. To understand how this is done, we must take a closer look at patterns.

Every pattern is a concatenation of one or more *elements*, each of which has zero or more *alternatives*. Each alternative may itself be an arbitrarily complicated pattern. Some patterns have alternatives that are determined dynamically as they are needed during pattern matching.

If a pattern has only one element, the scanner determines if it matches at a given point by trying to match each of the pattern's alternatives at that point. If no alternative matches at that point, the element cannot be matched.

If the pattern has more than one element, matching that pattern at a given point means: (a) finding an alternative for the first element of the pattern that matches a subject substring starting at the given point, and that also (b) allows the remaining elements of the pattern to match a substring that starts immediately after the substring matched in (a).

During pattern matching, the scanner keeps its place by means of an internal value called the *cursor*, which represents the number of characters in the subject that precede the current location. These characters are counted from the beginning of the subject even when the scanner is trying to match a substring that starts at some other place in the subject.

The concatenation of two patterns P1 and P2 is a pattern whose elements are P1 and P2. The alternation operator | similarly constructs alternatives. When a string is used as a pattern, it matches only itself. Thus,

```
"a" | "e" | "i" | "o" | "u"
```

is a pattern that matches any lower-case vowel.

Consider the following statement:

```
P1 = ("ab" | "a") && ("b" | "c")
```

This assigns to P a pattern with two elements. The first one matches either ab or a, and the second matches either b or c. Look at:

```
"ab" ? P1
```

The scanner first tries the first alternative of the first element of P1 by matching ab in the subject with ab in the pattern. This alternative matches, so the element ("ab" | "a") matches, and the scanner goes on to the second element ("b" | "c"). Here, it will be unsuccessful in matching both b and c, because it has already exhausted all the characters of the subject. Thus the scanner fails to match the second element of P1, and must back up and rematch the first element. Fortunately, the first element has an alternative in a; this matches, and now the scanner can try to match the second element ("b" | "c") again. The first alternative (b) succeeds, so the ? operator therefore succeeds. If we wanted to examine just how the pattern elements matched, we could have written:

```
P1a = ("ab" | "a") . OUTPUT && ("b" | "c") . OUTPUT
"ab" ? P1a
```

This would print a and b on separate lines, showing that "ab" | "a" matched a and "b" | "c" matched b.

When any pattern is first encountered during pattern matching, it will match some string, and when the scanner backs into it, it may match some other string. Because many patterns behave differently on their initial match than when rematched, it is useful to describe the two cases separately.

For instance, when a string is used as a pattern, it initially matches itself. If the scanner later backs into it, it fails.

As another example, consider P1 | P2. This pattern matches every possibility for P1, and when it has exhausted P1, then matches every possibility for P2 before finally failing.

With this in mind, we can describe the various pattern-matching operators, procedures, and pre-defined variables:

- P1 && P2 Tries to match P1, fails if it can't. Then tries to match P2. If P2 fails, tries the next alternative for P1, and then tries P2 again. Eventually, it either runs out of alternatives for P1, in which case P1&&P2 fails, or it finds alternatives for both P1 and P2 which allow them to match.
- P1 | P2 Tries to match P1. If successful, P1 | P2 matches the string that was matched by P1. Otherwise, matches whatever P2 matches, and fails if P2 fails.
- P \$ V Tries to match P. If P matches, a copy of the substring matched by P is immediately assigned to the variable V.
- P . V Tries to match P. If the entire pattern match of which this element is a part is ultimately successful, a copy of the substring matched by P is assigned to V after the entire match completes.
- @N Matches a null string, and immediately assigns cursor position to the variable N. The cursor position is the number of characters that precede this null string in the subject of the entire pattern match. Thus:

 "abcde" ? @OUTPUT && "c"

 prints 0, 1, and 2 on separate lines.
- ABORT The entire pattern match is aborted.
- ARB A pre-defined variable that matches anything at all. More specifically, it initially matches the null string. When backed into, it matches a string one character longer than the one it matched last time.
- BAL A pre-defined variable that matches a non-empty parenthesis- balanced string. This string is balanced only with respect to parentheses, and not any other kinds of brackets.
- FAIL A pre-defined variable that always fails to match. It is sometimes useful to force

the scanner to try all alternatives for a pattern. For instance:

```
s ARB $ OUTPUT && FAIL
```

prints all substrings of *s*.

FENCE	Initially matches the null string, but if the scanner backs into it, the entire pattern match is aborted. Identical to " " ABORT.
REM	Matches from the current position to the end of the subject. Identical to RTAB(0).
SUCCEED	Identical to ARBNO(" ")
ANY(<i>s</i>)	Matches any single character in <i>s</i> .
ARBNO(<i>p</i>)	A pre-defined procedure that yields a pattern that matches zero or more copies of the pattern <i>p</i> . Initially matches the null string. Each time the scanner backs into it, it tries to extend the substring already matched by matching one more instance of <i>p</i> .
BREAK(<i>s</i>)	Matches a string starting at the current position up to but not including the next character in the subject that is also found somewhere in <i>s</i> . Failure if no such character is found. No alternatives.
BREAKX(<i>s</i>)	Like BREAK, but if backed into, it matches up to the next subject character also found in <i>s</i> , and so on.
LEN(<i>n</i>)	Matches exactly <i>n</i> characters.
NOTANY(<i>s</i>)	Matches any single character not in <i>s</i> .
POS(<i>n</i>)	If exactly <i>n</i> subject characters precede the current position, POS matches the null string. Otherwise it fails.
RPOS(<i>n</i>)	If exactly <i>n</i> subject characters remain to be matched, RPOS matches the null string. Otherwise it fails.
RTAB(<i>n</i>)	If at least <i>n</i> subject characters remain in the subject, RTAB matches characters until exactly <i>n</i> remain. Otherwise it fails.
SPAN(<i>s</i>)	Starting at the current position, matches as many characters as possible taken from <i>s</i> .
TAB(<i>n</i>)	TAB matches from the current position forward up to and including the <i>n</i> th character of the subject. If more than <i>n</i> characters have already been matched in the subject string, TAB fails.

All pattern-valued procedures can take an unevaluated expression as argument. The expression will be evaluated when the pattern element is matched.

System Variables

The & operator looks at the name of its operand, not its value. For each of a small set of names, & yields a *system variable*, whose value affects the operation of the system in some way. For instance, we have already seen &ANCHOR, which controls whether or not the scanner is restricted to examining initial substrings of the subject. The complete list of system variables is:

&ABORT	The same value as the pre-defined variable ABORT.
&ALPHABET	A string that contains all the characters of the machine's collating sequence, in order.
&ANCHOR	If zero, all pattern matches are unanchored, otherwise they are anchored.
&ARB	The same value as the pre-defined variable ARB.
&BAL	The same value as the pre-defined variable BAL.
&CODE	This variable is initially zero. Its value is returned to the operating system when the program finishes executing.

&DUMP	This variable is initially zero. If it is 1 at the end of execution, the values of all variables are printed. If it is 2, the values of all array, table, and structure elements are also printed.
&FAIL	The same value as the pre-defined variable FAIL.
&FENCE	The same value as the pre-defined variable FENCE.
&FNCLLEVEL	The current level of procedure nesting.
&INPUT	If this variable is set to 0, all input association is suspended.
&MAXLNPTH	The maximum length of a string. This value cannot be increased beyond its initial value.
&OUTPUT	If this variable is set to 0, all output is suppressed until it is again set nonzero.
&REM	The same value as the pre-defined variable REM.
&STCOUNT	A count of how many statements have been executed so far. Because this counts SNOBOL4 statements, not Snocone statements, it should be considered only approximate.
&STLIMIT	When &STCOUNT reaches this value, execution terminates with an error message. It is initially 50000, and may be set freely. If it is set to a negative value, statement counting is disabled.
&SUCCEED	The same value as the pre-defined variable SUCCEED.

Examples

Hello World

This is the canonical sample program:

```
OUTPUT = "Hello world!"
```

The present implementation of Snocone is case-insensitive in identifiers, but future versions may well become case-sensitive. If so, it is likely that pre-defined names, such as INPUT and OUTPUT, will have to be written in upper case. Keywords, such as if and while, will be written in lower case.

Topological Sorting

We now develop a topological sorting program based on the algorithm described on page 262 of *Fundamental Algorithms*.¹² The reader may wish to compare this program with the SNOBOL4 program based on the same algorithm that appears on pages 221-222 of *The SNOBOL4 Programming Language*.

The input is a set of pairs of objects, where the first object in each pair is considered to precede the second. The output is a list of objects in a sequence that meets all the constraints implied by the input. In other words, the program generates a total ordering that includes a given partial ordering.

If the input contains a loop, the program will detect this fact and complain.

For example, given the following input:

```
letters alphanum
numbers alphanum
blanks optblanks
numbers real
numbers integer
letters variable
alphanum variable
binary binaryop
blanks binaryop
unqalphabet dliteral
unqalphabet sliteral
sliteral literal
dliteral literal
integer literal
real literal
```

the program will produce the following output:

```
letters
numbers
blanks
binary
unqalphabet
alphanum
real
integer
optblanks
binaryop
dliteral
sliteral
variable
literal
```

The basic strategy of the program is simple: for each object, we remember how many immediate predecessors it has, and store a list of all its immediate successors. When we have finished reading the input, we can immediately output the objects that have no predecessors. Each time we output an object, we remove it from the data structure and decrement the predecessor count of each of its immediate successors.

We will eventually reach a state in which we run out of objects without predecessors. When that happens, we are done. If any objects remain, they form a loop.

To reduce the time spent searching for objects without predecessors, we keep a queue of such objects. We must also keep a queue of the successors of each object. In both cases, we could use a stack instead of a queue, but using a queue tends to favor the order in which objects appeared in the input, which makes the output more intuitively useful.

The following structure declarations and subroutines manipulate queues. A queue consists of a header (of type `queue`) which points to the first and last elements of a singly-linked list of queue elements (of type `qel`).

```
struct queue {head, tail}
struct qel {obj, link}

procedure enqueue (q, x) y {
  if (head(q) :: "")
    head(q) = tail(q) = qel(x)
  else {
    y = qel(x)
    link(tail(q)) = y
    tail(q) = y
  }
}

procedure dequeue (q) x {
  if (head(q) :: "")
    freturn
  x = head(q)
  if ((head(q) = link(x)) :: "")
    tail(q) = ""
  return obj(x)
}
```

By convention, we use the null string to indicate the end of a list. This is convenient because uninitialized variables and structure fields and missing arguments are automatically set to the null string. Thus, the test

```
head(q) :: ""
```

is a convenient way of testing whether `head(q)` has been set or not.

We represent each object as a structure containing the object's name (so we can print it), the count of immediate predecessors, and the queue of successors:

```
struct object {name, count, suc}
```

Since we will be reading the names of objects rather than the objects directly, we will need to map names to objects. This can easily be done with a table and a mapping subroutine that creates elements in the table as needed:

```
namemap = TABLE()
objects = queue()

procedure getobj (name) {
  if ((getobj = namemap[name]) :: "") {
    getobj = namemap[name] = object (name, 0, queue())
    enqueue (objects, getobj)
    nobj = nobj + 1
  }
}
```

This procedure uses the feature that if no return value is explicitly given, the value of the variable with the same name as the procedure is used. If an appropriate table element already exists, the `::` operator will fail and the value of `getobj` will be the value retrieved from the table. As a side effect, we maintain a global queue of all known objects in `objects` and count them in `nobj`.

Now that we can map from names to objects, it is an easy matter to enter a new relation into our data structure. Procedure `enter` takes the names of two objects:

```
procedure enter (p, q) {
  p = getobj (p)
  q = getobj (q)
  count(q) = count(q) + 1
  enqueue (suc(p), q)
}
```

We first locate the objects to which *p* and *q* refer, creating them if necessary. Since *p* precedes *q*, we increment the predecessor count of *q* and append *q* to the successor list of *p*.

Building the data structure is now just a matter of scanning the input file:

```
while (line = INPUT) {
  if (line ? FENCE && BREAK(' ').p && SPAN(' ') && rem.q)
    enter (p, q)
  else
    TERMINAL = "bad input line: " && line
}
```

The pattern match assumes that everything up to the first blank in the input line is the first object in a relation, and everything after the first blank is the second object. If the match fails, the program complains.

Once all the input has been read, we must initialize the queue of minimal objects (objects without predecessors). This was the reason for keeping a queue of all objects, which is now destroyed to build the queue of minimal objects. It is not necessary to destroy the queue, but it is more convenient because it is then possible to use `dequeue`:

```
zeroes = queue()

while (x = dequeue (objects))
  if (count (x) == 0)
    enqueue (zeroes, x)
```

As long as there is a minimal object, we can print its name, delete it, and decrement the predecessor count of each of its successors. If we decrement a predecessor count to zero, that object is now minimal.

```
while (x = dequeue (zeroes)) {
  nobj = nobj - 1
  OUTPUT = name(x)
  while (y = dequeue (suc (x))) {
    if ((count(y) = count(y) - 1) == 0)
      enqueue (zeroes, y)
  }
}
```

This loop runs until there are no more minimal objects. If there are still elements remaining (`nobj` is nonzero), then those elements form a loop.

```
if (nobj != 0)
  TERMINAL = "The ordering contains a loop."
```

References

1. R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language (second edition)*, Prentice-Hall, Englewood Cliffs, New Jersey (1971).
2. R. E. Griswold, *String and List Processing in SNOBOL4; Techniques and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey (1975).

3. J. F. Gimpel, *Algorithms in SNOBOL4*, John Wiley and Sons, New York, New York (1976).
4. B. W. Kernighan, "RATFOR -- A Preprocessor for a Rational Fortran," *Software Practice & Experience* **5**(4), pp. 395-406 (October, 1975).
5. Stuart I. Feldman, "The Programming Language EFL," Comp. Sci. Tech. Rep. No. 78, Bell Laboratories, Murray Hill, NJ (June 1979).
6. David R. Hanson, "RATSNO -- An Experiment in Software Adaptability," *Software -- Practice and Experience* **7**, pp. 625-630 (1977).
7. R. E. Griswold, "Rebus -- a SNOBOL4/Icon Hybrid," Technical report TR 84-9, Department of Computer Science, Tucson, Arizona.
8. R. E. Griswold, *The Icon Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1983).
9. Robert B. K. Dewar and A. P. McCann, "Macro Spitbol -- a SNOBOL4 Compiler," *Software -- Practice and Experience* **7**, pp. 95-113 (1977).
10. S. R. Bourne, "UNIX Time-Sharing System: The UNIX Shell," *Bell Sys. Tech. J.* **57**(6), pp. 1971-1990 (1978).
11. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.* **57**(6), pp. 1905-1929 (1978). Also in *Comm. ACM*, **17**, pp. 365-375 (1974).
12. Donald E. Knuth, *Fundamental Algorithms*, Addison-Wesley (1973). Volume 1 of *The Art of Computer Programming*