SNOBOL4 as a Language for Bootstrapping a Compiler

Richard Dunn

Abstract: The programming language SNOBOL4 is evaluated for the task of bootstrapping the compiler for a new language. Based on the results of an actual project, SNOBOL4 was found to have adequate power to write the bootstrap, but could not produce an acceptably efficient translator.

There are circumstances in the design of a new programming language in which it is necessary to have a working compiler before the language specification is complete, but where this compiler will not be the final "production" compiler. One obvious reason for the existence of such a compiler is to allow the language designer to test language features for ease of use by writing actual programs, and for ease of compilation by actually trying to translate them. Another use is for languages which are suitable for compiler writing and in which the production compiler for the language is to be written in the language itself. The initial compiler is then termed a "bootstrap"; it is this case which we wish to consider. The evaluation of SNOBOL4 as a language for such a bootstrap is based on an actual attempt to bootstrap a compiler for a language called JANUS. The structure and complexity of JANUS closely resemble PASCAL(1), although JANUS has no operator precedence relations. The bootstrap was attempted on a Control Data 6400 computer.

First let us consider the general criteria for producing a bootstrap compiler; we may then evaluate particular reasons for which SNOBOL4 meets or fails to meet these criteria. First, we shall require that the compiler be easily and quickly written. It is important not to get carried away with the construction of this compiler, since it will only be used until the production compiler is operational and then discarded. Related to this is the need for a program which can be easily modified, since the language that we are compiling will probably change as we are writing the bootstrap. The third requirement is that the bootstrap compiler must be reasonably efficient. It will be acceptable if it is several times slower than the projected speed of the production compiler, since the bootstrap will only be in use for a short time. Nevertheless, during that short time, it will have to translate the entire production compiler until it is

working. The production compiler will be a substantial amount of code--say, several thousand lines--and may have to be translated several dozen times before it is healthy enough to translate itself. (It is this last criterion which limits the usefulness of otherwise elegant, powerful schemes for producing experimental compilers. It was also this criterion which brought the downfall of the compiler described in this paper.)

The requirement that the bootstrap be easily and quickly written implies a need for higher-level language features. SNOBOL4 has a convenient and unusually flexible procedure mechanism (in which any line of code may be executed as a part of any procedure), including recursive procedures. The data structuring facilities include dynamically created arrays, tables with automatic lookup on any type of key, and a structure-with-fields facility. By use of the structure facility, trees, branched lists, and recursive structures may be built. A general dynamic storage allocation scheme is built in and completely transparent to the programmer. (At least until he starts running out of storage, but more on that later.) The only feature common in other languages but not present in SNOBOL4, which proved to be irritating at all, was the lack of a conditional of the form:

IF condition THEN option$_1$ ELSE option$_2$

which in SNOBOL4 must be written in one of the forms:

```
        condition   :S(L1)                condition option1   :S(M1)
        option2     :(L2)     or         option2
    L1  option1                      M1  ...
    L2
```

In either case, the programmer must invent a label or two; it can be a nuisance to invent many labels (keeping them unique) in a large program.

Another factor in allowing easy implementation is the existence of good debugging facilities. Again, SNOBOL4 looks good in general, but this time there is a glaring exception. It is possible to trace variables as they change values, and procedures and labels as they are referenced. A post-morten dump labeled with variable names is available. All debugging facilities are at the source language level, making them easy to use. The one flaw is that there is no provision for examining either the structure or the contents of a variable whose value is an instance of a data structure (defined by the DATA function.) The only

information directly available for printing is the name of the overall structure assigned to a variable. In our compiler, almost all variables were structured and we were unable to use many of the debugging facilities. It would be possible to write a structure-dump procedure for a particular program in SNOBOL4, but this is too tedious. By comparison with other debugging facilities of SNOBOL4, it seems unusual that no structure dump is available. Even the diagnostics(perhaps "diagnostic") give practically no information. The usual error detected is an attempt to reference a nonexistent field of a structured value; the diagnostic is "ILLEGAL DATA TYPE." No identifying information about which reference was illegal is given. Consider the following (taken from our compiler):

```
XMODE(X) = XMODE(LEFT(L))  :($RPROC(LAST(X)))
```

Even in this statement, "ILLEGAL DATA TYPE" could mean:

```
L has no LEFT field          LEFT(L) has no XMODE field
X has no XMODE field         X has no LAST field
LAST(X) has no RPROC field
```

If it still seems that this is only one small point, let me add that bugs of this sort, along with the debugging runs to find and correct them, accounted for over fifty runs in about three months.

Another debugging facility which would have been helpful is a provision for declaring the modes of variables, forcing the compiler to check usage of such variables. First, notice that this is not necessarily at odds with the SNOBOL 4 design philosophy of not requiring the programmer to provide declarations; it only allows him to provide declarations as a check against his own carelessness. Second, notice that this provides a more useful check than an execution-time diagnostic about the use of a value of the wrong mode. It allows the error to be detected when the incorrect value is first stored, not later when it is referenced (possibly after being copied about from one variable to another several times.)

The requirement of easy modification of the compiler manifests itself in a need for succinct, lucid ways of expressing the structures and operations of the compiler. In SNOBOL4, the easy manipulation of character strings allows storage and use of constants with mnemonic value such as "INT", "REAL,ARRAY", and "PROC,BOOL,EXT" rather than encoding all such values into integers or bit strings. The free use of data structure field selectors

allows one to follow pointers into a complicated structure in a natural fashion. Modifications to the lexical analyzer of a SNOBOL4-coded compiler are naturally trivial since a strong point of SNOBOL4 is its ability to manipulate strings of characters. The high-level language features mentioned earlier are an asset in making the program easy to modify. For example, the compiler may not use recursive procedures in its initial form, but a change in the syntax may introduce recursion of a particular syntactic element which can best be expressed by simply allowing the code which processes that element to call itself. In our experience, the bootstrap was very easy to modify once the main sections of code were well debugged.

The final criterion was that the bootstrap compiler be reasonably efficient. In a multiprogramming environment (such as the CDC 6400 on which our project was attempted) the program should be efficient with respect both to time and to memory used. Fortunately, SNOBOL4 allows the programmer to have control over the balance here. He may choose to use less memory with a consequent increase in running time due to more frequent garbage collections. Unfortunately, SNOBOL4 tends to use unusually high amounts of storage and time for rather simple tasks. The fact that all variable modes are latent forces run-time checks before every operation, and it requires storage for type information associated with each value. With heavy use of data structures the problem gets worse, since the interpreter must first find each field which is referenced, and additional information must be provided to identify the fields. In a compiler which is constantly generating and discarding structured values, the extra storage used for each structure also becomes apparent in increased execution time--due to more frequent garbage collections. (In our bootstrap compiler, it was not unusual at all to have a garbage collection occurring every five or six lines of code.)

A separate problem of excessive storage use is the lack of a data initialization statement, particularly if the compiler is essentially tabledriven (as ours was.) Tables must be preset by executable code which will stay in memory after it has served its purpose. In our compiler, the initialization code represented about one-fourth of the total number of statements in the compiler, and these statements were by far the most complex due to the number of structure references. This extra code caused a problem which was compounded by the large amount of memory used by the SNOBOL4 interpreter itself. For our bootstrap to run efficiently, it required about three times the memory that the FORTRAN compiler (which itself is somewhat large and not overlaid)

normally uses. Stating it differently, the bootstrap needed about 80% of the available memory in the whole machine to run without too-frequent garbage collections. It would not run at all without at least 67% of the available memory.

SNOBOL4 provides no "object program" format, which required that the compiler be retranslated each time that it was used, using about 17 seconds of CPU time. (With a special assembly-language subroutine, a thorough knowledge of the machine and operating system, and a couple of tricks, one might be able to circumvent this problem.) The initialization code took another two seconds--all before any compilation actually took place. When the bootstrap was finally running, it would translate about four lines of simple code (excluding comments) per second of CPU time. This is at least a factor of 40 slower than the FORTRAN compiler on similar text, and was reached after some careful tuning of certain parts of the bootstrap. We regretfully decided that this was unacceptably slow, showing no prospect of admitting enough optimization to make it acceptable. We were able to make the decision even without considering the compiler's voracious appetite for memory.

I should emphasize that the preceding, rather dismal appraisal applies only to the use of SNOBOL4 for the particular problem of bootstrapping a compiler and not to the general utility of the language. Part of the problem may have been an overly general parsing algorithm. This would not have made a difference in our case, but it does seem to indicate that SNOBOL4 is useful for writing "toy" (non-production) compilers which will only translate a small number of comparatively short programs in their useful lifetimes, either for instructional or for research uses. For the bootstrapping problem, perhaps the reasons for the failure of the SNOBOL4-coded bootstrap will indicate a more suitable language. I feel that the problems can be summed up by stating that SNOBOL4 allows for too much freedom and too many generalities, without treating simple cases specially, to allow it to be made sufficiently efficient to produce a good bootstrap compiler for a practical higher-level language.

1. Wirth, N. "The Programming Language Pascal," Acta Informatica, Vol. 1, Fasc. 1(1971).