SETL Newsletter No. 93 Kent Curtis
January 5, 1973

A Note on Optimization and Programming Style in SETL:

A brief set of experiments described in SETL Newsletter 92 indicates that auxiliary variables and auxiliary objects formed during execution cause SETL program to expand and are costly in terms of both memory and execution time. There are two effects which spring from this same source.

a) Auxiliary variables or objects (e.g. tuples, sets, or strings) use memory and are only removed by garbage collection. Each garbage collection in a simple program required approximately 1.1 seconds and it was found necessary to do 20 - 25 garbage collections in 70 seconds execution time on the programs used in those experiments.

b) Auxiliary variables or objects which are live at any point in the program occupy memory space which is inaccessible for garbage collection and increase the field length required for successful completion of a program.

These observations point toward a type of optimization in SETL programs which is less accessible and less useful for other programming languages and they also suggest some guidelines on programming style in SETL which may help reduce the field length and execution time required for applications written in SETL. They originate from the fact that memory space is time in SETL programming.

Optimization

By reducing the span of a program during which variables are live and by burying dead variables as soon as they die (e.g. by promptly redefining them to be null) the minimum field length required may be reduced and space may be released for garbage collection.

The first process, live span reduction, can be accomplished sometimes by simple rearrangement of code. This is susceptible to automatic analysis using data from live-dead tracing.

The same data can show where funeral rites could be held at the earliest moment for dead variables and an optimizer could provide suitable services.

A third type of useful optimization is to eliminate auxiliary variables by replacing them with equivalent SETL dictions. The analysis of this is a complex semantics and memory space-execution time trade-off problem which is probably outside the range of practical automatic optimization but is an appropriate consideration for programming style.

Programming Style in SETL

These considerations suggest the following guidelines for programming style in SETL:

1. Always postpone expression evaluation until the value is needed.
2. Use the SETLB dictions (functional application, compound operator, etc.) instead of defining auxiliary variables.
3. Kill variables as soon as possible. (Early output of generated results which will not be used again is desirable.)
4. If a variable is dead, bury it promptly.

These rules favor full line coding in SETL but the careful optimization of the SETL primitives through the use of hashing makes this advantageous.