MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 80

SPIL: A Language for System Design and Implementation

by

Barbara H. Liskov

Work reported herein was supported in part by Air Force Contract No. F19(628)-71-C-0002, and in part by the National Science Foundation under research grant GJ-34671.

May 1973

ABSTRACT

Structured programming is gaining wide acceptance as a technique for increasing the reliability and understandability of software. Nowhere is the need for better software felt more keenly than in the area of systems programming. In the near future it is likely that systems will be programmed in structured programming languages, just as systems have been programmed in higher level languages in the past.

This paper describes SPIL: a structured programming language intended to be used for system design and implementation. The paper was not written to introduce yet another programming language; rather it is concerned with (1) showing how the objectives of structured programming and system programming may be realized in a programming language, and (2) evaluating SPIL as a design language. Structured programming, a term with many different interpretations, is defined, and its influence on SPIL carefully delineated. The requirement of system programming is interpreted to mean that the user requires efficient program execution and access to special machine capabilities not ordinarily available in a higher level language. SPIL satisfies the user's needs by mirroring a somewhat unusual, underlying machine architecture; the architecture is briefly described and its effect on the language traced. Desirable properties of a design language are introduced in the form of criteria which a design language should satisfy. The criteria are related to SPIL and structured programming in general.

INTRODUCTION

SPIL (System Programming Implementation Language) is a higher level language intended to support software design and implementation on the Venus machine (1, 2). It belongs to a class of languages such as BLISS (3) which have the control structures of a higher level language but which are machine-dependent in other respects. This type of language is suitable for system programming because it provides many of the advantages of a higher level language (for example, increased programmer productivity), while permitting the user to retain control over machine capabilities which are needed to perform basic system functions.

SPIL was designed to be a tool for a project concerned with evaluating pragmatic techniques for improving the reliability of software. Since SPIL was merely a tool and not a product of the project, constraints were placed on the amount of effort which could be spent in designing and implementing it. For example, it was not feasible to base the design of SPIL on the development of new language concepts, nor could we afford the effort of building an optimizing compiler.

The fact that pragmatic techniques were desired determined the whole approach of the project. Important but impractical approaches, such as proving the correctness of programs, were ruled out. Instead, we selected the "constructive" approach, that is, the combination of programming techniques (in particular, structured programming) and management practices that encourage construction of correct software in the first place. The rationale behind this decision is fully discussed in (4). Our choice was heavily influenced by the success of an IBM project (5) which combined structured programming and management techniques to achieve an impressively reliable system.

We intended to use SPIL, in conjunction with programming and management guidelines developed concurrently with SPIL (6), to construct a complex software system. We had in mind a data or file management system providing controlled sharing of data (along lines suggested by Habermann (7)). The new system was to be constructed in the environment of the Venus Operating System (2). However, when completed it was to be moved to its own computer, and part of the design of the system was to consist of determining what the architecture of the new computer should be. The new architecture was constrained to be Venus-like in nature but was certain to contain new primitives, for example, primitives supporting file access. In addition, the architecture might deviate from Venus, either by extending Venus capabilities (e.g. there might be more processes available on the new machine), or by changing the definitions of certain Venus primitives.

SPIL was designed to satisfy two main criteria: it was supposed to be (1) a structured programming language , and (2) support system programming on Venus. We also definitely intended SPIL to be a design language, that is, a language which expresses the design as it develops in a natural and economic way. This paper discusses how the design of SPIL was influenced by the design criteria, and briefly describes the most important SPIL features. It then concludes by evaluating SPIL as a system design language.

THE VENUS MACHINE ARCHITECTURE

The intention to use a language for writing system programs places special constraints on the language design; it is more than usually important that programs written in the language be efficient, and the user of the language may need access to capabilities of the machine available at the assembly language level but not ordinarily supported by a higher level language. SPIL was designed to satisfy both criteria by reflecting the architecture of the Venus machine. The following are the most important features of the Venus architecture (more detail may be found in (1)):

1) The Venus machine supports segments (named virtual memories). Segments may contain up to 64 K bytes of data and have 15-bit names. The microprogram performs the mapping of segment addresses to core addresses; segments are paged on demand between back-up store and core memory. Venus software (both programs and data) is almost always stored in segments.

2) The Venus machine supports 16 processes. Each process has an in-core work area containing process-related information (the state of the process), and an address space consisting of segments which it may share with other processes. Coordination of sharing and synchronization of processes are supported by semaphores and the two semaphore operations: P and V.

3) The Venus machine has a microprogrammed I/O channel, so that Venus software is not constrained by any real-time requirements of the I/O devices. Semaphores are used to synchronize the termination of I/O. There are no I/O interrupts in Venus.

4) The Venus machine recognizes certain segments as procedure segments. Procedures are intended to be pure (reentrant). The only way in which control may be switched between procedures is via the call and return instructions, which provide a reentrant and recursive procedure interface, saving

and restoring the state of the process at the time of the call in a special "control stack" segment. Arguments and values may be passed in segments which are treated as pushdown stacks, referenced by push and pop instructions.

5) The Venus machine provides a software interrupt structure for instrumentation and debugging. For example, there is an "every instruction interrupt" which, when enabled, causes a specified instruction (generally a call instruction) to be executed before every instruction of the procedure being debugged.

Essential Machine Features

A careful analysis of the Venus machine features was performed, in order to determine how to incorporate them in SPIL. As a result of this analysis, certain features were identified as essential to SPIL and influenced its design directly. The remainder had less effect on SPIL. The essential features include: procedures, allocation of data in segments, and register allocation.

Procedures. Procedures were deemed to be essential on the grounds that they are inherently a higher level language feature; it would have been unnatural to design a higher level language for Venus which did not utilize them directly. In fact, procedures are the basic building blocks from which Venus programs are made. The same is true of SPIL programs, and SPIL is a procedure-oriented rather than a block-structured language.

Segments. Control over the allocation of data in segments was deemed essential to obtaining efficient program execution. Almost all data in Venus is stored in segments. If space for this data is not allocated intelligently, extremely high overhead may result in the form of page flutter. Intelligent allocation involves understanding of the reference patterns to data items, and reference patterns happen at execution time and may span many procedures

since segments can be shared among procedures. Therefore we decided to allow users to specify exactly how data should be allocated in segments, and the concepts of segments, of segments structured to contain data, and of accessing data in segments all became part of SPIL.

Registers. Register allocation was also deemed essential to efficient program execution. Each Venus process has at its disposal 16 general purpose registers (stored in its work area) in which to store information. Information in registers may be accessed much faster than information not in registers. Therefore, the most frequently used information should reside in registers if code is to be efficient.

One way to make sure registers are used effectively is to have them be the main entity which a program manipulates, as is true in machine language and also in PL/360 (8). We rejected such an idea as leading to confusion: programs are really concerned with variables and asking them to be concerned with registers adds to the complexity of the programming task (and is therefore not conducive to the project goal of software reliability). SPIL programs use variables, but include a REG statement to inform the compiler of how to allocate the registers to the variables.

Other Machine Features

All other Venus machine features were deemed inessential to SPIL. Nevertheless, SPIL procedures require access to the features (via the instructions supporting them) if they are to be used to build a system.

If all the useful Venus instructions had been brought into SPIL, the SPIL grammar would have been greatly expanded to contain the new terminal symbols and non-terminals expressing the special requirements of the particular operators (instructions). Also many data types would have been needed since some operators can only operate on special types: e.g. P, V

on semaphores.

Instead, we chose to "insert" machine instructions into SPIL programs via a syntactic rule

< S > :: = $ < machine op > : < first operand >, < second operand >

which permits the compiler to check that the machine instruction indicated by < machine op > is one of the allowable ones, and that the operands, which may be variables or expressions, are suitable for the instruction.

The instructions incorporated under < machine op > are intended to be used infrequently, and with care: the $ is like a warning flag. There are strict constraints on which instructions can appear as < machine op >s. Certain instructions will never be supported -- in particular instructions affecting control: branch, call, return. < machine op > is intended to augment SPIL, not to provide alternative ways of doing things. In addition, there is no provision for insuring contiguity of a sequence of < machine op >s, since a single < machine op > may generate several instructions if necessary to massage the operands into the appropriate forms.

It is interesting that < machine op > provides a mechanism for extending SPIL. For example, instructions added to the new computer to support file management might be used as primitives in SPIL programs (with a $) and their meaning embedded in the compiler. The SPIL programs could then be moved to the new machine without any changes being necessary.

STRUCTURED PROGRAMMING

What is Structured Programming?

The phrase "structured programming" is ambiguous -- almost everyone has a different idea of what it means. Three different meanings may be easily distinguished:

M1) Structured programming is goto free programming (9).

M2) Structured programming is top-down programming (control only)(10).

M3) Structured programming is top-down programming (11, 12).

Only M1 and M2 are really well-defined. M3 is not supported by any existing programming language, as Dijkstra pointed out in his Turing Lecture (13). For one thing, M3 is really a philosophy about how programming ought to be done -- an issue not yet resolved. For another, those parts of M3 which are understood require new programming language constructs for their support, and many programming language issues must be rethought (e.g. block structure (14)).

$$M1 \subset M2 \subset M3$$

in the sense of containing ideas or concepts; the reasons why M1 is necessary in each case are discussed in (6).

Top-down programming (both M2 and M3) involves the following: The first code written is the very "top" of the system or program; it describes the way control flows among the major functional components of the program. The code constitutes a structured program module. The components are represented in the code by writing their module names.

Figure 1 contains an example of a structured program module which is the "top" of an operator precedence compiler. The example, written in an Algol-like notation -- not in SPIL -- uses module names push, finished, scan_next_symbol, precedence_relation, top, and perform_operation_based_on_relation. The module is a complete

```
begin
integer relation;
boolean must_scan;
string symbol;
stack parse_stack;
must_scan := true;
push(parse_stack, eof_entry);
while not finished(parse_stack) do
  begin
  if must_scan then symbol := scan_next_symbol( );
  relation := precedence_relation(top(parse_stack), symbol);
  perform_operation_based_on_relation(relation, parse_stack,
                                      symbol, must_scan)
  end
end
```

Figure 1.
An example of a structured program module.

description of the compiler in the sense that if we had a machine with all the module names as primitives, it would run. However, such a machine is unlikely to exist, so the next step is to select a module name and code the module which explains it in terms of other module names. The process will continue until all module names are defined.

So far, the description and example could fit either M2 or M3. The difference between them may be illustrated by considering the module name "push" and the data type "stack". The two are obviously related: stack is an abstract data object which is to be operated on by the abstract operator push. The M2 approach is concerned only with control; it ignores the problem of how to handle stack and instead permits ad hoc solutions (for example, stack could be defined on the spot as an array). The M3 approach requires the definition of stack to be delayed just like the definition of push is being delayed -- an idea not supported by existing programming languages. A further problem is that access to stacks should be limited to just those modules which should access them (push, top and finished in the example)

SPIL is defined using the M2 definition of structured programming, a logical decision in view of the requirement that SPIL was only a tool, not a goal: we could not justify the effort of developing new programming language concepts. Also, the success of the IBM experiement, which uses the M2 meaning of structured programming, encouraged us to believe that M2 provided a sufficient foundation for a system design language.

Levels of Abstraction

We expected SPIL programs to exhibit some M3 characteristics because we intended to group SPIL modules into levels of abstraction (15). A level of abstraction is a group of related modules which share common resources, and

access to the resources is denied to modules in all other levels of abstraction. Thus levels of abstraction provide the type of limited access desirable for M3 structured programming. Resources include real machine resources (e.g. I/O devices), common shared data, and information. For example, if there were a level of abstraction supporting stacks, modules "push" and "top" (among others) would belong to it, and its resources would include information (the format of stacks), and data (the pointer to the top of the stack and possibly the stack itself).

Levels of abstraction do not exist in SPIL; we merely intended that levels be used by the programmer writing in SPIL. However, one important SPIL design decision is due to this intention. When considering a structured program module, the question arises of whether the modules named by that module have access to the variables declared in that module; for example, in Figure 1, can module scan_next_symbol access variable symbol? We decided that they can not. In SPIL each module must specify precisely which data resources it uses; these should be the resources belonging to its level of abstraction and they constitute its entire non-local environment. In SPIL, there are no free variables and consequently no implicit data connections between modules.

How SPIL Supports Structured Programming

The most important way in which structured programming can be supported is to permit the independent processing (compilation) of modules, and to provide for the linking (binding) of module name with module.

SPIL modules. A SPIL module is usually a procedure -- the same Venus machine procedure which became the basic building block of SPIL. However, a module may also be a macro definition. Macros are used in two different ways. They are used to contain information about data structures, thus pro-

viding standardization on the description of shared data, and also permitting the actual definition of a data structure to be delayed. Macros may also be used to permit a name to stand for a body of code without making the code into a procedure. Transfer of control between procedures is so fast on the Venus machine that very small procedures are feasible, and a macro would rarely be chosen over a procedure because of size of code. However a macro is useful to attach a name to a body of code which is used in only one place and runs in the environment of that place. Permitting macros to be used in this way enhances the readability of programs, an important benefit of structured programming.

SPIL Module Names. Any module accepted by the SPIL compiler has a module name. The module names constitute a global name space, which leads to the possibility of name conflicts. SPIL eases the conflict problem by having module names be two-level names. The intention is that the first part of the name be the name of the level of abstraction to which the module belongs, and the second part of the name identify the module within the level.

The SPIL compiler handles the referencing of a module by its module name differently for procedures and macros. For procedure modules, the SPIL compiler stores the module under its name; then a reference to the module name causes the compiler to build a link through the name to the appropriate module. If the module name is used before the module exists, the compiler prepares a spot for the module name and builds the link, and when it finally compiles the module, stores it (under its name) in the spot already prepared.

When the SPIL compiler processes a macro module, the module is also stored under its name; however a reference to the module name causes the module to be retrieved (through its name) and inserted bodily into the referencing module. If the module does not exist yet, the compiler ignores

it completely, implying that a macro's body must constitute a syntactic category for which "empty" is a reasonable value. The two interesting uses of macros, to stand for a data structure definition or a statement of a program, both satisfy this requirement.

Incremental Compilation. The SPIL compiler provides the incremental compilation necessary to building systems, which tend to be so large that it is impractical to wait for the whole system to exist before compiling. When a system is built bottom-up, incremental compiling is no problem, since names are defined before they are used (except for recursion). In top-down programming, just the opposite is true, but the need for incremental compilation still remains, as does the need for check-out of part of the system before the whole system exists. During check-out, the undefined modules may be simulated, if necessary, either by providing simulated definitions of them, or, in an interactive system, by allowing the programmer to simulate them on-line.

MAIN FEATURES OF SPIL

This section contains brief descriptions of the most important features of SPIL. It does not describe SPIL in any detail; a user interested in the details should consult (16).

Procedures

SPIL is a procedure-oriented rather than a block-structured language. SPIL procedures constitute the structured program modules, and may be compiled separately. Each procedure has a completely private local environment and a non-local environment consisting of the data segments it shares with with other procedures in its level of abstraction. There is no concept of a free variable in SPIL.

A procedure consists of a number of declarations followed by the procedure body. The procedure body is a goto-free program built of the following control structures: concatenation (like the compound statement in Algol 60), iteration (the WHILE statement), selection of a statement based on the testing of a condition (the IF statement), and selection of a statement based on the value of an expression (the CASE statement). The statements which the control structures link together include assignment statements, calls of other procedures (using their module names), < machine ops > , and the return statement.

Data Segments

The most important type of data available to a SPIL procedure is stored in a data segment and shared with other procedures in the same level of abstraction. The level of abstraction really owns the data segment, and it is not usually convenient to consider the segment as belonging to a particular procedure. SPIL views data segments as having an existence of their own,

independent of any procedure. This implies that data segments must have their own names, in just the same way as procedures (or modules) have their own names. SPIL data segments have two-level names just like procedures, and the SPIL compiler treats data segment and procedure names the same, although it will not compile a call to a data segment nor a data access to a procedure. The first part of a data segment's name is intended to be the name of the level of abstraction which owns the segment as a resource.

In order to access data segments, a procedure must include a declaration stating the names of the segments (it must also contain a similar declaration stating the name of each procedure it calls). The procedure must also contain a structure declaration (similar to a PL/1 structure) for each segment describing how data is stored in (allocated within) the segment. Examples of data segments for a compiler are the segments containing the symbol table, the parse stack, and the tables which describe the grammar of the language.

The analogy between SPIL procedures and data segments even extends to allowing the compilation of data segments. Compiling a data segment is one way of initializing the level of abstraction which owns the segment. An example of a data segment initialized in this way is the segment containing the grammar of a language; whenever the grammar is changed, the segment is merely recompiled.

Local Environment of Procedures

The local environment of a procedure consists of the arguments passed to the procedure (all by value) and simple (unstructured) local variables. The types of these variables are just sufficient to permit computation of arithmetic or logical expressions and to permit data in segments and, in rare cases, in core, to be accessed. Computation variables, declared as TEMPS, provide space for

16 bits of information. POINTER variables are used to access data in segments, but must first be associated with a segment via the ASSOC declaration. This association expresses the fact that a segment address has two parts: the name of the segment and the address within the segment.

A pointer may be used to access data in a segment either by byte (8 bits) or by halfword (16 bits); the type of access is indicated by a prefix of % for bytes and @ for halfwords. Use of a pointer without a prefix means the value of the pointer itself, rather than the value which the pointer points to. Information in the structure declaration of the segment ASSOCiated with the pointer is used to initialize the pointer and as an offset to the current value of the pointer.

A procedure may occasionally need to access core; for example, low-level I/O control procedures in the Venus Operating System (2) must fill and empty core buffers. Core access is achieved by prefixing a TEMP with % or @. The SPIL compiler will compile core access, but it always warns the programmer when it does so.

EVALUATION OF SPIL

The primary issue in evaluating SPIL is its success as a system design language. However, a few remarks are in order about how complete SPIL is as a tool for constructing systems.

It may seem to the machine language programmer that SPIL is not an adequate language for doing system programming because it appears to pay so little attention to basic system programming issues like I/O and interrupts. Take, for example, the I/O control programs which are part of any operating system: can they be programmed in SPIL? Although we have not had the experience of writing such programs in SPIL, we do know that the I/O control programs of the Venus Operating System (2) could easily be rewritten in SPIL (in fact, they could be directly translated into SPIL). They require, in addition to core access, the P, V and SIO (start I/O) instructions, all of which are available through the < machine op > part of SPIL.

In fact, every piece of Venus software could be rewritten in SPIL except for the program providing paging of segments. This program, which is really an extension of the Venus microprogram, must be core-resident, and may only use a subset of the Venus instructions; for example, it may not use segments. The program currently consists of approximately 800 instructions; whether it should remain in machine language or be bootstrapped into SPIL is an open question.

SPIL as a Design Language

The following properties, all of which are desirable in a design language, provide a framework within which SPIL can be evaluated.

1) A design language should express the design as it develops in a natural and economic way. It can do this by allowing the designer to build an outline of the system, with unimportant details suppressed. It also should permit the outline to grow as the designer sees fit. When designing a system, more attention is paid to some areas than others because of efficiency considerations, or just from concern over whether an idea is practical. Thus the design grows in a lopsided way; the design language should permit this and insure that no effort is wasted and no unnecessary effort is required.

2) A system design language should provide a vocabulary of abstractions which help the designer to think about his design.

3) A single language should be used for both design and implementation. This property is desirable for three reasons: First, and most important, there is really no clear distinction between design and implementation, although traditional software project organization implies that there is. Design decisions are made at every stage of program construction although the effects of these decisions become less global as the system nears completion. Thus having a separate language for design introduces an artificial distinction between design and implementation; a good design language should express the neverending nature of design. Second, the structure of the system should continue to be apparent even after implementation is complete, implying that the system will be easier to understand, modify, and maintain, and that less documentation may be required. Third, errors may be introduced in translating from the design to the implementation if two languages are involved; also extra effort is required in the translation.

4) A design language should encourage the development of "good"

system structure. The meaning of "good" must come from a coherent design methodology. (For work in this area see (6, 17).)

SPIL satisfies the first three objectives, as should any top-down, structured programming language. Its power comes from the ability to have a module name stand for a module. The module names constitute the vocabulary of abstractions. A description of the system in which some module names are undefined is an outline of the system, in which unimportant details (the missing module definitions) are suppressed. SPIL will compile such a partial system description, which can even be executed and debugged. SPIL insures that the structure of the design remain apparent by keeping all the modules separated; the definition of a module never replaces the use of its module name in a program listing, even if the module is a macro.

To determine how well SPIL satisfies the fourth objective, we must rely on our experience in using it. A compiler for SPIL has been written in SPIL and is now running on the Venus machine. SPIL has proved convenient to use and the results obtained were satisfactory in terms of programmer productivity and reliability of software. However, in the design of the compiler, levels of abstraction are well-defined only where obvious (for example, the scanner is a separate level); in other places the distinctions between the levels are blurred and the resources not clearly segregated. SPIL therefore does not entirely satisfy requirement four, because it is clear that we had in mind a design methodology, levels of abstraction, providing a definition of "good" system structure which we hoped that SPIL programs would have. The problem is that SPIL does not directly support the methodology: it is possible to build a system composed of levels of abstraction using SPIL, but it is not necessary to

do so. This implies that the burden of designing a good system structure falls on the programmer; SPIL does not help the programmer discover the structure as much as it should.

Our conclusion is that structured programming, interpreted to mean top-down control and ad hoc data, is not a sufficient basis for a system design language. Instead, the issues involved in data must be faced, and new programming language concepts developed by appealing to a design methodology. The resulting language will be a complete structured programming language. It should have all the advantages that SPIL possesses as a design language, and also contribute to the development of a good system structure.

ACKNOWLEDGEMENTS

The author gratefully acknowledges the efforts of Chip Gulden and Nancy Ivan in building the SPIL compiler, and the helpful suggestions made by Jack B. Dennis, Steve Zilles and Leroy Smith about the content of this paper.

REFERENCES

1. B. J. Huberman, "Principles of Operation of the Venus Microprogram", The MITRE Corporation, MTR-1843, Bedford, Massachusetts, 1 May 1970.

2. B. H. Liskov, "The Design of the Venus Operating System", Communications of the ACM, 15, 3, (March 1972), 144-149.

3. W. A. Wulf, D. B. Russell, and A. N. Habermann, "BLISS: A Language for Systems Programming", Communications of the ACM, 14, 12 (December 1971), 780-790.

4. B. H. Liskov and E. Towster, "The Proof of Correctness Approach to Reliable Systems", The MITRE Corporation, MTR-2073, Bedford, Massachusetts, 9 March 1971.

5. T. Baker, "Chief Programmer Team Management of Production Programming", IBM Systems Journal, 11, 1 (1972).

6. B. H. Liskov, "A Design Methodology for Reliable Software Systems", AFIPS 1972 FJCC, Vol. 41, Pt. 1, Spartan Books, New York, 191-199.

7. A. N. Habermann, "Prevention of System Deadlocks", Communications of the ACM, 12, 7, (July 1969), 373-377, 385.

8. N. Wirth, "PL360, A Programming Language for the 360 Computers", Journal of the ACM, 15, 1 (January 1968).

9. E. W. Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM, 11, 3, (March 1968), 147-148.

10. H. D. Mills, "Structured Programming in Large Systems", Debugging Techniques in Large Systems, R. Rustin (ed.), Prentice Hall, Inc., Englewood Cliffs, New Jersey, 41-55.

11. E. W. Dijkstra, Notes on Structured Programming, Technische Hogeschool, Eindhoven, Netherlands, August 1969..

12. N. Wirth, "Program Development by Stepwise Refinement", Communications of the ACM, 14, 4, (April 1971), 221-227.

13. E. W. Dijkstra, "The Humble Programmer", Communications of the ACM, 15, 10, (October 1972), 859-866.

14. W. Wulf and M. Shaw, "Global Variable Considered Harmful", SIGPLAN Notices, 8, 2, (February 1973), 28-34.

15. E. W. Dijkstra, "The Structure of the 'THE'-Multiprogramming System", Communications of the ACM, 11, 5, (May 1968), 341-346.

16. B. H. Liskov and L. A. Smith, "SPIL - A Systems Programming Implementation Language", The MITRE Corporation, MTR-2344, Bedford, Mass., 15 June 1972.

17. D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, 15, 12, (December 1972), 1053-1058.