SP/k: A System for Teaching Computer Programming

R.C. Holt, D.B. Wortman, D.T. Barnard, and J.R. Cordy
University of Toronto

SP/k is a compatible subset of the PL/I language that has been designed for teaching programming. The features of the SP/k language were chosen to encourage structured problem solving by computers, to make the language easy to learn and use, to eliminate confusing and redundant constructs, and to make the language easy to compile. The resulting language is suitable for introducing programming concepts used in various applications, including business data processing, scientific calculations and nonnumeric computation. SP/k is actually a sequence of language subsets called SP/1, SP/2, ... SP/8. Each subset introduces new programming language constructs while retaining all the constructs of preceding subsets. Each subset is precisely defined and can be learned or implemented without the following subsets.

Key Words and Phrases: programmer education, universities, community colleges, high schools, PL/I, SP/k, minicomputers, programming language design, teaching programming, introductory computing

CR Categories: 1.5, 4.2, 4.12, 4.13

Authors' address: Computer Systems Research Group, University of Toronto, Toronto, Canada M5S 1A4.

1. Choosing a Programming Language

Each year many thousands of students take courses in introductory programming using a variety of languages, equipment and teaching materials. This paper presents a carefully worked-out system intended for use in these courses. The system consists of a language called SP/k, SP/k processors for both minicomputers and large computers, and textbooks.

The first major decision in the development of the system was the selection of a suitable introductory programming language. We felt the language should meet these criteria: (1) it should be (part of) a "real" programming language used in business, science, and government and should be appropriate for introducing programming concepts used in those areas, (2) it should encourage systematic problem solving and structured programming, (3) it should be a small, convenient, easy to master language, and (4) it should be easy to support by economical, diagnostic language processors on a variety of machines including minicomputers.

Given the constraint of using a "real" language, Fortran must be considered carefully, because it is one of the original high-level languages and is the *de facto* standard language in many programming applications. Certainly, it is possible to teach programming using as crude a tool as standard Fortran, but it requires extra time and effort in a programming course. Clumsy language features too often distract students from learning orderly methods of program construction. In a course directed at teaching programming concepts, and not just language details, it seems appropriate to abandon standard Fortran in favor of a language that provides structured and convenient facilities such as **while** loops, simple input-output, character string manipulation, records, and nested procedure definitions. After programming and its associated structuring methods have

been mastered, it is a simple matter to learn the limitations and idiosyncracies of standard Fortran.

Cobol is also a widely used language that must eventually be learned by many programmers. However, its pedagogic and linguistic limitations make it a poor introductory language.

The fact that Basic is implemented on many minicomputers and has an extremely simple syntax makes it attractive. A student can learn Basic more quickly than a language like Algol 60. But he is then hampered by inept language constructs, such as non-mnemonic variable names, when he tries to solve nontrivial programming problems. Basic seems to be even less appropriate than standard Fortran for teaching programming concepts.

Algol 60, Algol-W [1] or Pascal [17] may be chosen because of their cleanliness and elegance. But for many educators, students, and employers the advantages of a more widely accepted language outweigh any advantages of additional elegance. And the temptation to invent YATL (Yet Another Teaching Language) which is supposedly "ideal" for teaching programming concepts, should certainly be resisted.

The PL/I language has the advantages that it includes reasonably good programming constructs and is widely used—though not as widely used as Fortran or Cobol. Because of these advantages, we chose to make our introductory language (SP/k) a subset of PL/I. The disadvantages of PL/I are that the full language includes peculiar and anomalous features and is expensive to process [10]. These disadvantages can be overcome if PL/I is appropriately restricted, leaving a subset language whose size and elegance is comparable to Algol 60. (Also see Gries' argument [9] for teaching a subset of PL/I.)

2. Design of the SP/k Language

PL/I satisfies our criterion that the introductory language should be a "real" language, and PL/I includes modern constructs, such as **while** loops, which help in writing well-structured programs. However, it must be extensively pruned to produce a subset that satisfies our other criteria that the language be easy to learn and use and easy to process.

The subset language should be "complete," meaning that it should be a language in its own right (independent of omitted PL/I constructs) and it should be convenient for programming at the introductory level. It should be a small, clean language, so a student spends less time memorizing and choosing among language constructs and more time concentrating on programming, and so he can master the language, gain confidence in using it as a problem-solving tool, and progress to more important matters such as algorithm design.

SP/k was designed—and PL/I pruned—by omitting and restricting language constructs to minimize:

(1) *Redundant constructs.* Where PL/I provides several equivalent mechanisms, all but one were eliminated. For example, the PL/I programmer can use either binary or decimal numbers; this choice is eliminated in SP/k, thereby removing some technical trivia from the classroom. Generally, constructs were omitted unless there was a pressing reason for keeping them, so the student would have less to memorize and the compiler could be smaller.

(2) *Unpleasant surprises.* Perhaps the best known unpleasant surprise in PL/I is the expression 25+1/3 which evaluates to neither 25 nor 25.333 . . . but to 5.333 . . . . Other notable examples involve anomalies in implicit conversions among decimal numbers, binary numbers, logical values, and character strings. These surprises are weeded out by SP/k constraints upon the offending constructs. These constraints allow an SP/k processor to do a better job of automatically diagnosing a class of subtle programming errors.

(3) *Easily misused constructs.* Pointers and **go to** statements are not appropriately constrained in full PL/I and are eliminated in SP/k.

(4) *Constructs suited for more advanced programming.* PL/I has constructs that are useful in particular programming situations, but are not needed in an introductory course. Perhaps the most notable examples are compile-time processing and concurrent programming (multitasking); these are omitted from SP/k.

(5) *Difficult to compile constructs.* One unpleasant aspect of full PL/I is its convoluted set of arithmetic precision rules. Mechanisms like these greatly complicate a compiler, and they also confuse the programmer. Various restrictions in SP/k, especially the omission of scaled arithmetic, remove this confusion and simplify the compiler. Generally, considerations (1) to (4) have led to a clean, easy to compile language subset; few of the language restrictions of SP/k were dictated by implementation concerns, although we kept these concerns in mind throughout the design of SP/k.

As (1) to (5) show, in many ways a "less is better" attitude was taken in designing SP/k. Very briefly, we will now give the restrictions on and omissions from full PL/I. In SP/k every variable must be declared. These declarations are not allowed to specify number bases (binary versus decimal) or precisions. Implicit conversions are not allowed among numeric, logical, and character types. Constructs implied by the following keywords are omitted: **binary**, **complex**, **initial**, **external**, **pointer**, **goto**, **on**, and **begin**. The following constructs are eliminated: fixed-length character strings, label variables, operations on entire arrays or structures, pseudo-variables, data-directed input-output, static allocation, controlled and based allocation, multitasking, and compile-time processing. These omissions and restrictions characterize SP/k as a pruned version of PL/I; in the next section, we describe SP/k as a

language in its own right by listing its included constructs.

Although many constraints are placed upon PL/I, the SP/k language remains a compatible subset of PL/I, meaning that correct SP/k programs generally run with the same results under a variety of manufacturer-supplied PL/I processors. SP/k is also a compatible subset of the PL/I language as it is presently defined by the international PL/I standards committee [7].

Students are motivated by the fact that their programs can be run at many different computer installations. The student who later needs to program occasionally has learned a language he can use in a variety of environments. If he continues on to professional programming, he has nothing to unlearn about PL/I. Hopefully the disciplined use of language constructs enforced by SP/k will give him the discretion he needs when using the constructs provided by full PL/I.

Although the SP/k language is quite small, it contains more than enough language features for introductory courses, and more extensive and elegant features than commonly used introductory languages such as standard Fortran and Basic. For example, SP/k provides format-free input-output as well as flexible formatting, character string manipulation, **if-then-else** statements, **while** loops and recursive procedures. Students and teachers are relieved of a number of bothersome Fortran-like restrictions; for example in SP/k output items and **do** loop parameters can be expressions, and names of variables can be meaningful because they can be up to 31 characters long.

More advanced courses in programming may find that the restrictions of SP/k are too severe. Additional language constructs, such as explicitly controlled memory allocation, may be desired. Since SP/k is a compatible subset of PL/I, the appropriate PL/I feature can be learned and SP/k as extended by the feature can be processed by a standard compiler for full PL/I.

3. A Sequence of Subsets

SP/k is actually a sequence of subsets that are called SP/1, SP/2, . . . SP/8. The SP/1 subset allows for evaluation of expressions, like a calculator, and printing of character literals. SP/2 introduces variables and SP/3 provides loops and **if-then-else** statements. Each subset introduces new language features while retaining features of previous subsets.

This sequence solves one of the perennial problems of introductory programming. As J.J. Horning once put it, the subject requires that everything must be taught first! SP/k provides a convenient ordering such that the constructs of each subset can be completely understood and used in programs without knowledge of the remaining subsets. Here is a summary of the language features of the subsets.

SP/1: expressions and output. The first subset introduces arithmetic: plus, minus, multiply, divide, sin, square root, etc. The only statement is **put list**, which is used to print either values of numeric expressions or character string literals. This subset gives the student immediate access to the computer. This access is fun, but more importantly, it provides concrete examples of concepts such as "program preparation," "source program," and "program output." The rules of arithmetic for integers and floating point numbers can be covered completely in SP/1 before variables are introduced.

SP/2: variables, assignment, and input. This subset introduces integer and floating point variables along with the assignment and input statements. Using this subset, students can write programs to evaluate formulas, convert measurements in one type of unit to another and do simple tax calculations. Hopefully the fundamental concept of a variable is mastered before relatively sophisticated uses such as loop control are tackled in the next subset.

SP/3: selection and repetition. The third subset introduces control constructs, specifically selection via **if-then-else**, and looping via indexed **do** loops and **while** loops. Comparisons, $<$, $>$, $=$, . . . , and logical operators, **and**, **or**, and **not**, are introduced because they are needed in the conditional tests of **while** loops and **if-then-else**.

This subset contains one of the most difficult programming concepts for students, namely the construction of loops with associated controlling variables. As well, the subset provides the first opportunity to exploit the computer's potential for rapidly carrying out a sequence of repetitive steps. Many interesting problems can be programmed, including the summing of mathematical series, finding the minimum of functions, searching and validating data, and calculating averages and standard deviations. Some short courses in a high school might go no further than SP/3.

SP/k's small set of strictly nesting control constructs immediately suggests the building block concept of program structuring to the student; this concept is too often left unlearned with other languages.

SP/4: character strings. Manipulation of character strings is fun for students because it provides a graphic display of results. Intuitively clear processes such as text editing provide well-motivated examples of indexing, nested loops and selection. The varying length character string variables of SP/k have a powerful and convenient set of operations: specifically, concatenation, substring, and length functions.

SP/5: arrays. This subset introduces arrays of the four basic types: integer, floating point, logical, and character string. It allows the programming of nearly all the traditional examples of algorithms, such as the sieve method of finding primes, searching, sorting, merging, solving sets of linear equations, statistical analysis, graph plotting via the printer, text processing, the shunting yard method of compiling arithmetic expres-

sions, payroll, accounts receivable, and inventory.

SP/6: procedures. The sixth subset introduces subroutines and functions together with parameters. Higher-level structuring concepts such as modularity are implemented by these constructs. The idea of top-down design is reinforced by the language because SP/k procedures are nested and can have local variables. Recursion is supported and recursive problem solutions can be developed. See Figure 1 for the statement syntax of SP/6.

SP/7: formatted input-output. The easily learned format-free input-output statements introduced in the first two subsets are sufficient for many programming problems. The character manipulation facilities of SP/4 help solve the problem of detailed control of output. For some problems, however, a more flexible method of input-output is warranted. SP/7 provides format control with specifiable field widths and precisions, as well as floating currency signs and insertion of commas and decimal points.

SP/8: records and files. The eighth subset provides PL/I structures, which are used for data records. These are used both as the constituent parts of external files and as nodes within internal data structures such as trees. Sequential files are introduced; these can be read and written one record at a time.

The SP/k subsets are unique in that they provide a convenient partitioning of basic programming language constructs; this partitioning could also be applied to languages such as Algol 60 and Pascal. It serves as a useful framework for analyzing and defining a language and is the basis of the SP/k language specifications [11]. These provide a precise definition of each of the SP/k subsets in only 22 typewritten pages (not counting appendices).

The syntax of SP/k is presented in a concise format that is especially designed to be readable. For example, the statement syntax of SP/6 (Figure 1) requires only 39 lines and has been printed on one side of a punched card as a handy reference for students.

Since each subset is exactly defined and is independent of succeeding subsets, a compiler can be constructed to support SP/k at any particular level. In fact, our first SP/k compiler supported only the first six subsets.

At the University of Toronto we have been using the SP/k subset by subset approach for introducing programming. It has been quite successful and has benefited from the expected advantages of an orderly sequence and precisely defined programming language mechanisms at each stage. Since the language, its syntax, and its semantics are quite clean, students are able to rely on the definition of the language, rather than on the observed properties of a particular compiler.

The textbook by Hume and Holt [16] presents a structured approach to introductory programming that proceeds from subset to subset. However, the SP/k language can also be used in an introductory programming course without explicitly following the subsets; this is done in the textbook by Conway, Gries, and Wortman [4].

Fig. 1. Statement syntax of SP/6.

```
A program is:    identifier: PROCEDURE OPTIONS(MAIN);
                   {declaration}
                   {definition}
                   {statement}
                   END;
A declaration is: DECLARE (variable {, variable}) attribute
                   {,(variable {, variable}) attribute};
An attribute is one of the following:
                 FIXED
                 FLOAT
                 CHARACTER(maximum length) VARYING
                 BIT
A definition is: identifier: PROCEDURE
                   [(identifier {, identifier})]
                   [RETURNS (attribute)];
                   {declaration}
                   {definition}
                   {statement}
                   END;
A statement is one of the following:
                 PUT [SKIP] LIST(expression {, expression});
                 PUT PAGE LIST(expression {, expression});
                 GET [SKIP] LIST(variable {, variable});
                 variable = expression;
                 IF condition THEN
                   statement
                 [ELSE
                   statement]
                 DO WHILE (condition);
                   {statement}
                   END;
                 DO identifier = expression TO expression
                   [BY expression];
                   {statement}
                   END;
                 DO;
                   {statement}
                   END;
                 CALL procedure name
                   [(expression {, expression})];
                 RETURN [(expression)];

Notation: [item] means the item is optional.
          {item} means the item is repeated zero or more times.
```

4. A Student-Oriented System

The best possible introductory programming language would not be of much use without a good language processor. The processor should be convenient and inexpensive to use and sympathetic in its handling of programming errors.

Since SP/k is a compatible subset of PL/I, a programming course that is based on SP/k can be taught using a manufacturer-supplied PL/I compiler. However, in many instances, due to high cost or poor error

handling, these compilers are not suitable for use by students. The PL/C compiler implemented at Cornell University [5] provides efficient handling of student PL/I jobs on IBM 360/370 computers and has been used successfully in university programming courses based on the SP/k language. In such a course the students can think of SP/k as a set of programming conventions that rules out the use of some of the PL/C constructs.

We decided to develop a special SP/k language processor for several reasons. We wanted to be able to use SP/k on minicomputers. We wanted a processor that would automatically limit students to SP/k features to prevent them from using poorly structured or confusing features such as scaled arithmetic. We wanted to demonstrate that it is not hard to implement a highly diagnostic and efficient processor for SP/k. We were interested in software research and felt that new ideas in compiler design and programming methodology could be exploited to help us produce a high quality, student-oriented system [6, 12, 23].

We wanted the processor to take advantage of SP/k restrictions to improve error handling. For example, an SP/k processor can detect most misspellings of the names of variables because SP/k requires that all variables be declared. By contrast, in full PL/I each misspelling is implicitly declared to be a new variable. Another class of errors that can be automatically diagnosed by an SP/k processor, but not by a full PL/I processor, is associated with implicit conversion between numeric and non-numeric types. For example, in full PL/I the statement `I = J = 5;` compares J to 5 and assigns 1 or 0 to I depending on whether the comparison is true or false. Probably the programmer intended to assign 5 to both I and J. The error goes undiagnosed in full PL/I due to the implicit conversion of logical values to numeric values.

Error Handling

One goal for the SP/k processor was to maximize the useful information given to the student about errors in his program [13]. We expected this to result in more efficient use of computer resources and supplies, because fewer runs should be required to complete an assignment. More important, we expected this to minimize student frustration and to allow students to learn more about programming because less time would be spent "fighting the system."

In general, the compiler attempts to isolate and repair a particular error, then continues processing until another error is found, which is isolated and repaired, and so on. This approach can diagnose a number of errors in a program on a single run. The nature of each repair is reported to the student in terms of the source program. Short, to the point error messages are issued, rather than multiple line messages, because it typically happens that the processor is able to localize an error but is not able to relate it to the student's ideas about the nature of the error. Putting this another way, it is very difficult to have the processor automatically explain errors to a student. In general, the implemented compiler is able to repair any arbitrary student program into a legal SP/k program, though of course not necessarily into what the student intended.

We wanted to keep the signal-to-noise ratio high in error messages. Thus, at most one syntax error message is issued for a single line of source program; but a modified version of the source line shows all the repairs that were made to render it syntactically legal.

The compiler diagnoses and repairs semantic errors such as undeclared variables and illegal use of arithmetic operators. Repair of errors in expressions is done by supplying default values. For example, if the student attempts to add 5 to 'CAT' then a default result of 1 is supplied. Semantic errors for which there is no reasonable repair cause termination, with an appropriate message, when execution reaches the offending construct.

Once the compiler has translated a program and repaired any errors, execution takes place. For example, if the (redundant) keyword END required at the end of an SP/k program is omitted, it is inserted by the compiler and the repaired program is executed. To do otherwise would unnecessarily penalize the student for a simple clerical error and would in many cases increase the required runs for completion of an assignment. The high percentage of reasonable repairs made to errors justifies execution of the repaired programs. Figures 2 and 3 show a program in which several typical errors are repaired.

Runtime errors are also diagnosed and repaired; for example, uninitialized integer variables are given the value one. Out-of-bounds array subscripts are replaced by in-bounds values. If the program has too many runtime errors, it is terminated because this is an indication of serious logic errors that are quite possibly being repeated in a loop.

It is sometimes argued that processing after errors are detected encourages students to be sloppy in their preparation of programs. There may be some danger of extra sloppiness, but this can be overcome by emphasizing to the students the simple-minded nature of automatic repairs and by refusing to accept completed programs containing errors.

Several schools presently using the SP/k compiler provide their students with overnight turnaround. Students who are given such slow turnaround learn quickly not to rely on automatic repairs to correct their programs, and these same students are often able to accomplish much more on a single run because trivial errors do not cause processing to terminate.

The example program in Figure 2 illustrates the kind of assistance through automatic repairs that the SP/k processor provides to students. The program is obviously intended to print a list of names. The following typical programming errors are present in the program.

- –The keyword PROCEDURE is misspelled as PROCDEURE.
- –The final */ of the second line of comment is omitted.
- –The semicolon after VARYING is missing.
- –The variable NAME has no value when first tested against 'DUMMY'.
- –The required parentheses are missing from around the test NAME¬= 'DUMMY'.

The handling of these errors by the SP/k processor is shown in Figure 3. All of the syntax errors are of a clerical nature and the repairs reflect the programmer's apparent intent. The most serious error, forgetting to read in the first name before the loop begins, is detected at runtime, and results in setting NAME to '?' and this is printed by the PUT LIST statement. From then on the program is back on track, reading and printing names up to the dummy value. As the program's output shows, the input data was: 'FRANKIE', 'JOHNNY', and 'DUMMY'.

The compiler and run-time support detect all violations of the language specifications, and all error messages and repairs are related to the source program. The student can understand every action of his program in terms of the SP/k language, and he is insulated in general from the underlying hardware and operating system. This encourages him to think of programming in terms of a precise algorithmic notation (in this case the SP/k language) rather than in terms of a particular compiler with its supporting software and hardware.

Paragraphing

The SP/k processor always paragraphs students' programs. Each statement is placed on a separate line. Statements nested inside compound statements are indented. Nested procedure definitions are indented. Statements that do not fit on a single line have their continuations indented. (See Figure 4).

The main arguments for this automatic program formatting are as follows. The successive indentations provide a graphic display of the program's structure. Paragraphing by hand is very tedious for students and requires extra time at a keypunch. Consistency of paragraphing enhances readability and automatic paragraphing is certainly consistent. When a student always sees his programs paragraphed, it helps him learn the close relationship between program structure (emphasized by paragraphing), program construction and program execution.

The main argument against automatic paragraphing is that it may prevent students from learning to do paragraphing by themselves. This is not a very strong argument because it is easy to learn to paragraph – the automatic paragrapher teaches how. What is hard is the associated keypunching. As will be explained, the SP/k compiler accepts mark sense cards as well as punched cards. Hand paragraphing programs when using mark sense cards can be even more tedious than when using

Fig. 2. Example SP/k program containing errors.

```
LISTING:PROCDEURE OPTIONS(MAIN);
   /* READ AND PRINT NAMES UNTIL */
   /*   THE DUMMY NAME IS FOUND.
   DECLARE(NAME)CHARACTER(20)VARYING
   DO WHILE NAME¬ ='DUMMY';
      PUT LIST(NAME);
      GET LIST(NAME);
      END;
   END;
```

Fig. 3. Repair of errors by the SP/k processor.

```
   1 LISTING:PROCDEURE OPTIONS(MAIN);
                ?
****    SYNTAX ERROR IN PREVIOUS LINE.  LINE IS REPLACED BY:
   1 LISTING:PROCEDURE OPTIONS(MAIN);
        /* READ AND PRINT NAMES UNTIL */
        /*   THE DUMMY NAME IS FOUND. */
****    COMMENT IS ENDED WITH */
   2    DECLARE(NAME)CHARACTER(20)VARYING
                                       ?
****    SYNTAX ERROR IN PREVIOUS LINE.  LINE IS REPLACED BY:
   2    DECLARE(NAME)CHARACTER(20)VARYING;
   3    DO WHILE NAME¬='DUMMY';
                  ?
****    SYNTAX ERROR IN PREVIOUS LINE.  LINE IS REPLACED BY:
   3    DO WHILE(NAME¬='DUMMY');
   4       PUT LIST(NAME);
   5       GET LIST(NAME);
   6       END;
   7    END;
EXECUTION BEGINS.
****    ERROR IN LINE 3: CHARACTER VARIABLE HAS NO VALUE;  SET TO '?'
?                       FRANKIE                 JOHNNY
```

punched cards. (See [12] for a description of the automatic paragraphing techniques. See [18] for the description of a paragrapher for full PL/I.)

Errors in student programs should be handled sympathetically, but not at too great a sacrifice in processing time. Neither can the complexity of error handling routines be allowed to degrade system reliability. The error handling strategies in the SP/k processor have been designed with these constraints in mind and the implemented system processes jobs at a speed that is comparable to other widely used student language processors, with what appears to be a higher degree of reliability.

(The approach taken to error handling is in many respects similar to that used in the PL/C compiler and its predecessors [5]. The technique of automatic syntax error repair is discussed in [12] and automatic semantic error repair is discussed in [6].)

SP/k Processors

There are presently two SP/k processors, one called SP/k/360 that runs on IBM 360/370 computers and one called SP/k/11 that runs on Digital Equipment PDP-11 minicomputers. The SP/k/360 processor also runs on the UNIVAC 90/30 computer (which has a 360-compatible instruction set). These two processors are written in a high-level implementation language, the SUE System Language [2], and are the same, except for minor machine-dependent variations, at the source level. The SP/k/360 processor is produced by feeding its source through SUE/360, which is a SUE System Language compiler that produces 360/370 machine language.

Analogously, SP/k/11 is produced using the SUE/11 compiler.

The two SP/k processors are functionally identical. Teaching materials including texts can be used with both types of computer and students can run the same jobs, including control cards on both types of computer. In this paper we refer to "the SP/k processor" when the remarks apply to both SP/k/360 and SP/k/11.

The prototype version of the processor was implemented by the four authors during the summer of 1973; it was first used in a programming course in the fall of 1973 on an IBM 370. Since that time major parts of the system have been augmented or rewritten to expand the system, meet users' suggestions, and implement faster, smaller compilation strategies; considerable effort has gone into system validation and development of operating system interfaces.

There is currently no mechanism in the SP/k processor to allow specification of a particular level of SP/k for automatic restriction of student programs to that level; however, such a mechanism could be implemented without difficulty.

The processor was designed as a multiple pass system so that it can be overlaid when memory is scarce, as on a minicomputer, or loaded completely into memory on a large computer. The memory requirements are modest (see Appendix) and the overlaid version can run with a supporting operating system in 56K bytes of memory on a PDP-11 minicomputer.

The SP/k processor generates code for a pseudo-machine, rather than for the 360/370 or the PDP-11. The pseudo-machine is designed to allow easy code generation from SP/k programs, as well as to allow efficient execution by a software simulator. This simulator acts as the last pass of the SP/k processor.

The processor has a very small and simple interface to the operating system. Thus far SP/k/360 has been installed under three distinct IBM operating systems (DOS, MVT and VS-2). It has also been installed under OS/3 on the UNIVAC 90/30 computer, which is compatible with IBM 360 computers.

The SP/k/11 processor presently comes with its own stand-alone operating system [8], and has been installed under Digital Equipment's RT-11 operating system. SP/k/11 runs on a PDP-11 with 56K bytes of memory, a card reader, printer and teletype. These are off the shelf components, so the equipment is easy to specify, assemble and maintain. The SP/k software will work on most PDP-11 CPU models, and extra cost hardware features, such as hardware multiply and divide instructions or memory mapping, are not required. The equipment for the complete minicomputer system can be purchased for between $15,000 and $45,000, depending upon speed of devices, disk capacity, and choice of component suppliers. At the University of Toronto we have assembled a mobile SP/k/11 system that can be rolled into a classroom on a 1 m by 2 m cart; the system's components, including CPU, memory, reader, printer, disk and cart, cost about $26,000.

Fig. 4. An SP/k program as automatically paragraphed.

```
/* CLASSIFY PEOPLE AS TALL, SHORT OR TYPICAL */
HOWTALL:PROCEDURE OPTIONS(MAIN);
   DECLARE(PERSON)CHARACTER(20)VARYING;
   DECLARE(HEIGHT)FIXED;
   GET LIST(PERSON,HEIGHT);
   DO WHILE(PERSON¬='DUMMY');
      /* CLASSIFY THIS PERSON */
      IF HEIGHT>210 THEN
         PUT SKIP LIST(PERSON,'IS TALL');
      ELSE
         IF HEIGHT<120 THEN
            PUT SKIP LIST(PERSON,'IS SHORT');
         ELSE
            PUT SKIP LIST(PERSON,'IS TYPICAL');
      GET LIST(PERSON,HEIGHT);
      END;
   END;
```

Operational Aspects

The SP/k software was designed for convenient, low-cost operation. The minicomputer version of the system provides automatic job-to-job transition along with buffering of input cards and output lines, and an operator is not needed. Students can run their own jobs by simply stacking their cards in the card reader and pushing the reader's "start" button. They can retrieve their own output by removing it from the printer. The system can run stacks of jobs unattended and more jobs can be added to a stack of jobs currently in the reader's input hopper. The supplied operating system requires no attention from the students except for clearing card and paper jams and keeping the printer loaded with blank paper.

The SP/k/360 processor also provides automatic job-to-job transition, and a student-run operation is possible by use of a dedicated reader and printer. This is the mode of operation at the University of Toronto on an IBM 370 model 165-II. If this mode of operation is not feasible, student jobs can be collected and run as a batch; for example, some Toronto school boards run SP/k/360 on IBM 360 model 30's and provide overnight turnaround to high school students via courier service.

SP/k/360 and SP/k/11 monitor the amount of execution and lines of output. If either exceeds specified limits, the job is terminated and the next job is started. Control cards can be used by students to increase these limits, up to absolute maximum limits set by each installation.

An effort is made to conserve paper by minimizing the lines printed per student job. Source listings, error messages and job headings are compact (see Appendix). Jobs can be printed on various page widths, from 72 columns to 132 columns. The narrow width is possible because the automatic paragrapher already has a strategy for continuing long statements from printer line to printer line. The use of narrow paper can cut down paper costs and allows minimal printing devices, such as teletypes, to be used [22].

The software will accept SP/k programs on either

Table I. Comparison of Processors.

Test	1. Certification		2. Validation		3. Demonstration		4. 25K user program (200 symbols)
	229 jobs 7280 cards		88 jobs 3430 cards		50 jobs 2135 cards		1 job
Measure (units) / Processor	CPU (min)	lines	CPU (min)	lines	CPU (min)	lines	memory (bytes)
SP/k/360 Release 2.1	0.57	13096	0.24	5716	0.14	3346	76K
PL/C Release 7.5	0.81	16660	0.72*	6642	0.18	4416	112K
IBM Checkout Version 1 Release 2.2	1.71	16322	1.02	8215	0.61	5555	96K

* processor aborted; one job deleted and batch rerun.

mark sense cards or on punched cards. The advantage of mark sense cards is that they do not require keypunches, so a school is spared the expense of these machines. Besides, the students can prepare mark sense cards at home or in a regular study period. A number of Toronto high schools are using SP/k via mark sense cards and a courier service to a remote school board computer.

5. Experience with the System

The SP/k system is in use at a number of universities and high schools. It has been used on several models of 360 and 370 computers. A PDP-11 minicomputer was installed in the back of a high school classroom and the students used SP/k via mark sense cards. During the fall of 1975 approximately 1,000 University of Toronto students took introductory programming using SP/k and ran their jobs on the 370 model 165-II.

An Evaluation by Teachers and Students

During the spring of 1975 the SP/k language was used by 10 classes in five different high schools, one of which used the classroom minicomputer. Eight teachers and over 200 high school students were involved. After the classes were taught SP/k, the teachers and students filled in questionnaires evaluating the SP/k language, the use of mark sense cards and the value of fast turnaround of student jobs [15].

All the teachers and about 50 of the students knew Fortran before they learned SP/k. Due to the existing predominance of Fortran in these high schools, SP/k was evaluated by comparing it to Fortran. When asked for their preference between SP/k and Fortran in seven categories, the 50 students preferred SP/k over Fortran in a ratio 2 to 1, averaged over the seven categories (8% said "no preference" and 5% said "do not know"). The teachers preferred SP/k over Fortran in an average ratio of 5 to 1 (27% said "no preference" and 7% said "do not know"). See the report [15] for more detailed figures.

Well-designed mark sense cards were felt to be suitable for teaching programming, although the students found punched cards convenient for large programs, i.e. more than about 40 cards.

The PDP-11 minicomputer installed in the back of the classroom stimulated interest and provided a tangible measure of the computer's performance, capability and cost. The questionnaires showed that, to the students, the main advantage of the classroom computer was that it provided essentially instant turnaround of their SP/k programs and was available when needed. To these students, who were accustomed to either overnight turnaround or traveling to a computer center, the classroom computer made programming easier and more enjoyable. The minicomputer system had no operator and was run by the students. One teacher was in charge of supplies and responsible for the equipment. Students quickly learned on their own how to take care of operational details like clearing card and paper jams. In general the system was very reliable.

Performance of the System

The PDP-11 version of the system has been tested with a relatively slow CPU (model 10) and a high-speed printer (600 lines per minute). For a batch of small student jobs, the throughput of the system was limited primarily by the printer. Of course, it is possible to become CPU-bound on this minicomputer system by submitting programs with long-running computation loops, so this type of calculation should be avoided in programming assignments to be done by many students.

During the fall term of 1975 the statistics from the University of Toronto Computer Center showed that SP/k/360 is quite efficient at processing student jobs. For the entire fall term the average time to process an SP/k job, including both compilation and execution, was approximately half a second of CPU time on the IBM 370 model 165-II. This average covers the jobs of the approximately 1,000 students studying programming using SP/k. During this same period no bugs were detected in the SP/k/360 processor, making it one of

the most reliable pieces of software ever installed at the computer center.

Appendix. Processor Performance Comparisons

The only widely used PL/I processors that are available for testing at the University of Toronto Computer Centre and that are intended for somewhat similar purposes as SP/k/360 are PL/C [5] and the IBM PL/I Checkout compiler [20]. We ran four sets of tests (SP/k jobs) to compare the speeds and sizes of SP/k/360, PL/C and Checkout, with the results shown in Table I. The first two tests are batches of SP/k jobs that contain many deliberate program errors and entail very little execution. The third test consists of example SP/k programs, mostly from the textbook by Hume and Holt [16]; these programs contain no errors and require a modest amount of execution. The results show that SP/k/360 produces much less output than PL/C or Checkout. They indicate that SP/k/360 compiles three to four times faster than Checkout. SP/k/360 was found to be faster than PL/C by rates varying from 22% (for test 3) to a factor of three (for test 2).

The last test is an attempt to compare memory requirements of the three processors. In this test, SP/k/360 uses three overlays per job and PL/C used four overlays per job. Checkout uses a relatively elaborate overlay scheme; although we attempted to make a reasonable estimate of required space for efficient compilation, the figure of 96K remains open to question. (Checkout was run in 250K bytes for the first test and in 150K for tests two and three.) At the cost of more overlays, SP/k/360 can be run in about 50K bytes with 10K bytes for the user program; it runs unoverlaid in 124K bytes with 25K bytes for the user program.

It is difficult to provide meaningful performance comparisons among language processors. Not only are the tests subject to question, but the processors are constantly being improved. One of the principal emphases in the implementation of SP/k has been high reliability; since this seems to have been attained, future developments may be directed at further decreasing of time and space requirements. (See also [19] for an analysis of execution speed for six PL/I compilers including SP/k/360, PL/C, and Checkout.)

Acknowledgments. The SP/k language design benefited from valuable suggestions by T.E. Hull, J.J. Horning, F.C. Phillips, D.G. Corneil, K.C. Sevcik and many others. The compiler was implemented by the four authors, with assistance from B.L. Clark, F.J.B. Ham, M.S. Fox and W.K. Fysh. The special purpose PDP-11 operating system was implemented by I.E. Greenblatt, P. Khaiat, E. Wong and H.G. Arnaud. The Ontario Ministry of Education supported the development of the SP/k language specifications and the evaluation of the language, mark sense cards and use of minicomputers in high schools [14, 15]. The associated research on parsing methods, compiler organization and operating system structures was supported in part by the National Research Council and the Defence Research Board of Canada. Digital Equipment Canada generously provided a PDP-11 minicomputer system for evaluation as a classroom computer. T.E. Hull, J.N.P. Hume and R.W. Conway provided valuable suggestions for improving this paper.

Received March 1976, revised August 1976

References

1. Bauer, H., Baecker, S., and Graham, S. Algol-W implementation. Tech. Rep. CS98, Comptr. Sci. Dept., Stanford U., Stanford, Calif., May 1968.
2. Clark, B.L., and Horning, J.J. The system language for Project SUE. SIGPLAN Notices (ACM) *6*, 9 (Oct. 1971), 79–88.
3. Clark, B.L., and Ham, F.J.B. The Project SUE system language reference manual. Tech. Rep. CSRG-42, Comptr. Syst. Res. Group, U. of Toronto, Toronto, Ontario, Sept. 1974.
4. Conway, R., Gries, D., and Wortman, D.B. *An Introduction to Structured Programming using PL/1 and SP/k.* Winthrop Publishers, Cambridge, Mass., 1977.
5. Conway, R.W., and Wilcox, T.R. Design and implementation of a diagnostic compiler for PL/1. *Comm. ACM 6*, 3 (March 1973), 169–179.
6. Cordy, J.R. A diagrammatic approach to programming language semantics. Tech. Rep. CSRG-67, Comptr. Syst. Res. Group, U. of Toronto, Toronto, Ont., March 1976.
7. Standard Programming Language PL/I, BASIS/1-12. European Computer Manufacturers Assoc. and ANSI, New York, 1974.
8. Greenblatt, I.E., and Holt, R.C. The SUE/11 operating system. *INFOR Canadian Oper. Res Inform. Processing 14*, 3 (1976).
9. Gries, D. What should we teach in an introductory course? SIGCSE Bulletin (ACM) *6*, 1 (Feb. 1974), 81–89.
10. Holt, R.C. Teaching the fatal disease (or) introductory computer programming using PL/1. SIGPLAN Notices (ACM) *8*, 5 (May 1973), 8–23.
11. Holt, R.C., and Wortman, D.B. Structured subsets of the PL/1 Language. Tech. Rep. CSRG-55, Comptr. Syst. Res. Group, U. of Toronto, Toronto, Ont., May 1975.
12. Holt, R.C., and Barnard, D.T. Syntax-directed error repair and paragraphing. Comptr. Syst. Res. Group, U. of Toronto, Toronto, Ont., June 1976.
13. Horning, J.J. What the compiler should tell the user. In *Compiler Construction: An Advanced Course,* F.L. Bauer, and J. Eickel, Eds., Springer-Verlag, New York, 1974.
14. Hull, T.E., Holt, R.C., and Phillips, C. An investigation of support materials for teaching computer studies in Ontario high schools. Ontario Inst. for Studies in Education, Toronto, Ont., 1975.
15. Hull, T.E., and Holt, R.C. Classroom computers and programming languages. Ontario Inst. for Studies in Education, Toronto, Ont., Jan. 1976.
16. Hume, J.N.P., and Holt, R.C. *Structured Programming using PL/1 and SP/k,* Reston of Prentice-Hall, Sept. 1975.
17. Jensen, K., and Wirth, N. *Pascal User Manual and Report.* Lecture Notes in Computer Science, Springer-Verlag, N.Y., 1974.
18. Conrow, K. and Smith, R.G. NEATER2: A PL/1 source statement reformatter. *Comm. ACM, 13,* 11 (Nov. 1970), 669–675.
19. Wortman, D.B., Khaiat, P.J., and Lasker, D.M. Six PL/1 Compilers. *Software Practice and Experience* 6, 3 (1976), 411–422.
20. OS PL/1 Checkout and Optimizing Compilers: Language Reference Manual, IBM Form No. GC28-8201-3.
21. Boulton, P.I., and Jeanes, D.L. The structure and performance of PLUTO, a teaching oriented PL/1 compiler system. *INFOR Canadian J. Oper. Res. and Inform. Processing* 10, 2 (June 1972).
22. Dewar, D.K.B., Hochsprung, R.R., and Worley, W.S. The IITRAN programming language. *Comm. ACM 12,* 10 (Oct. 1969), 569–575.
23. Barnard, D.T. Automatic generation of syntax-repairing and paragraphing parsers. Tech. Rep. CSRG-52, Comptr. Syst. Res. Group, U. of Toronto, Toronto, Ont., April 1975.