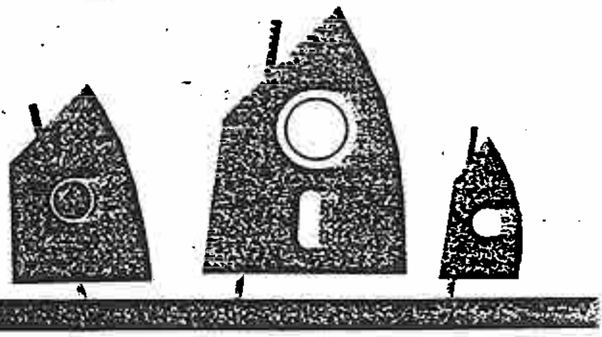
PSAIL: SAIL to C



By Peter F. Lemkin

Wouldn't it be nice to be able to use dynamic text strings in a Pascal-like environment as we currently are able to do in BASIC? Well, it appears we can— plus more.

The SAIL language has had dynamic strings for years. The original SAIL compiler, developed between 1969 and 1976 at the Stanford University Artificial Intelligence Laboratory, Stanford, Calif., runs only on DEC-10 and DEC-20 computers. Unfortunately, most of SAIL was written in 36-bit dependent assembly language and is thus nonportable.

A portable version of SAIL, called PSAIL, is being constructed to run in most medium size C environments. This article will present PSAIL in the context of the original SAIL language.

What is SAIL? It is a robust ALGOL-dialect programming language with many useful extensions.[1,2,3] Because of its expressivity, SAIL lends itself to writing large, complex application and system programs. Its robust approach to type coercion makes it closer to a D.W.I.M (Do What I Mean) algorithmic language than other popular block structured, strong type-checking languages such as Pascal, Modula-2 or Ada. On the other hand, C— which has looser type checking—does not have enough type coercion capability, especially in forcing arguments to the types expected in procedure calls. Without a lintlike syntax checker, writing large modular programs can be difficult.

Although not an ideal D.W.I.M. language by any means, SAIL comes closer to this approach than the other common block-structured languages mentioned. Of course, the price of more type coercion is that you can hang yourself more often.[4] Our experience, however, is that you learn to live with occasional pitfalls and take advantage of D.W.I.M. most of the time for greater overall productivity. Repeatedly SAIL has been selected in the National Institute of Health DECsystem-10 community as the implementation language of choice for large, complex systems.

Language overview

SAIL can be easily partitioned into several independent language subsets. This reduces what a new user has to learn to start programming quickly. These include: dynamic strings, dynamic arrays, dynamic records, macro expansion, conditional compilation, code inserted from other files, separate compilation of modules, bit manipulation, *if* and *case* statements and expressions, flexible I/O, powerful string scanning and conversion functions, LEAP (an associative data structure facility which includes dynamic data entities called items, datums, sets, lists, associations as well as associative search constructs—more on this later),[1,2,3] processes, contexts, interrupts, in-line assembly language, etc.

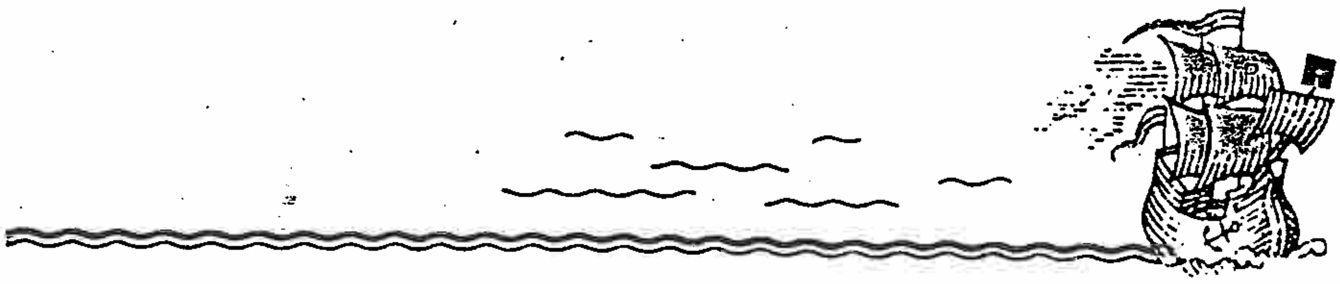
Note that complete exclusion of some of these sublanguages or features is easily achieved in actual programming practice. The most often used remainder is the core subset of SAIL, which captures its paradigm. This is relatively small—on the order of Pascal. For example, the PSAIL source code itself, written in SAIL, uses no records, LEAP, embedded assembly language, interrupts, processes, or context language constructs.

Table 1 illustrates some of the language elements with examples of SAIL fragments. Obviously, a much longer list would be required to illustrate the full language described in the *SAIL* reference manual.[1] In the examples, keywords appear in uppercase and user-defined symbols in lowercase. However, SAIL does not make alphabetic case distinctions as C does.

The PSAIL compiler project began as a result of the announced demise of the 36-bit/word DEC-10 and the increasing popularity of small UNIX systems. I had written a large data analysis system in SAIL called GELLAB,[5,6] which involved image processing and data base analysis for two-dimensional electrophoretic gels. To export this system, as well as other SAIL programs, the SAIL source code needed to be translated to some other language.

PSAIL was derived from an early program called SAITOC, written to translate GELLAB to C. Later, the goal of PSAIL was changed from that of a translator performing a partial translation to a full compiler capable of:

■ Handling any SAIL program and flagging illegal SAIL syntax, unimplemented SAIL language elements, and C portability problems



A few examples of SAIL features implemented in PSAIL

1. Dynamic strings, string concatenation, substrings, string garbage collection, and some explicit string conversion functions.

```
STRING s, s1, s2;  INTEGER i, j, k;
s1  := "Adam";  s2 := "Eve";
s   :=s1 & s2;
i   := LOP(s); COMMENT extract 1st char of s, shorten s;
s1  := s & s1[i for 10] & " AND " & s2[k for j];
r   :=CVD("3."&"14159");  s:= CVS(j+2**k);
s   := "The value of k is " & CVS(k) & " decimal and" &
     CVOS(k) & " octal";
```

2. Extensive macro facility including conditional compilation.

```
DEFINE # = "COMMENT ";  COMMENT # is short for COMMENT;
DEFINE dbug(x) = "PRINT(""x=""",x,CRLF)";
IFC (tops10=10 AND nFiles<15)
      THENC ... ELSEC ... ENDC
```

3. Dynamic n-dimensional arrays with ragged (i.e. > 0 or < 0 bounds).

```
REAL ARRAY abc[-10:10], xyz[p:q];
STRING ARRAY  xyzzy[500:1000,-7:25,-10:-5];
```

4. Optional dynamic array bounds checking which may be toggled on/off.

```
SAFE REAL ARRAY x[1:10]
NOW!UNSAFE x; ...  # No bounds checking on 'x';
NOW!SAFE x; ...    # Resume bounds checking on 'x';
```

5. Nestable variable declarations inside of LABELED blocks. Note redefinition of x and y and that QUOTED block labels must match.

```
BOOLEAN z;
DO BEGIN "block one"
   INTEGER x, y;
      BEGIN "block two"
      REAL x, y;
      ... Compute z ...
      END "block two";
   END "block one"
UNTIL z;
```

6. Type coercions automatically generated where they are least likely to cause trouble. (Some Pascal supporters might disagree with this!)

```
STRING s;  INTEGER i;
i := s;           s := i + "A";
IF s = "Q" THEN ...
```

7. Separate compilation with INTERNAL/EXTERNAL storage modifiers and REQUIRED source code file modules with optional VERSION checking.

Table 1. *(Continued on following page)*

- Compiling itself so that PSAIL would be portable and programs could continue to be developed in a SAIL environment
- Running on a wide variety of systems.

At the NIH, we have had FORTRAN, ALGOL-60, and SAIL available. Over the past decade, a large body of SAIL code (greater than 200,000 lines) was written for some very large systems (GELLAB, about 70K lines, MLAB,[7] over 25K lines, and MATEXT,[8] over 35K lines). Other SAIL programs include BRIGHT, PUB, and the original TEX. Like C, SAIL supports modularity but with most of the type-checking capability of Pascal and similar languages. SAIL has often been used to build quickly large, reliable systems.

Studies have shown that the majority of algorithmic code, especially code involving numerical routines[9] and salvaging user libraries, could benefit from a high-level language translation rather than rewriting the code from scratch in a target language. It is then possible to translate verbatim much of the existing SAIL code to C using PSAIL in those cases where 36-bit dependencies are not a problem.

Using a high-level language translator preprocessor with a target system compiler causes some additional overhead. The arguments made for the FORTRAN preprocessor RATFOR (added portability, added language functionality, and user productivity) also hold for PSAIL since a SAIL environment is superior to C for program development.

By modularizing large SAIL programs into small files that are separately compiled, the additional translation cycle time should not be excessive. W. Teitelman[10] suggests that short compilation times are conducive to rapid interactive development. PSAIL should be fast enough to minimize this time. It will continue to improve as processor speeds and memory size increase.

Selecting C

To achieve portability over a wide variety of target computer systems, a portable target language is required. By writing a machine-independent, run-time library in the same target language, using machine-conditional, high-level language code where required, the feasibility of portability is greatly enhanced.

Assembly language was ruled out because it is not portable. It was decided to use a high-level language as the target language; thus the compiler is more a translator or transliterator between high-level languages than a conventional compiler which generates assembly language or machine code.

A number of popular languages, including C, Pascal, Ada, Modula-2, and MESA, were evaluated for their ability to express high- and low-level structures available in SAIL. A. Feuer and N.G. Gehani[11] give a valuable comparison of Ada, C, and Pascal, exposing their strengths and weaknesses. A further constraint we imposed was that the language should be commonly available and consistently implemented. C was selected as coming closest to meeting most of these goals.

Others have also selected C as a target language for translators. Recently there have been a number of high-level language translators including: S-TRAN for BASIC-to-C, a Pascal-to-C by J. Peterson, and FORTRIX for FORTRAN-to-C. Another advantage of a C high-level language translator is that the target code is just C code. Generated code can be used with the PSAIL run-time library or merged with other C programs, which are becoming increasing available.

PSAIL is based on the published language standards for SAIL[1] and for C.[12] Brian Kernighan and Dennis Ritchie's *The C Programming Language* is developing into the ANSI X3J11 standard.[13] There are several advantages in trying to stay within the existing language standards. Both have been well documented by these existing reference manuals.

Adhering to the standards helps ensure portability of much existing SAIL code and target C code. However, a price for standardization is not being able to polish some of the rough edges of SAIL with more modern language forms. A PSAIL language extension facility lets us optionally do this to some degree, giving us the best of both worlds.

Sublanguages

In general, adding additional language facilities missing from Pascal and C to an abstract, block-structured language increases the complexity of the language. It is generally understood that one price of

```
INTERNAL INTEGER p, d, q;    # in defining module;
EXTERNAL INTEGER p, d, q;    # in other modules;
EXTERNAL STRING PROCEDURE match(REFERENCE STRING s);
REQUIRE "boobah.sai" SOURCEIFILE;
REQUIRE "48.55" VERSION;
```

8. Records, checked record classes and record garbage collection.

```
RECORDICLASS spot  (REAL angle, rad;
                    INTEGER area, xMom, yMom;
                    RECORDIPOINTER (spot) next;
                   );
RECORDIPOINTER (spot) rp1;
spot:xMom[rp1] := spot:rad[rp1]*COS(spot:angle[rp1]);
```

9. More natural Boolean relations as well as assignment embeddings.

```
STRING s; INTEGER ch;
s := "This string has 28 characters.";
WHILE s NEQ NULL DO
   If ("0" LEQ (ch: = LOP(s)) LEQ "9") OR
      (ch = ".") DO OUTSTR(ch);
   IF "Z" LEQ ch LEQ "A" THEN ch := ch-"A"+"a";
```

10. String scanning facility comparable to C's scanf() and scans().

11. Arrays which are dynamically defined and active in local blocks.

```
For i:= 1 STEP 10 UNTIL 101 DO
   BEGIN "Perform dynamic array allocation"
   REAL ARRAY vectorA[0:i], matrixB[0:2*i,0:i];
      ... process the arrays ...
   END "Perform dynamic array allocation";
```

12. Additional control statements: NEXT<label>, DONE<label>, CONTINUE<label> where <label> is optional.

```
DO BEGIN "outer loop of nonsense code"
   INTEGER x; REAL y, z;
   FOR x := 0 STEP 1 UNTIL 511 DO
     BEGIN "Loop x"
     FOR y := 25 STEP -(x+0.141) WHILE (x Leq 3001) DO
        BEGIN "Loop y"
        IF (z:=x+y)=123 THEN CONTINUE "Loop x";
        IF (z:=x-y)=1234 THEN DONE "Loop x";
        IF (z:=z+y)=321 THEN NEXT
                           ELSE NEXT "Loop x";
        END "Loop y";
     END "Loop x";
   END "outer loop of nonsense code"
UNTIL z<3.14;
```

Table 1. *(Continued from prrceding page)*

increasing the power of a language by extending its vocabulary and semantics is to increase the complexity perceived by the listener or writer in understanding and using the language. Computing history has shown that many programmers avoid the excessive complexity and size of languages such as PL/I and ALGOL-68. Even new languages such as Ada and MESA suffer from similar problems.

In addition, complex programming language solutions are more difficult to construct and port to other systems. I would suggest that it is this cost of portability that is partly responsible for the demise of these large languages' popularity.

So what can we do to get expressive power in a language at minimum cost? I suggest breaking the language into natural sublanguages so that programmers can use as much of the language as they want (or need)—and require the compiler enforce these language partitions.

We are experimenting in PSAIL with the hypothesis that a reduction in apparent language size is achieved by partitioning a large language into smaller disjoint sublanguages enforced by the compiler and held together by other connecting sublanguages. This should be reflected in reducing apparent complexity as observed by the user and would seem especially useful if some exotic aspects of the language are only needed in one or two modules.

Programmers normally think about their programming environment in terms of a language subset. A compiler should be able to enforce this thinking by having the programmer declare sublanguages either to exist or other sublanguages to be excluded. PSAIL currently contains the following four sublanguages and can handle more defined by the user:

- L0 = ALGOL-60 subset, strings, macros, etc.
- L1 = LEAP associative structure language
- L2 = Processes events and interrupts
- L3 = PSAIL extensions
- L4 = User definable

. . .

As a consequence of language partitioning:

- Less experienced users can work with a reduced language, which is easier to learn.
- Sophisticated users can expand the language to take advantage of advanced features.
- The PSAIL L3 extensions can be added. These currently include: generic procedures, array slices, the C ++, --, <opr> = operators, and embedded C code.
- Programmers may supply their own dynamic extensions to PSAIL (through *PSAIL!FORGET* and *PSAIL!DEFINE* compiler directives to modify the compiler by adding new keywords using existing parser capability as well as associating new run-time procedures). These extensions can become available to other users by specifying the new language definition in a REQUIRE file module. Some possible extensions might be to implement a concurrent LISP or add relational data base language extensions.

Portability

By assuming a consistent standard C environment such as ANSI X3J11, another hurdle to portability is overcome as target language code generators are easier to write. By divorcing the code generated by PSAIL from the much more machine- and operating system-dependent run-time library, we greatly facilitate the ease of porting SAIL source code between systems. The PSAIL run-time library consists of a number of *#include* type .h header files for C code files <sairun.c>, <pmath.c>, <gcrun.c>, <leaprun.c> and <procesrun.c>.

A potential disadvantage of this approach is that some run times could call <stdio>—effectively doing double interpretation. To avoid this, PSAIL run-time packages do their own file buffering and string handling rather than calling <stdio.c> repeatedly where major bottlenecks would appear.

These run-time libraries are also written in the same portable dialect of C that PSAIL generates. They use conditional C code, which is machine dependent only when required to handle machine-specific problems, whereas PSAIL generated code is completely machine independent. This method has been used for years in writing portable C code for different UNIX environments. One needs to compile the PSAIL run-time library only once after adjusting the <config.h> file to the specific system being ported.

PSAIL abstracts some of the code generation to a higher level than basic C code. For instance, whether the target system has a 32- or 16-bit integer is not important since PSAIL emits all integers as type *INTEGER*. This in turn will be defined differently in the only machine-specific file <config.h> for the two classes of machines, for example:

```
typedef int INTEGER; /* for 32-bit
                        word int */
```

or:

```
typedef long INTEGER; /* for 16-bit
                         word int */
```

Strings are another instance of this abstraction. PSAIL uses the C string pointer strategy to facilitate use of PSAIL strings with other C packages. The *STRING* declaration uses the C *typedef* of:

```
typedef char *STRING;
```

Using a compacting string garbage collector, PSAIL keeps track of all active string pointers using a run-time string pointer stack. Garbage collection is performed when the string allocator run-time *salloc()* runs out of space. For example, *salloc()* is called from the concatenation run-time procedure *catlist()* when it needs space the size of the strings to be concatenated. Strings may be as large as the memory available from the system. Another very useful SAIL feature is dynamically allocated arrays, which may have nonzero or negative lower bounds (sometimes called ragged arrays) and optional bounds checking.

The SAIL subset used in PSAIL was derived using several constraints. By restricting the language of PSAIL to a useful subset of full SAIL constrained by several factors, the resulting PSAIL language is a relatively robust portable language. These constraints include:

■ A few subsets of the full SAIL specification are omitted at this time from PSAIL as few SAIL users in our programming community use them—for example, contexts. Although LEAP is translated, run-time code will not be written for L1 (or L2) in the first release of PSAIL. The open library scheme permits PSAIL users to redefine or write these initially missing run times in C.

■ Features that are difficult to map to C or expensive to implement in terms of run-time efficiency are not currently implemented—for example, contexts. Similarly, PSAIL does not currently support nested procedures or global *GOTO*s, although code with warnings is produced.

■ Features that take advantage of the DECsystem-10 specific instruction set but are not efficiently simulated on different architectures (for example, variable byte size operators for 36-bit words) are not currently implemented.

■ There are also subtle data structure difference issues. In all of these cases, warning errors or comments are issued for the programmer. PSAIL's "warning" and "trash graphics" comments are used to catch these types of problems. For example, as the DECsystem-10 has a 36-bit word size, 36-bit arithmetic operations are also checked and when found, reported using "trash graphics."

PSAIL, a language translator as well as compiler, uses the philosophy of optional embedded C-style warning messages to warn about possibly illegal constructs. A summary appears at the end of compilation. Warnings are given both for SAIL source code syntax errors and target C code that would be nonportable due to limitations of C or in computation (for example, 36-bit operations). Programmers should use these warnings to edit their SAIL source code and try again. These forms include:

■ "Trash graphics" are used to warn of possible problems. There is a continuum of warning levels to let you pick the message level you are comfortable to work with (Figure 1).

■ Embedded warning C-style comments of the form:

```
/*WARNING*/ = check manually for
               possible semantic or
               syntax problem
/*UNDF*/ = Undefined symbol
/*N.P.*/ = Nonportable
/*N.I.*/ = Not implemented
/*LEAP*/ = LEAP sublanguage syntax
```

■ Fatal SAIL syntax error messages that indicate what is wrong with the code, suggest what should be fixed (if it can figure it out), and point to the actual illegal SAIL source code.

■ C portability tests are built into PSAIL. For example, you can optionally test for identifier name uniqueness to *n* characters. The proposed ANSI X3J11 standard has six-character external symbols, but other systems may have different lengths.

Although both SAIL and C are block-structured languages, simple one-to-one translation from SAIL to C is not always possible. For example, the following SAIL and C fragments are similar but not a simple 1:1 mapping:

```
SAIL: For i:=(a+b) Step -c Until d Do
      <SAIL statement>;
C: for (i=(a+b); i>=d; i-=c) <C
      statement>;
```

Therefore we perform simple translation where possible and elsewhere do recursive descent parsing and generate the nonlinear mappings where required. In some cases, where it is impractical to generate in-line C code, run-time procedure calls are generated instead.

By abstracting the target language, more attention can be paid to local optimization, taking advantage of the consistency of the target language and of C's power as a high-level assembler language. For example, the SAIL fragments:

```
INTEGER ARRAY a[1:10,1:20,1:30];
INTEGER b;
STRING s1;

a[i,j,k] := a[i,j,k] + b;
(s1[k For 1] = "Q")
```

are optimized to:

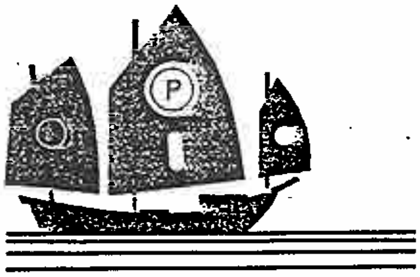
```
(a) a[i][j][k] += b;
     /* No dynamic arrays, C-style */
(b) *aAry(&a,i,j,k,3) += b;
     /* Dynamic arrays, no bounds
        check */
(c) *aChk(&a,i,j,k,3) += b;
     /* Dynamic arrays, bounds check*/
(d) (*(s1+k-1) == 'Q')
```

rather than the unoptimized C code, which includes redundant calculations and higher overhead with a run-time call:

```
/***************************/
/*<5>    /\         /\      */
/*         /------\         */
/*       --|      |--       */
/*      |  | o  o |  |      */
/*       -\------/-         */
/*        |  \/   |         */
/*        +       +         */
/* BEWARE!                  */
/***************************/

/* WARNING! SHIFT OF > 31-BITS IS NOT PORTABLE */
or
/* WARNING! LOGICAL OPERATION > 31-BITS IS NOT PORTABLE */
```

Figure 1.



Current status of SAIL features implemented in PSAIL

SAIL language feature	Implemented?
'DEFINE' macros	Yes, ~~with arrays and recursion~~ with multiplelines using '\'
Conditional macro expressions	Yes, = = > #if or eval in PSAIL
FORC, WHILEC, etc.	Not yet, extended SAIL macro facility
REQUIRE file statements	Yes, map = = > #include, run-time pkgs
ALGOL-60-like declarations	Yes
Declaration type checking	Yes
Automatic type coercion	Yes
Nested procedure declarations	Yes, but may be nonportable
Nested variable declarations	Yes
Dynamic arrays	Yes, arymk(), aryfree(), aChk()
Positive ragged arrays	Yes, both C and dynamic arrays
Negative ragged arrays	Yes, currently only for dynamic arrays
Safe array bounds checking	Yes, C arrays, aAdr() for dynamic
ALGOL-60-like control statements	Yes
String concatenation	Yes, a&b&c = = > catlist(a,b,c,0)
String operators	Yes, LOP,sub-strings, EQU(), etc.
Infix to prefix operators	Yes, MAX, MIN, ABS, etc.
PRINT and OUTSTR	Yes, map = = > pprintf(. . .) in <sairun.c>
TOPS10 I/O run time	Yes, most are handled in <sairun.c>.
TOPS20/TENEX I/O run time	Only those which are TOPS10 compatible
In-line assembly code	Yes, mapped to #asm/#endasm
RECORDS	Yes, mapped to struct's, no ralloc() yet
LEAP sublanguage	Most, maps through to <leaprun.c>
PROCESSES, EVENTS, INTERRUPTS	Most, maps through to <procesrun.c>
DEC-10 byte pointers in SAIL	Yes, Map = = > procedure calls, /*N.P.*/
CONTEXTS AI sublanguage	Not yet
<sairun.c>	Yes, written—not debugged
<gcrun.c>	Yes, written—not debugged
<pmath.c>	Yes, written—not debugged
<leaprun.c>	Not yet, not written, calls generated
<procesrun.c>	Not yet, not written, calls generated
STRING garbage collection	Yes, <gcrun.c> written—not debugged
RECORD garbage collection	Not yet, rec_gc() will mimic str_gc()
Procedure profiling	Yes, different from SAIL—not debugged
BAIL dynamic debugger	Not yet (maybe never . . .)
36-bit dependent code	Forget it! Recode your program!!!

Table 2.

```
(a) a[i][j][k] = a[i][j][k] + b;
(b) *aAdr(&a,i,j,k,3) =
        *aAdr(&a,i,j,k,3) + b;
(c) *aAdr(&a,i,j,k,3) =
        *aAdr(&a,i,j,k,3) + b;
(d) (*subsr(s1,k,1) = = 'Q')
```

The generation of these different array access forms *(a)-(c)* is controlled by *COMPILER!SWITCHES /CHECK* or */NOCHECK, SAFE, NOW!SAFE, NOW!UNSAFE* compiler directives.

Status of PSAIL implementation

Because PSAIL can handle most of SAIL and has an extensive error and warning facility, the compiler is relatively large—certainly more than 64K bytes. No attempt is being made to try and squeeze it into today's toy computers since real machines with lots of memory and large disks are becoming increasingly available (at toy machine prices).

The first machine selected for export will be DEC's microVAX, then 68000-, 32016-, or 80286-class machines. Any reasonably sized machine with a decent C compiler is a likely candidate for porting PSAIL. It may also be possible to fit an optimized PSAIL into the limited memory 8088-class machines.

A SAIL code validation test suite and large number of actual working source code programs were and are being used as a compiler construction and debugging tool. These programs represent radically different styles and requirements in areas of numerical analysis, string processing, and data base analysis. Working program sources are extremely useful for finding subtle compiler bugs because, as has often been stated, no compiler is completely bug-free.

Table 2 lists the features currently implemented. It is the author's intention to place PSAIL in the public domain when it is released and to make it available on various bulletin boards and to user groups. (Watch for a *COMPUTER LANGUAGE* Users Group and Bulletin Board Service announcement of PSAIL's availability.) For those interested in working with this emerging, portable SAIL environment, it should be available later this year.