PSAIL: A Portable SAIL to C Compiler - Description and Tutorial

Peter F. Lemkin
Image Processing Section,
Lab. Math. Biology, DCBD,
National Cancer Institute, NIH, FCRF
Frederick, MD 21701

April 29, 1988

ARPA: lemkin@ncifcrf.gov

1. INTRODUCTION

SAIL (1,2) is an easy to use block–structured compiler dialect of the ALGOL–60 programming language. It has been used successfully in writing many large application programs because it has sufficent expressivity and efficiency as well as language 'tools' which help control complexity. It has several powerful language extensions (over basic ALGOL) one of which is the LEAP associative sublanguage (2–4) and dynamic strings. I will attempt to convey some of the flavor and power of SAIL in this tutorial which draws upon the reader's familiarity with C and Pascal–like languages. If you know Pascal then you almost know SAIL. It is a large language with about 300 keywords compared with C's 30 (not counting the C runtime library which is of comparable size). However, SAIL may be divided into much smaller independent sublanguages which are not that difficult to master.

SAIL was written in assembly language at Stanford University Artificial Intelligence Laboratory to run only on DECsystem –10 and –20 computers. To get around this problem, a portable recursive descent SAIL compiler called PSAIL (5) has been written. PSAIL translates a large useful subset of SAIL into a portable subset of C (6,7). Using PSAIL as a preprocessor, C code can be generated for different levels of runtime support. Although a preprocessor increases the cycle time of an edit–compile–link–run–edit loop, using small modules can keep the this time within acceptable bounds. Full PSAIL runtime support includes C runtime functions to support dynamic arrays, dynamic strings with string garbage collection and emulation of many SAIL built–in functions. Alternatively, PSAIL can generate C code without PSAIL runtime support. The code can then be edited lightly to run with a target system's C runtime library. After giving an overview of the relationship of PSAIL to the SAIL language, they will be introduced with a short tutorial.

PSAIL's construction was motivated by need. I had written a large system called GELLAB (8) for the statistical analysis of 2D electrophoretic protein gel images. This system consists of SAIL application programs performing data acquisition, image processing, disk–based data base management, statistical analysis and graphics. When Digital Equipment Corporation announced the demise of the DECsystem–10/–20 series computers, these programs needed to be translated to other machines. Hand translation using a text editor was rejected because it would have introduced too many subtle bugs. A machine translator effort was started to handle the large amount of code involved.

SAIL has been used to write many other large application programs such as the mathematical or statistical modelling of data using a powerful matrix manipulation interpreter with 2D/3D graphics (MLAB), text processing, LR(1) parser generator, nucleic acid sequence and structure analysis, graphics and tape conversion packages as well as many others.

After reviewing several popular target languages including C, Pascal, Ada and MODULA–2, I concluded that SAIL syntax was still better for rapid prototyping during application development yet efficient enough for production as well as developmental systems. This paper will be illustrating why with several short examples. The C++ language on the other hand has more features in common with SAIL - although it has additional object-oriented and generic function features not found in the original SAIL.

Briefly, SAIL has few restrictions on the syntax of compile time in–line macro expansion, strings, arrays, block–structure and procedure argument passing. SAIL protects procedure arguments using expression type coercion where required (C and Pascal do not to the same extent although this has been

added to ANSI C and C++). For example, if a procedure expects a real and you pass it an integer argument, SAIL will coerce the argument into a real before passing it to the procedure (eg. Sin(3) → sin(3.0)). However, arguments passed–by–reference which mismatch the expected types are caught as an error since the procedure would not know how to coerce different instances of referenced variables. Dynamic "ragged" arrays allow lower bounds which can be < 0 or > 1. Dynamic arrays also permit optional bounds checking which aids debugging of exceptional cases. These language facilities let you focus attention on programming algorithms at a higher level and not be deviated by concerns with their low level data structure implementations.

Any language translator mapping from one high–level language to another high–level language can easily introduce inefficiencies. However, if the translator can maintain target code efficiency and is reasonably fast, than it is feasible to use it for developing new software applications. For this reason many semantics preserving optimizations are performed in mapping SAIL to C.

The PSAIL compiler currently runs only on DEC-10/-20 systems. The C runtimes to support PSAIL generated C code have been written but are still being debugged. PSAIL is available on the DEC-10/-20 as a cross–compiler where the generated C code can be lightly edited and used with the libraries of specific target systems. [A validation test suite of SAIL code compiles at about 400 lines/minute on our DEC-2020 (comparable to a VAX/750).] It *is* able to compile itself (PSAIL is written in SAIL). However, the generated C code - can not yet be bootstrapped. When the runtimes are debugged and some remaining bugs fixed in the compiler, it should be possible to bootstrap PSAIL to any C/UNIX system with a good C compiler (i.e. robust and accessing a large address space) and enough memory (about 0.7–1MByte). The SAIL Reference Manual (1) and tutorials (9,10) can also be used with PSAIL. In addition, a PSAIL User Guide and Reference Manual are in preparation.

What is SAIL good for? What are its strengths and in what type of programming environment is it most useful? In analyzing the SAIL code for the above applications, it became apparent that several language features were used extensively. These programs often consisted of many separately compiled modules which were then linked together to make a run–time module. They generally contained many user defined functions and used large numbers of macros and external global variables (i.e. external to the module being compiled). They required efficient numeric and bit-level processing combined with extensive string processing, the latter often being used for effectively acquiring and presenting processed data. Dynamic arrays were also frequently used. Flexible string as well as binary data random access I/O were also used extensively.

The emphasis on what to implement in PSAIL was been determined which SAIL features were most heavily used. These are listed in (5). Of the few restrictions on this implementation of PSAIL, the most important is that some subsets of SAIL were not implemented. Some are partially implemented at the runtime level and some not at all. The associative sublanguage LEAP, processes, events and interrupts *do* have C function call runtime code generated which preserves SAIL semantics. However, no runtimes have been written for them – although they are specified as **extern** symbols in files <psrunL.h> and <psrunP.h>, and skeleton files <psrunL.c> and <psrunP.c> exist and could be fleshed out by users. Currently, contexts, backtracking and record garbage collection are not implemented.

While not ideal, C has sufficient expressivity and runtime efficiency to serve as the translation target language for SAIL code and emulation of SAIL runtimes. C has a portable subset which is widely available in its current weak form. This is being standardized with the X3J11 ANSI standard (7). There are a few restrictions in implementing PSAIL resulting from semantic flaws in C. These include: global GOTO to a label in main() is not allowed from within a procedure; no nested procedure declarations; no bounds checking if C arrays (i.e. non-dynamic) are used; no arbitrary size multi–dimensional C arrays as procedure arguments; SAIL macros are block–structured while C macros are not, so that macros declared in inner blocks exist for the rest of the compilation in C; SAIL's 'Require "file" Load!module' is not always portable. Even given these restrictions, it was anticipated that there would be minimal problems caused by the mismatch of C's capabilities and the semantics required of the translator target language. Where a portability problem is detected, PSAIL issues a warning.

Several upwards-compatible *optional* enhancements to the SAIL language were implemented in PSAIL to maximize use of the C environment and add more modern language concepts. Some of these will be

mentioned later.

2. SAIL/PSAIL TUTORIAL

This tutorial attempts to introduce SAIL and PSAIL using short examples of PSAIL code fragments with discussions illustrating some of the key syntax concepts. Obviously only a small part of SAIL can be introduced here. Because of the terseness of the following examples, a knowledge of C and Pascal–like languages helps. Although SAIL is a large language, it may be broken into naturally distinct subsets which are easy to learn. The SAIL Reference manual (1) is written with this partitioning in mind and has an adequate index. The PSAIL User Guide and Reference manuals detail differences between SAIL and PSAIL as well as giving examples of C code generated by PSAIL.

As in many computer languages, we can use macros as a shorthand where appropriate to save typing and make code easier to read.

```
Comment THIS IS A COMMENT. ;
Define # = "Comment"; # ==> This is another one! <==;
Define CR='15, LF='12, TAB='11, SPACE='40, FF='14, CRLF="CR&LF";
```

In this tutorial *SAIL keywords* start with a capital letter, *user defined symbols* with a lower case letter and *user defined procedures and macros* are all capitalized although SAIL/PSAIL does not care about case distinctions (eg. Rex = rEx = REX = rex etc). Where instructive, the PSAIL–generated C code fragment is also given (indented and with all symbols in lower case. C syntax within discussion is given with a **bolder font**). These *fragments* do not necessarily compute anything useful but *illustrate the syntax of the language*. Often you will see functions in the C code which are not in the standard C libraries – *these are supplied by the PSAIL runtime C libraries* (see Tables 1 and 2).

When the PSAIL compiler is run, it requests a file name followed by a list of any compiler option switches (each preceeded by '/'). For example, to compile a SAIL file *qwerty.sai* viewing the C code generated on the terminal screen and generating a default output file *qwerty.c*, type:

```
RUN PSAIL
?QWERTY.SAI/SCREEN
```

2.1 Changing compiler options during compilation

It is often desirable for experienced programmers to change compiler options in different parts of a program to generate optimized code. This "pragma" declaration is possible using PSAIL statements of the form:

```
Compiler!switches /NOCHECKarray/NOGCcode;
```

This tells PSAIL to generate C–style rather than the default PSAIL runtime dependent dynamic arrays (i.e. */CHECKarray* for dynamic strings and */GCcode* for generating string Garbage Collection, G.C., code).

```
Compiler!switches /REPORT/COUNTFSMs/PRETTYPRINT:72,4/PROFILE;
```

/REPORT directs PSAIL to generate a final report of all warnings, errors, portability problems as well as various types of storage used by PSAIL during compilation. The */COUNTFSMs* switch gathers a histogram profile for the final report of the types of SAIL statements found in the program during compilation. */PRETTYPRINT:72,4* tells PSAIL to output C code indented by block–level with 72 columns/line and 4 spaces/block–level. */PROFILE* inserts C runtime counters and CPU timers for all procedures on your program being compiled. After the program being run terminates, it prints a sorted profile summary of all procedures which can be used for finding procedures which are program bottlenecks.

```
Compiler!switches /UNDF/UNIQUE:8/CLUTTER;
```

/UNDEF directs PSAIL to append */*UNDF*/* to all undefined symbols rather than indicating them by fatal error messages. */UNIQUE:8* causes all visible (i.e. block–structured symbol table) symbols to be tested for *uniqueness* of 8 characters – 6 for **extern**'s is the C ANSI standard. This is especially useful for detecting non–portable symbols. */CLUTTER* finds all variables declared but never used. Various other switches, over 40, are available which are useful for error reporting, debugging, and as portability tools.

2.2 Dynamic arrays and passing variable size arrays as procedure arguments

A dynamic array means that specifying the number of dimensions and index bounds (i.e. range) may be delayed until run time when the actual storage is allocated. Block structure determines an array's *lifetime*. A range is specified as '<lower-bound>:<upper-bound>'. SAIL has no restrictions on passing arrays of variable dimensions and ragged bounds and one can use the 'Arrinfo(arrayName,opr)' function to learn what they are at runtime.

```
Integer Procedure PDQ(Integer i,j; Real Array zippy);
Begin "PDQ"
Integer n;
Define LB1=1, UB1=2, LB2=3, UB2=4;  # -- i.e. Array information operators
Real Array xyz[-i:j,i:2*(i Max j)];   # -- 2D dynamic array with ragged bounds;
External Procedure ABC(Reference Real Array z);
ABC(xyz);   # -- procedure ABC itself can use Arrinfo() to get bounds;
n := Arrinfo(zippy,UB2);   # -- get 2nd dimension upper bound;
Return(xyz[1,2]*xyz[2,1]*zippy[1,n]);
End "PDQ";
```

The SAIL keyword 'Procedure' is used to declare both typed and untyped (i.e. `void`) functions. PSAIL can generate either dynamic array code which requires PSAIL runtimes or C-style arrays which do not by using */CHECK* (default) or */NOCHECK* option switches respectively.

```
Procedure XYZ(Integer i,p,q,r);
Begin "XYZ"
Compiler!switches /CHECK;
Integer Array x[-10:10,15:20,-30:-20];   # -- Do bounds check;
Safe Real Array y[p:q,p:r];   # -- Don't do bounds check;
i := x[p,q,r];
i := i + y[5,6];
Now!safe x; i :=x[p,q,r];
End "XYZ";
```

Generates C code:

```
/*COMPILER!SWITCHES: /check*/
INTEGER /*ARRAY*/ *x;   /* -- Do bounds check */
/*SAFE*/ float /*ARRAY*/ *y;   /* -- Don't do bounds check*/
x = arymkI(3,-10,10,15,20,-30,-20,"x",'i');
y = arymkF(2,p,q,p,r,"y",'f');   /* Create dynamic arrays*/
i = *aChkI(x,3,p,q,r);   /* check bounds, return pointer*/
i += *aAdrF(y,2,5,6);   /* No check, return pointer*/
i = *aAdrI(x,3,p,q,r);   /* No check, return pointer*/
aryfree(x,(INTEGER *)y,(INTEGER *)0);   /* Free List of Dynamic Arrays */
```

The general format of PSAIL runtime functions `arymk$()`, `aChk$()`, `aAdr$()` and `aryfree()` are:

```
arymk$(#bounds, lb1, ub1, lb2, ub2, ..., "printName",'typeCode')
aChk$(array--ptr,#actual-args N, index1, ..., indexN)
aAdr$(array-ptr,#actual-args N, index1, ..., indexN)
aryfree(aryPtr1, ..., aryPtrN, (INTEGER *)0)
```

Dynamic storage is allocated for an array header and data array by `arymk$()` which returns a pointer to the data area part. The '$' is the generic data type (`I` = integer, `F` = float, `C` = string, etc). Note that array y has variable bounds specified at run time by the procedure's arguments. The 'Safe' array declaration modifier omits the bounds check generating `aAdr$()` rather than `aChk$()`. If there is a runtime bounds error detected by `aChk$()`, then a detailed error message is printed. The 'Arrinfo(<array-name>, <inquiry-code>)' runtime function may be used to find out about dimensions, bounds, data type and print name of any dynamic array.

Alternatively, non-dynamic C-style array access code can be produced.

```
Compiler!switches /NOCHECK;
```

```
Integer Array z[2:10,15:20,15:20];
i := z[p,q,r];
```

Generates C code:

```
/*COMPILER!SWITCHES: /nocheck*/
INTEGER /*ARRAY*/ z[11][21][21];
i = z[p][q][r];
```

Arrays which are to be permanent or preloaded with data as with C's initializers are declared as 'Own' arrays.

```
Preload!with
"Ala", # 'A'; "--B--", # 'B'; "Cys", # 'C';
"Asp", # 'D'; "Glu", # 'E'; "Phe", # 'F';
                . . .
"Tyr", # 'Y'; "--Z--"; # 'Z';
Own String Array aminoAcid["A":"Z"];
```

Generates C code:

```
static STRING /*ARRAY*/ *P_aminoacid[26+1*('Z'-'A'+1)]
      ={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
      "Ala", /*'A'*/ "--B--", /*'B'*/ "Cys", /*'C'*/
      "Asp", /*'D'*/ "Glu", /*'E'*/ "Phe", /*'F'*/
            . . .
      "Tyr", /*'Y'*/ "--Z--"/*'Z'*/};
static STRING /*ARRAY*/ *aminoacid = 0;
aminoacid = aryHdrC(&P_aminoacid,1,(INTEGER)'A',(INTEGER)'Z',"aminoacid",'c');
```

PSAIL runtime function `aryHdr$()` is similar to `arymk$()` but uses an existing static C array to store the array header (in the 0's inserted below the data) rather than getting storage from the PSAIL runtime heap. The pointer to the start of the actual data area (i.e. the `"ala"` entry) is returned by `aryHdr$()`. If */FOLDCONSTEXPR* switch is used, it will optimimize the array bound `26+1*('Z'-'A'+1)]` to `[52L]`.

2.3 Dynamic strings

C allows a weak form of dynamic strings. However their maintenance is the C programmer's responsibility. They must request string space from runtime procedures like `malloc()`, copy the string into the new space and return the space using `free()` when finished with the string. SAIL and PSAIL have dynamic strings which *automatically* do storage allocation and deallocation based on need for string space. When a string is no longer needed, as determined by block-structured symbol visability, the string is a candidate for string garbage collection which can recover string space from those strings no longer in use. Although involving more computation, the use of a compacting string garbage collector, allows the SAIL programmer to focus on their application rather than on low level storage allocation problems. The string G.C. `str_GC()` is invoked only when an internal runtime request for a dynamic string is made to internal runtime function `salloc()`and there is not enough contiguous string space left to meet the request. `str_GC()` may also be called by the user if they want to control when collection occurs. PSAIL keeps track of active string pointers by generating C support code for a runtime active–string–pointer–stack and active-blockstack-frame used by the G.C. Because of space limitations, this is not discussed in this paper.

A string constant is any ASCII character string (excluding '\0') enclosed in " (quotes). The " itself may be included by delimiting it with an extra ".

```
"These are string constants"
"Ab\CD""Xy'Z"
```

Generates C code:

```
"These are string constants"
"Ab\\CD\"Xy\'Z"
```

These are mapped to C quoted strings using '\' where needed for characters which are special in C. Since string constant memory space is separate from dynamic string allocation space, string constants are not garbage collected.

Concatenation of two SAIL strings is specified by the infix '&' operator to create a new string. Assigning a string simply copies the string pointer.

```
String s, s1, s2, s3;
s := s1 & s2 & s3;
```

Generates C code:

```
STRING s, s1, s2, s3 = PNULL;   /* PNULL = "\0" */
s = catlist(3L,s1,s2,s3);
```

The `catlist()` function is a PSAIL variable number of arguments runtime function. The C data type `STRING` is defined by

```
typedef char *STRING;
```

String constants which are concatenated are optimized where possible to a single C string constant. Note that octal numbers in SAIL have a single quote ' prefix while C uses a 0 prefix to the digits.

```
"abc" & '15 & '12 & FF & 9 & "xyz"
```

Generates C code:

```
"abc\15\12\14\11xyz"
```

```
s := "Size " & s1 & " and " & s2 & "=" & Cvs(Length(s1&s2&s3));
```

Generates C code:

```
s = catlist(6L,"Size ",s1," and ",s2,"=",cvs(length(catlist(3L,s1,s2,s3))));
```

The `catlist()` and other PSAIL runtime functions which need string space call the PSAIL runtime string allocator `salloc()`. PSAIL's 'Cvs' function converts an integer to a string (eg. 123 → "123") and 'Length' is equivalent to C's `strlen()`. In general, wherever a string expression s is used in SAIL, a substring range expression may similarly be specified by '[...For...]' or '[...To...]'. The keyword 'Inf' in this context indicates `length(s)`. PSAIL runtime functions `subsr()` and `subst()` return substrings illustrated by the following expression fragments.

```
String s, s1;
String Array sA[1:10,-10:30];
Integer ch, i, j, k;
s[j To k]
s[Inf-j To Inf]
s[j+i For k-i+Length(s1)]
"0123456789ABCDEF"[j+1 For 1]
```

Generates C code:

```
subst(s,j,k)
subst(s,length(s)-j,length(s))
subsr(s,j+i,k-i+length(s1))
subsr("0123456789ABCDEF",j+1,1)
```

String constants are treated as any other string. The last example extracts a hex character corresponding to j in the range [0:15].

```
Equ(s,sA[j,k][1 For Length(s)])
```

Generates C code:

```
equ(s,subsr(aAdrC(sa,2,j,k),1,length(s)))
```

and tests if s is in the leading substring of `sA[j,k]` string array element.

```
ch := Lop(s);   # -- remove the first character of s if any;
```

uses the PSAIL runtime C macro:

```
#define LOP(x) ((x==0||*x=='\0') ? '\0' : (*(x++)))
```

When a string expression is used in a numeric expression, SAIL assumes that you want to coerce the first character in the string to an integer operand.

```
i + s
3 + "ABC"
```

Generates C code:

```
i + *s
3 + *"ABC"
```

The latter is optimized in PSAIL to:

```
3 + 'A'/*WARNING*/
```

The /*WARNING*/ indicates that you should check the coercion. Other string optimizations include:

```
Equ("ABC","XYZ")
Equ("ABC","ABC")
Length(s)=0
s[j For 1]
```

Generates C code:

```
FALSE
TRUE
(*s=='\0')
*(s+j-1)
```

SAIL has an extensive string scanning facility for string variables and for scanning input file streams. Up to 54 so called 'break-tables' may be defined which specify characters to be *omitted* and characters on which to *break* as well as the *disposition* of the character which caused the scanning process to break (i.e. stop). The 'Getbreak' and 'Relbreak' functions allocate and deallocate break table numbers. A break table is set up with

```
String brkStr, omitStr, scanControl, s, s1;
Integer brkChar, brkTbl;
brkTbl := Getbreak;   # -- get free break table number;
brkStr := ",+-*/&^_""\=< > ()[]{}~|% ;?:'";
omitStr := TAB&SPACE&CR&LF&FF;   # -- omit form feeds;
scanControl := "INS";   # -- see below ...;
Setbreak(brkTbl,brkStr,omitStr,scanControl);   # -- create table;
```

or

```
Setbreak(brkTbl:=Getbreak, ",+-*/&^_""\=< > ()[]{}~|% ;?:'", TAB&SPACE&CR&LF&FF,"INS");
```

There are various combinations of scan control characters which includes: "I"/"X" – specify break character set by Inclusion/eXclusion; "S"/"R"/"A" – specifies break character disposition should be Skipped/Retained in input string/Appended to output string. The table defined above will break after a token, include '! # $' as token characters, and omit any form feeds. It skips the break character. Eg.

```
s := "APPLE ! sauce + PEAR=ORANGE";
s1 := Scan(s, brkTbl, brkChar);
```

will scan "APPLE!sauce" into s1, '+' into brkChar, and s will then contain " PEAR=ORANGE".

2.4 Embedded assignment and auto assignment optimization

SAIL, like C, allows embedded assignments. An assignment returns a value so it is also an expression. It may be used as a value in another assignment with appropriate type coercion generated automatically.

```
Long Real aaa,bbb;   Real a,b;   Integer i,j,k;   String s,s1;
aaa := i := a := s := b := j := s := k := bbb;
s := i := a := s1 := b := s2 := i;
aaa := 1 + 1.2 + "A" + "AB" + s1 + bbb;
a Swap b;   i Swap j;   a := Abs b;   k := i Max j;
```

Generates C code:

```
float a, b;
double aaa, bbb;
INTEGER i, j, k;
STRING s, s1;
aaa = i = a = (float)(*s = makstr((INTEGER)b = j = (*s = makstr(k = bbb))));
s = makstr(i = a = (float)(*s1 = makstr((INTEGER)b = (float)(*s2 = makstr(i)))));
```

```
aaa = 1 + 1.2 + 'A' + ('A'/*WARNING*/) + *s1 + bbb;
SWAPF(a,b);  SWAPI(i,j);  a = ABS(b);  k = MAX(i,j);
```

SAIL's 'Swap', 'Abs', 'Max/Min' get mapped to macros in <sairun.h> for speed.

Auto assignment optimization was added to PSAIL since C has auto increment/decrement and auto operator assignment when the <lValue> is a variable. Therefore it handles cases of the form

```
<lValue> := <lValue> +/- <rValue>;
```

when <rValue> is 1 the expression is optimized to generate C code:

```
++<lValue>  or  --<lValue>
```

For <rValue> not 1 and operator +, -, *, / or Mod, it generates C code:

```
<lValue> += <rValue>
```

Assignment optimization is useful for scalars but is even more useful for arrays. A defect in SAIL is that it does not do this optimization as often as it could. This is corrected in PSAIL,

```
abc[i,j,k,l] := abc[i,j,k,l]+1
```

Generates C code for the form:

```
        ++abc[i][j][k][l]                  /* with /NOCHECKARRAY */
or      ++*aAdr$(abc,4,i,j,k,l)            /* with SAFE arrays and /CHECKARRAY */
or      ++*aChk$(abc,4,i,j,k,l)            /* with unsafe arrays and /CHECKARRAY */
```

Similarly,

```
abc[i,j,k,l] := abc[i,j,k,l]+p
```

Generates C code:

```
        abc[i][j][k][l] += p               /* with /NOCHECKARRAY */
or      *aAdr$(abc,4,i,j,k,l) += p         /* with SAFE arrays and /CHECKARRAY */
or      *aChk$(abc,4,i,j,k,l) += p         /* with unsafe arrays and /CHECKARRAY */
```

2.5 SAIL records are mapped to C structures

The 'Record!class' and 'Record!pointer' declarations in SAIL are mapped to struct and struct pointer declarations in C. SAIL and PSAIL check Record!pointer variables for the Record!class with which they were defined. However, it also allows the 'Any!class' wildcard Record!class to defeat this check. Although SAIL allows a particular Record!pointer to be defined for a *list* of Record!classes, PSAIL does not since there is no simple mapping to C structs. Currently, the string garbage collector will not work correctly if strings are declared in records.

```
Record!class fish (Record!pointer (fish) zog;
                   Real x, y);
Record!pointer (fish) Array zim[1:4];
Record!pointer (fish) rp1, rp2;
zim[3] := rp2 := New!record(fish);
fish:x[rp2] := 1.234;
rp1 := fish:zog[zim[3]];
fish:x[rp1] := 3.14;
Del!record(rp2);
```

Generates C code:

```
static struct *fish {struct *fish zog;
                     float x, y;};
struct *fish /*ARRAY*/ *zim = 0;
struct *fish rp1, rp2 = 0;
zim = arymkQ(1,1,4,"zim",'q');   /* Dynamic Array */
(*aChkQ(zim,1,3)) = rp2 = new_record((12)/*(fish)*/);
```

```
rp2/*(fish)*/->x = 1.234;
rp1 = (*aChkQ(zim,1,3))/*(fish)*/-->zog;
rp1/*(fish)*/-->x = 3.14;
del_record(rp2);
```

The PSAIL 'New_record(#bytes)' and 'Del_record(ptr)' runtime functions allocate and deallocate record storage from the PSAIL runtime heap using `Pmalloc()` and `Pfree()`.

2.6 Data type conversion

In addition to the implicit data type conversion just discussed, SAIL offers a set of explicit string data type conversion functions listed in Table 1. You can use these when you may not want the compiler to use the implicit conversion mappings. These functions are especially useful when one wishes to compose a string from several different types of variables.

```
Integer i, j, brkChr, width, precision, oldW, oldP;
Real x, y, z;
String s, s1;
Getformat(oldW,oldP);   # -- get previous width and precision;
Setformat(width := 13, precision := 8);   # -- default values;
s1 := "S1";   i := j := 8;   x := 3.3;   y := 4.4;   z := 5.5;
Setformat(4,2);   # -- redefine width and precision;
s := s1 & "," & Cvs(i) & "," & Cvos(j) & "," & Cvf(x) & "," & Cve(y) & "," & Cvg(z);
Setformat(oldW,oldP);   # -- Restore previous width and precision;
```

would generate the string contained in s:

```
"S1, 8, 10, 3.30, .44E1, 5.5"
```

Where 'Cvs', 'Cvos' convert integers into decimal and octal strings, while 'Cvf', 'Cve' converts reals into decimal and scientific notation strings and 'Cvg' does the smaller of 'Cvf' and 'Cve'. The conversion width and precision are specified globally by 'Setformat' and read by 'Getformat' offering functionality similar to the C `sprintf()` format specification.

```
x := Cvd(s); j := Cvo(s);
```

convert integer strings to decimal and octal numeric values like C's `atof()` and `atoo()`.

```
i := Intscan(s,brkChr1);   x := Realscan(s,brkChr2);
```

scan numbers from a string `s` showing the character `brkChr1` and `brkchr2` which causes the scan to stop when the remaining string is no longer an integer or real number respectively. Then, the remaining string `s` no longer has the leading substring which was scanned away.

```
s := "123.456E-3 AND CATS OR 314e-2 OR 27.@-1";
x := Realscan(s,brkChr2);
```

will set `x` to the real value '0.123456', `brkChr2` will contain a space, and `s` the string `" AND CATS OR 314e-2 OR 27.@-1"`.

Automatic type coercion is performed in expressions and procedure calls where required. Although almost as controversial as the GOTO problem, automatic type coercion has its advocates (C++ and many SAIL users among them). It forces expressions into the *type necessary to perform the operation*. Sometimes this is not what was intended, but more often it is.

```
String s;   Integer ch;
If "A" Leq s Leq "Z" Then ch := s - "A";
```

Generates C code:

```
if ('A' <= *s && *s <= 'Z') ch = *s - 'A';
```

Type coercion is also used to force actual procedure arguments to match the type expected in the procedure call. Coercion is *not performed* if the argument is passed by reference – in which case the mismatch error is caught by PSAIL.

```
External Procedure ABC(Real x);
External Procedure XYZ(Reference Real x);
```

```
Integer j;
ABC(j);
XYZ(j);
```

Generates C code:

```
abc((float)j);          /* note generation of C cast */
```

`XYZ(j)` will be caught as a type mismatch because of the 'Reference' attribute.

2.7 Memory access using Memory[] and Location() as with C's '*' and '&'

SAIL offers direct access of memory though integer pointers similar to C. However unlike C, SAIL does not allow an arbitrary number of '*' type declarators in its declarations. Instead you are responsible for keeping track of pointer chains. Since the other dynamic data structures of SAIL are more powerful than those of C, this construct is not used very often.

```
Integer p,q,j;
Real x;
p := Location(j);              # -- access an integer as a real;
x := Memory[p+5,Real];
q := '37777700632;             # -- some memory mapped address;
j := Memory[Memory[q],Integer];
```

Generates C code:

```
p = &j;                              /* -- access an integer as a real*/
x = (float)(*(p+5));
q = 037777700632;                    /* -- some memory mapped address */
j = (INTEGER)(*(*(q)));
```

The 'Memory[]' construct lets you optionally *cast* the type of the contents of the pointer.

2.8 Bit manipulation of integer data is similar to that in C

Integers may be thought of as an ordered set of 32–bits which can be individually manipulated.

```
Integer a,b,c;
c := Lnot a;   # -- Bit complement;   c := Not a;  # -- Boolean negation;
c := a Land b; # -- Bit And;    c := a And b; # -- Boolean conjunction;
c := a Lor b;  # -- Bit Inclusive Or;  c := a Or b;  # -- Boolean disjunction;
c := a Xor b;  # -- Bit Exclusive Or;
```

Generates C code:

```
c = ~ a;     c = ! a;
c = a & b;   c = a && b;
c = a | b;   c = a || b;
c = a ^ b;
```

```
c := a Lsh b;   # -- Logical bit shift of a by b bits;
```

Generates C code:

```
        c = a << b;      /* For + b if true logical shift in C */
or      c = a >> b;  /* For - b if true logical shift in C */
or      c = lsh(a,b);    /* For + b or +b var use C function */
or      c = rsh(a,b);    /* For - b or -b var use C function */
or      c = LSH(a,b);    /* If when building PSAIL compiler you
                            * specify that it map Lsh to LSH/RSH
                            * macro which in turn does one of the
                            * above mappings but with end-bits protected. */
```

```
c := a Rot b;   # -- Rotate a by b bits (+left/-right);
c := a Ash b;   # -- Arithmetic shift of a by b bits;
```

Generates C code:

```
c = rot(a,b);   /* -- Rotate a by b bits */
c = ash(a,b);   /* -- Arithmetic shit of a by b bits */
```

2.9 Compiler macro expansion with arguments

SAIL compile–time macros are mapped to C **#define** macros where possible. However, when there is not enough information to delay evaluation until C compilation, instances of macros are expanded during compilation by PSAIL.

```
String ss;   Integer jj;
Define PRINT!IT(a,b) = "PRINT(""a="",a,"" b="",b,CRLF)";
PRINT!IT(ss,jj);
```

Generates C code:

```
/* #define print_it(a,b) print("a=",a," b=",b,crlf) ==> PSAIL!EVAL */
PSprintf("%s%s%s%l%s","ss=",ss," jj=",jj,crlf);
```

It does not generate a C '#define' statement and occurances of the macro will be expanded in PSAIL since types **a** and **b** are ambiguous. A PSAIL extension allows you to specify the parameter types to aid the compiler and thus let it generate the C **#define** statement and coerce arguments in actual macro calls to the correct arguments.

```
Define FAST!EQU(String p,q) = "(((p)=(q)) And EQU((p),(q)))";
```

Generates C code:

```
#define fast_equ(p,q) ((*p==*q) && equ(p,q))
```

Unlike C, the use of optional explicit macro body delimiters in SAIL permits unambiguous use of quotes or any other characters. Changing the macro body delimiters from quotes (see '{}' below) allows one to easily nest macro declarations. Otherwise, SAIL macros are similar to C macros except that no continuation character (such as '\' for C macros) is required for SAIL macros longer than one line.

```
Define XWORD(x) = "line[(x) Lsh -2]";
Define GETBYTE(byte,line,x) "{}" = {Case ((x) Land '3) Of
      Begin
      "0" byte := (XWORD(x) Lsh -8) Land '377;
      "1" byte := (XWORD(x)) Land '377;
      "2" byte := (XWORD(x) Lsh -24) Land '377;
      "3" byte := (XWORD(x) Lsh -16) Land '377;
      End};
```

PSAIL *does* append '\' to successfully translated multi–line C macros as required by most C compilers.

2.10 Conditional compilation

SAIL has conditional compilation like C's **#if <expression>**, but does not have the exact equivalent of **#ifdef** or **#ifndef** - although 'IfC Declaration(x)' is functionally equivalent. 'IfC' is semantically closer to C's **#if <expression>**.

```
Real testType;
Define TTYPE = "String";
IfC Declaration(testType)=Check!type(Integer)
      ThenC Redefine TTYPE="Integer";
ElseC IfC Declaration(testType)=Check!type(Real)
      ThenC Redefine TTYPE="Real";
EndC;
```

You may inquire about the data type of a declared variable using compile time function 'Declaration()' which may then be compared with that of legal SAIL declaration types using 'Check!type()'. In the above example this enables the 'TTYPE' macro to be redefined differently depending on the outcome of the test (in this case – "Real"). Alternatively, using the PSAIL extension 'C!code', one could use the **#ifdef** or **#ifndef** of the C compiler.

```
C!code
#ifndef tammy
#include <tammy.h>
#endif
```

```
End;
```

2.11 Modules and separate compilation

The SAIL 'External' declaration storage class declares a symbol which exists in another separately compiled module (i.e. C's **extern**). SAIL's 'Internal' storage class declares a symbol global (like C's non-**static** top level declaration) so that *other* modules can access it using 'External'. Note (below) that an external procedure's argument-list declarations as well as bounds of external arrays are also exported across modules so that all necessary information is available to the SAIL compiler to check type mismatches and generate dynamic array bounds checking code.

```
Require "MACROS.REQ" Source!file;  # -- add macros to symtab, gen. C '#include <macros.h>' ;
Require "GLOBALS.REQ" Source!file;  # -- add externs to symtab, gen. C '#include <globals.h>' ;
Evalinclude "NEEDED.SAI" Source!file; # -- always include code here;
Internal String magicStr;      # -- declare in this module;
External Integer Procedure XYZZY; # -- declared in other module;
Forward String Procedure AMPHOLYTES(Real acidPH, basicPH; Integer p(10));  # -- default 10;
External Integer Procedure PARSE(String s1; Integer k);
External Safe Real Array value[-10:100,4:50];  # -- note array bounds;
Internal Define IFNDEF(x)="IfC Not Declaration(x)"; # -- Emulate C's #ifndef;
Internal String Procedure CVTS(Integer dw; Integer Procedure Q; Real Array pdq);
      Begin "CVTS"
      String s1;  Integer k;
      k := dw + Q;  # -- note now EVAL procedure variable Q;
      pdq[k,k-1] := CVD(s1);  # -- use dynamic array formal arg with bounds checking;
      s1 := PARSE("eval",k) & magicStr;
      Return(s1&Cvf(value[0,dw]));
      End "CVTS";
CVTS(10,XYZZY);   # -- pass name of procedure to be evaled;
AMPHOLYTES(3.0,8.5);   # -- use default 10 for missing argument;
AMPHOLYTES(5.0,7.6,3);   # -- overide default with 3%;
```

Often, macros and External declarations are useful SAIL code to include with Require-Source!file-statements. With the default */REQUIRE* switch, 'Require-Source!file' statements map to C **#include** statements for later evaluation by the target C compiler. The */NOREQUIRE* switch forces these files to be included and compiled by PSAIL. Individual files may selectively always be included using the PSAIL extension 'Evalinclude' in place of 'Require'. Note that formal procedure arguments may include the 'Procedure' modifier which then causes the procedure name to be passed-by-reference and not be evaluated until desired in the procedure body (see Q above).

A PSAIL extension, the */MAKEREQUIREFILE* switch, specifies the *creation* of an auxillary file which contains a set of 'External' declarations corresponding to *all* instances of 'Internal' declarations, including 'Internal Define ... macro statements, found when compiling the source file. The generated list of External declarations can then be 'Require'd by other SAIL modules – similar in concept to Modula-2's DEFINITION modules and IMPORT statements. When used with the */EXTENSIONS* switch (to be discussed), one can then search for and *selectively import* external declarations and macro variables from this file. Using the previous example (file *xyzzy.sai*) compiled with the */MAKEREQUIREFILE* switch would generate *xyzzy.rqu*. Then we could import selected variables by:

```
From "xyzzy.rqu" Import magicStr, CVTS, IFNDEF;
```

2.12 Flexible block-structure

Whereas Pascal and MODULA-2 do not allow declarations inside of nested blocks, SAIL (and C) do. SAIL also optionally allows the naming of blocks so that one can exit or continue to *outer* blocks. If blocks are named, then the names must match for 'Begin' and 'End'. This is very useful as a consistency check for complex nested blocks as well as for transfering control to outer blocks using SAIL's labeled 'Continue', 'Done' or 'Next' statements.

```
Real r,x,y; Integer i,j,k; Boolean flg; Label myExit;
flg := TRUE;
While i<4 Do
      Begin "BLOCK LEVEL 1"
      Own Integer q1;
      For r := i Step 0.1 Until 3.14159 Do
            If Sin(q1:=q1+1) Leq 0.5
                  Then
                  Begin "BLOCK LEVEL 2"
                  Own Integer q2;
                  q2:=q2+1;
                  For x := r Step 0.001 Until r+0.099 Do
                        Begin "BLOCK LEVEL 3"
                        Integer q3;
                        For y := 9.1 Step -Sin(x) While y<q2 Do
                              q3:=q1+q2+y;
                        If q3 > 501 Or x > y+0.2 And Not flg Then Done;
                        If x < y-0.3 Then Done "BLOCK LEVEL 2";
                        If x = 2.7 Then Continue;
                        If x = 2.3 And flg Then Continue "BLOCK LEVEL 1";
                        If x = 1.7 Then Next;
                        If x = 1.3 Then Next "BLOCK LEVEL 2";
                        flg := Not flg;
                        If x = 1.3 And y = 3.14 Then Goto myExit;
                        End "BLOCK LEVEL 3";
                  End "BLOCK LEVEL 2";
      myExit:
      i := i+1;
      End "BLOCK LEVEL 1";
```

PSAIL maps the 'Done', 'Continue' and 'Next' block exit statements to C's **break**, **continue** and **goto** statements. When referring to *outer* blocks, PSAIL maps these statements to **goto <label>**'s which are generated by PSAIL at 'End' block boundaries – eg. **PS0001, PS0002,** etc. This was necessary since C does not have this facility with its **break** and **continue** statements. Unlike C, user defined labels in SAIL *must* be explicitly declared *before* they are used with 'Goto'. The labeled Done–statement is usually adequate so the 'Goto' is rarely used to exit blocks in SAIL.

2.13 Conditional control statements

SAIL If-statements are similar to those of C and Pascal with somewhat more flexibility because they allow Boolean expressions which can contain embedded assignments (like C). In addition, SAIL allows complex relational expressions of the form (a < b > c Leq d Geq e) which are expanded to conjunctive C expressions of the form:

```
(a < b && b > c && c <= d && d >= e).
```

and

```
String s;   Integer ch, i;   Boolean letFlg;
If ("A" Leq (ch := Lop(s)) Leq "Z") And letFlg
      Then i := ch-"A"
Else If Not letFlg And ("0" Leq ch Leq "9")
      Then i := ch-"0"+26
Else i := -1;
```

Generates C code:

```
if (('A'<=(ch=LOP(s)) && ch<='Z') && letflg) i= ch-'A';
else if (!letflg && ('0'<=ch && ch<='9')) i= ch-'0'+26;
```

```
else i= -1;
```

The Case–statement of SAIL is similar to the C **switch** statement. C requires the word **case** for each statement whereas SAIL does not. SAIL uses 'Else' whereas C uses **default** for the last case. The first of two forms of the SAIL Case–statement does not use explicit case-index statement indices but assumes sequentially numbered indices starting at 0.

```
Integer idx;
Real x,y;
Case idx Of
      Begin "list of statements"
      x := y;    # -- case 0 is assignment;
      x := Sin(y);    # -- case 1 is sin function;
      x := Cos(y);    # -- case 2 is Cos function;
      Begin "case 3 is Y factorial function"
      Integer i;
      x := 1;
      For i := y Step -1 Until 1 Do x := x*i;
      End "case 3 is Y factorial function";
      Else Usererr(0,0,"Illegal operation: "&Cvs(idx));
      End "list of statements";
```

Each Case–statement element can be any SAIL statement including a compound statement (i.e. a block). The alternate form of the Case–statement uses explicit labels (in '[]' similar to the value following C's **case**).

```
Case idx Of
            Begin "list of statements"
      [5] [14]          x := Sin(y);
      [7] ["Q"]         x := Cos(y);
            End "list of statements";
```

Generates C code:

```
switch (idx)
            {/*list of statements*/
      case 5: case 14:          x = sin((double)y); break;
      case 7: case 'Q':         x = cos((double)y); break;
            }/*list of statements*/
```

Unlike the C **switch** statement, no 'Done' statement is required in SAIL to exit each Case instance – although C **break**'s are added by PSAIL to each statement in the list.

2.14 Conditional expressions

SAIL has conditional expressions in the form of If– and Case– expressions functionally similar to that of C's **(expr) ? e1 : e2**.

```
Real a,b,x;
x := If a >= b Then Sin(a) Else Cos(a);
```

Generates C code:

```
x = (a >= b) ? sin((double)a) : cos((double)b);
```

Note PSAIL's casting of actual function arguments to the expected type. In addition to If–expressions, SAIL has a Case-expression:

```
String s;
Integer i;
s := Case i Of ("A","C","G","T",99,i);
```

Generates C code:

```
s = (i==0) ? "A"
      : (i==1) ? "C"
```

```
: (i==2) ? "G"
: (i==3) ? "T"
: (i==4) ? makstr(99L)
: (i==5) ? makstr(i)
: 0;
```

The type of If– and Case– expressions is the type of the *first* expression in the list. All subsequent expressions are coerced to the type of the first.

2.15 Iterative control statements

SAIL has both test–before (While <boolExpr> Do <statement>) and test–after (Do <statement> Until <boolExpr>) type loops.

```
While i < 5 Do
      Begin "loop" i := i+1; j := j+1; End "loop";
```

and

```
Do Begin "loop" i := i+1; j := j+1; End "loop"
Until i >= 5;
```

There are several variations of the For–loop.

```
For i := initVal Step 1 Until lastVal Do <statement>;
For i := initVal Step -1 Until lastVal Do <statement>;
For i := initVal Step incrVal Until lastVal Do <statement>;
For i := initVal Step -incrVal Until lastVal Do <statement>;
For i := initVal Step incrVal While boolExpr Do <statement>;
```

Generates C code:

```
for(i=initval; i<=lastval; i++) <statement>
for(i=initval; i>=lastval; i--) <statement>
for(i=initval; i<=lastval; i+=incrval) <statement>
for(i=initval; i>=lastval; i-=incrval) <statement>
for(i=initval; boolexpr; i+=incrval) <statement>
```

Note that the various limit values may be integer or real and positive or negative *expressions*. PSAIL allows explicit For–lists using *any* data type. The following example uses a list of strings.

```
Integer cnt; String s, sL, sFruit;
cnt := 0; sL := NULL; sFruit := "peach";
For s:= "apple", "orange", sFruit, "pear" Do
      sL := sL & " " & Cvs(cnt:=cnt+1) & " " & s;
```

Generates C code:

```
{INTEGER is;
STRING ivals[5];
ivals[1] = "apple";
ivals[2] = "orange";
ivals[3] = sfruit;
ivals[4] = "pear";
for (is=1,s=ivals[1]; is<=4; is++, s=ivals[is])
      sl = catlist(5L,sl," ",Cvs(++cnt)," ",s);
}
```

Running this program computes `sl` equal to:

```
" 1 apple 2 orange 3 peach 4 pear"
```

2.16 Flexible I/O

SAIL has flexible ASCII string, binary and random access binary I/O. It performs ASCII string sequential I/O with character scan control similar to C's `printf()`, `scanf()`, `fprintf()` and `fscanf()`.

```
String s; Integer k; Real x;
```

```
External Integer Procedure SCOUNT(String s);
External Real Procedure SVALUE(String s);
Outstr(CRLF&"*");  # -- output string to the terminal;
Ttyup(TRUE);  # -- on input, convert lower to upper case;
s := Inchwl;  # -- input string line<==terminal;
k := SCOUNT(s);   x := SVALUE(s);
Print("'",s,"' has count ",k," and value ",x,CRLF);
```

Generates C code:

```
PSprintf("%s%c",crlf,'*');  /* -- output string to the terminal */
ttyup(TRUE);   /* -- on input, convert lower to upper case */
s = inchwl();   /* -- input string line from terminal and wait for LF*/
k = scount(s);   x = svalue(s);
PSprintf("%c%s%s%l%s%f%s",'\'',s,"\' has count ",k," and value ",x,crlf);
```

SAIL's 'Inchwl', 'Outstr', and 'Print' are a few of the terminal I/O functions. A portable `printf()` function `PSprintf()` is included in the PSAIL C runtimes. Output field width and precision used by the `PSprintf()` function is set by the previous 'Setformat' call – unlike C's `printf()` which must be set explicitly in each control format entry specification. The following example copies a file *twiggy.txt* into *ziggy.txt* omitting all digits. A SAIL I/O channel is functionally equivalent to C's file descriptor.

```
Integer blkNbr, brkChr, brkTbl, eof, err, channel, ioMode, maxChrCnt, nIbuf, nObuf, wordCnt;
String s, iFile, oFile;
iFile := "TWIGGY.TXT";   oFile := "ZIGGY.TXT";
ioMode := 0;  # -- ASCII sequential;
maxChrCnt := 1000;  # -- max # characters on single input;
nIbuf := nObuf :=6;  # -- 6x512 character input & output buffers;
Open(channel:=Getchannel,"DSK",ioMode,nIbuf,nObuf,maxChrCnt,brkChr,eof);
If eof Then Usererr(eof,0,"Can't open DSK device.");
Lookup(channel,iFile,err);  # -- assign input file name here - unlike C's open();
If err Then Usererr(err,0,"Input file "&iFile&" not found.");
Enter(channel,oFile,err);  # -- assign output file name here - unlike C's open();
If err Then Usererr(err,0,"Output file "&oFile&" create error");
Setbreak(brkTbl := Getbreak,LF,"0123456789","INS");
While Not eof Do
      Begin "copy loop"
      s := Input(channel,brkTbl);  # -- break on LF, omit digits;
      Out(channel,s&LF);  # -- output reduced string;
      End "copy loop";
Release(channel);  # -- close files and release channelnel;
Relbreak(brkTbl);  # -- release break table;
```

The 'Input' function reads a string from the channel up to the break character (LF in this case) while 'Out' just outputs the string to the channel. SAIL also allows binary sequential buffered I/O which is similar to C's `fread()` and `fwrite()`. The SAIL 'Open' call would have been made with `ioMode` set to '10 (octal 10).

```
Integer Array buf[0:999];
While Not eof Do
      Begin "copy file"
      Arryin(chan,buf,wordCnt:=1000);   Arryout(chan,buf,wordCnt);
      End "copy file";
```

or

```
While Not eof Do
      Wordout(chan,(Lnot Wordin(chan));  # -- complement binary data in file;
```

Random access unbuffered binary I/O using SAIL's 'Useti' or 'Useto' is similar to C's `lseek()` except that the relative block number rather than a byte count must be specified. The SAIL 'Open' call would

have been made with ioMode set to '17.

```
Useti(chan,blockNumber);  # -- set input block position before read;
Arryin(chan,buf,wordCnt);  # -- read multiple of 128 32-bit words;
```

2.17 LEAP associative sublanguage - examples of some features

A full discussion of the LEAP sublanguage is given in (1–4). Leap declarations may be typed and untyped sets, lists, items, and item variables (itemvars) with the types being the basic types of SAIL.

```
Set xS, yS;  List xL, yL;  Item x, y;  Itemvar xV, yV; # -- Untyped;
String Item x1, y1;  Integer Itemvar x2, y2; Real Itemvar x4;  # -- typed;
Real Array Itemvar x3, y3;  # -- i.e. Array of Real Itemvar's;
Itemvar Array String Array sV[1:123]; # -- i.e. Itemvar array of String arrays;
Integer i, existsFlag;  Real r;  String s;
Integer Array spood[-256:255,512:1024];
```

Sets, lists and new items can be created and deleted dynamically and have global scope.

```
xS := {x, y};  yL := ((x, y));
Put x1 In xS;  Put x1 In yL Before y;
Put x2 In xL After x;  Put x3 In yL Before 2;
xV := New(r);  Put xV In xS;
Remove xV From xS;  Delete(xV);
```

The 'Datum' operator is used to access the *value* property of items and may be considered the same as a C <lValue> pointer variable but of type associated with its argument.

```
Datum(x4) := 3.14159;  Datum(y1) := s := Datum(sV[3])[7];
r := r + Datum(x4) - Datum(x3)[1,3];
s := s & " has string value =" & Datum(x1);
i := Typeit(xV);  # -- get type of item stored in itemvar;
If Typeit(xV)=Check!type(String) Then s:=Datum(xV)
Else If Typeit(xV)=Check!type(Real) Then r:=Datum(xV)
Else i:=Datum(xV);
```

Created items are initially copies of primary data structures and may acquire string *print* names.

```
x2 := New(k);  yV := New(spood);
New!Pname(x2,"Int-value");  New!Pname(yV,"2D-Array"); # -- Set item print names;
Print("Item print names are: ", Cvis(x2,existsFlag), " and ", Cvis(yV,existsFlag), CRLF);
Print("Value of ",Cvis(x2,existingFlag),"=",Datum(x2),CRLF);
```

'New!Pname assigns print names for item variables, while 'Cvis' retrieves the string print name of an item. 'Cvsi' looks up the item corresponding to a string (is possible print name). Sets or lists may be tested for item membership.

```
If (xV In xS) And (Not yV In yS) Then ...
```

Intersection, union, differences and concatenation may be performed on sets and lists.

```
xS := xS Inter yS;   xS := xS Union yS;   xS := xS - yS;   xL := xL & yL;
```

Associative triples '[attribute Xor object Eqv value]' may be created, deleted or retrieved given partial information of their association.

```
Item animal, fish, tuna, trout, mammel, camel, cat, dog;
Make animal Xor xV Eqv x1;  Make animal Xor fish Eqv tuna;  # -- create associative triples;
xV := [animal Xor xV Eqv x1];  # -- Retrieve a matching triple item;
xS := animal Xor Any;  # -- Derived set {animal Xor Any Eqv ?};
xS := animal Eqv tuna;  # -- Derived set {animal Xor ? Eqv tuna};
Erase animal Xor xV Eqv x1;
```

Associative search may be performed on sets, lists or triples.

```
Real Itemvar clue1, clue2;
```

```
Real Item keys, apples, chair, bed, gasoline, avocados;
Set house, car, foundIt, belongsIn, mapOfCalif;
foundIt := Phi;  # -- i.e. Null Set;
house := {keys, bed, chair, apples, car, avocados};
car := {keys, gasoline, mapOfCalif};
Foreach xV Such That xV In house Do
      Make belongsIn Xor house Eqv xV;
Foreach xV Such That xV In car Do
      Make belongsIn Xor car Eqv xV;
... code to stuff Datums and define print names ...
Foreach clue1,clue2 Such That
      (belongsIn Xor clue1 Eqv clue2) And
      (Datum(clue1) > Datum(clue2)) And
      (clue1 In car) And
      Not (clue2 In car) Do
            Put clue1 In foundIt;
While Length(foundIt) > 0 Do
      Print("Found Item ",Cvis(Lop(foundIt))),CRLF);
```

The 'Foreach' search generates item instances in the itemvars which can then be used by the statement following the 'Do'. In this case it builds a set of clue items in set `foundIt`. When used with sets or lists, function 'Lop' eats up set or list item elements returning the next element (item) each time it is called as well as shortening the set or list.

3. EXAMPLES OF SOME PSAIL EXTENSIONS

There are a number of PSAIL extensions which are invoked with the */EXTENSIONS* switch. These are justified in that they offer dramatic optimization of the PSAIL runtime C environment and in aiding algorithm expressibility by incorporating more modern language concepts. The default */NOEXTENSIONS* allows full code compatibility with older SAIL programs.

Sublanguages may be changed and extended using 'Psail!forget' to remove parts of the SAIL language and 'Psail!define' to add new syntax and corresponding runtime functions using *existing* FSM parser capability. For example – a functional equivalent of 'Print' called 'Ttywrite' could be implemented by the PSAIL statement to give 'Ttywrite' the same FSM parsing semantics as 'Print':

```
PSAIL!DEFINE "L4:TTYWRITE,PSprintf,32,:u:1,-1";
```

'Generic' procedures let the data type of the procedure's first argument dictate which procedure is actually called. 'Repeatable' procedure arguments allow a variable number of arguments. 'Untyped' procedure arguments lets you abandon type checking/coercion. The 1–Dimensional array-slice assignment

```
abc[i:i+k] := xyz[j:j+k];
```

found in other languages is useful for manipulating chunks of arrays. Also available are the C-style operators: `++`, `--`, `+=`, `-=`, etc. Pascal type record access is also allowed by mapping PSAIL syntax of the form 'recPtr.recMember' to SAIL's 'recClass:recMember[recPtr]' prior to final parsing. The 'C!array' modifier used in place of SAIL's 'Array' accesses dynamic arrays as if they were more efficient C–style arrays. A few of these extensions are illustrated in more detail.

3.1 Generic procedures

'Generic' procedures are useful for hiding lower levels of abstraction when dealing with different argument data types for a single procedure name. A different procedure is actually called to handle each data type.

```
External Generic(ZOP) Integer Procedure ZOPBITS(Reference Integer x);
External Generic(ZOP) String Procedure ZOPSTRING(Reference String x);
External Generic(ZOP) Set Procedure ZOPSET(Reference Set x);
External Generic(ZOP) List Procedure ZOPLIST(Reference List x);
```

associates the generic symbol 'ZOP' with the list of non-generic procedures (ZOPBITS, ZOPSTRING, ZOPSET, ZOPLIST). When ZOP(j) is encountered, it would: 1) find the type of the first argument 'j', 2) match it with the corresponding argument type of the non-generic procedure in the associated procedure-list for ZOP, and 3) substitute the matching non-generic procedure name for ZOP. Evaluation then continues with normal type checking and coercion performed on the remaining arguments (if any) of the new function name.

```
Integer i,j;
i := ZOP(j);
```

Generates C code:

```
i = zopbits(j);
```

If there are no procedure arguments, then an instance of a generic procedure will pick the first associated procedure name. This permits simple name mapping. Eg.

```
External Generic (TTYREAD) String Procedure INCHWL;
```

Additional procedure argument type modifiers include: 'Untyped' to disable type checking for the next argument, and 'Repeatable' to allow a *variable* number of arguments. Eg.

```
Procedure ABC(Untyped x; Repeatable String s);
ABC(1,"INT-type");  ABC(3.14159,"REAL","-type");
```

3.2 C!array optimization

Dynamically created PSAIL arrays may be evaluated as C-style arrays in selected parts of a program for efficiency. This is feasible since the array pointer is the same for both access methods and points to the start of the data area of the dynamic array. It is only possible to do this automatically for 1-dimensional arrays which start at index 0 as the C compiler would have no information about the other dimensions. Any array where one wants to generate C-style access code should be declared with 'C!array' instead of 'Array'. The following illustrates how this might be used:

```
Compiler!Switches /NOCHECK;  # -- disable dynamic arrays;
Procedure TEST!C!ARRAYS(Real C!Array dodoBuf; Real Array impalaBuf);
Begin "TEST!C!ARRAYS"
Compiler!Switches /CHECK;  # -- enable dynamic arrays;
Integer Array slowBuf[0:511], turtleBuf[0:511];
Integer C!Array fastBuf[0:511], zipBuf[0:511];
Integer i, j, size;  Real x;
size := 511 Min Arrinfo(impalaBuf,2);  # -- get UPPER BOUND;
For i := 0 Step 1 Until size Do
      Begin "Go with the wind"
      fastBuf[i] := zipBuf[i];
      fastBuf[i] := impalaBuf[i];
      End "Go with the wind";
size := 511 Min Arrinfo(dodoBuf,2);  # -- get UPPER BOUND;
For i := 0 Step 1 Until size Do
      Begin "Go like a snail"
      slowBuf[i] := turtleBuf[i];
      slowBuf[i] := dodoBuf[i];
      End "Go like a snail";
Compiler!Switches /NOCHECK;  # -- disable dynamic arrays;
End "TEST!C!ARRAYS";
```

Generates C code:

```
/*Compiler!Switches: /nocheck*/   /* -- disable dynamic arrays*/
void static test_c_arrays(dodobuf,impalabuf)
      float dodobuf[];  float impalabuf[];
{/*TEST!C!ARRAYS*/
/*Compiler!Switches: /check*/   /* -- enable dynamic arrays*/
```

```
INTEGER /*ARRAY*/ *slowbuf, *turtlebuf;
INTEGER /*C!ARRAY*/ *fastbuf, *zipbuf;
INTEGER i, j, ch, size; float x;
slowbuf=arymkI(1,0,511,"slowbuf",'i');  /* Dynamic Array*/
turtlebuf=arymkI(1,0,511,"turtlebuf",'i');  /* Dynamic Arry*/
fastbuf=arymkI(1,0,511,"fastbuf",'i');  /* Dynamic Array*/
zipbuf=arymkI(1,0,511,"zipbuf",'i');  /* Dynamic Array*/
size=MIN(511,arrinfo(impalabuf,2));  /* -- get UPPER BND*/
for (i=0; i<=size; i++)
      {/*Go with the wind*/
      fastbuf[i] = zipbuf[i];
      fastbuf[i] = (*aChkF(impalabuf,1,i));
     .}/*Go with the wind*/
size=MIN(511,arrinfo(dodobuf,2));  /* -- get UPPER BND */
for (i=0; i<=size; i++)
      {/*Go like a snail*/
      *aChkI(slowbuf,1,i) = *aChkI(turtlebuf,1,i);
      *aChkI(slowbuf,1,i)) = dodobuf[i];
      }/*Go like a snail*/
/* compiler!Switches /nocheck*/   /* -- disable dynamic arrays*/
aryfree(slowbuf,turtlebuf,fastbuf,zipbuf,(INTEGER *)0);  /* Free Dynamic Arrays */
}/*TEST!C!ARRAYS*/
```

4. PORTABILITY CONSIDERATIONS

Because not all C compilers and their environments are equivalent, we must be careful to generate C code which uses those facilities available to force equivalent semantics. PSAIL treats all integers as 32–bit values using `typedef long INTEGER` or `typedef int INTEGER` to guarentee known precision. SAIL strings are mapped to C style strings using `typedef char *STRING` for compatibility with other C code packages. Byte-order problems (i.e. byte order in a 32–bit word) are handled in the PSAIL runtimes where required or automatically by target C compilers.

String garbage collection uses its own runtime active–string–pointer stack, stack–frame and temporary–string–stack with the <psrunG.c> runtime package. Dynamic arrays are allocated by PSAIL with an additional array header area located below the actual data area. As noted, this header is used by PSAIL runtimes for address calculations and optional bounds checking. Because array variables are actually pointers to the data part of the header, they are compatible with C–style array accessing for 1–D arrays starting at 0.

PSAIL has an extensive set of warning and fatal error messages. In addition to standard compiler bad–syntax type error messages, it also warns of many possible portability problems. This includes detecting DEC10 dependent code, non–portable SAIL constructs (eg. global 'Goto' from within a procedure, nested procedures, etc.) and non–uniqueness of variable names. PSAIL performs semantic checks wherever possible and detects integers and some operations greater than 32–bits which are flagged with descriptive messages and `/*WARNING*/` appended to the generated C code. For example both

```
('777777777776 Land j)
```

and

```
(j LSH 35)
```

would be non–portable. Non-portable SAIL syntax is translated but is flagged with `/*N.P.*/`. SAIL syntax which is not implemented in PSAIL is flagged with `/*N.I.*/`.

Mapping SAIL I/O to C/UNIX I/O is done with an emulation runtime package called <sairun.c> which uses whatever system calls are available in the target C environment. Particular target system dependencies are defined in files <config.h> and <confg2.h> and are used with `#ifdef` conditional code in

runtimes packages <sairun.c>, <psrunG.c> (String G.C.), <psrunX.c> (extended math), <psrunL.c> (LEAP), <psrunP.c> (processes, events and interrupts) and <psrunW.h> (portable profiler). Table 1 lists some of these <sairun.c> runtime functions. A portable single precision math library <psrun9.c> (portable math) functions may be used by specifying the */MATH* switch. The default */NOMATH* directs it to generate C code for C/UNIX double precision <math.h> libraries. The <config.h> and <sairun.h> are automatically `#include`'d in your PSAIL generated C program. Other options will force other header files to be included (eg. <psrunG.h> if */GCcode* is set).

5. SUMMARY

The SAIL language contains features such as macros, conditional compilation, dynamic arrays and strings as well as robust type checking and coercion for rapid application prototyping. It gives adequate protection against sloppy programming but also gives access to lower levels of abstraction when explicitly required for systems programming. By translating SAIL to C using a portable compiler such as PSAIL, these language facilities can become available for a wider class of machines. During translation, PSAIL takes the target C language and its runtime environment into account when trying to generate optimized code. When using a subsequent optimizing C compiler, even more benefits can be realized. Other C packages, such as a graphics package like the X-window System, could be used with PSAIL by simply including their header (i.e .h file) functions using SAIL macro and 'External' variable and procedure declarations prior to their use in SAIL. Because PSAIL encourages modular programming, the additional overhead of running PSAIL as a preprocessor on small modules is not that high especially when given the additional power of the language and increasing speed of today's workstations.

REFERENCES

1. Reiser, J.F. SAIL. Stanford Artif. Intell. Lab. memo AIM-289, or Computer Science Dept. Report #STAN-CS-76-574, (1976). [Also available as #AD-A045-102 from NTIS, Springfield, VA, 22161 (703)-487-4600, Microfiche $6.95, paper $19.95.]

2. Bobrow, D.B., Raphael, B., New Programming Languages for AI Research, *Computing Surveys* 6(3) 153-174 (1974).

3. Feldman, J.A., Rovner, P.D., An Algol-Based Associative Language, *CACM* 12(8) 439-449 (1969).

4. Kenig, M., LEAP - An Alternative AI Language, *Computer Language* 3(8) 30-33 (1986).

5. Lemkin, P., PSAIL: SAIL to C, *Computer Language* 2(8) 39-45 (1985).

6. Kernighan, B.W., Ritchie, D.M., *The C programming language*, Prentice Hall, Englewood Cliffs, NJ, 1978.

7. ANSI, X3-IPS C Information Bulletin - preliminary draft of ANSI C programming language standard, Doc# X3J11/85-045, April 30, 1985. X3 Secretariat, 311 First Street N.W., Suite 500, Washington, D.C. 20001.

8. Lemkin, P., Lipkin, L., GELLAB: A Computer System for 2D Gel Electrophoresis Analysis, *Computers in Biomed. Res.* Vol. 4, Part I. Segmentation: 272-297, II. Spot Pairing: 355-380, III. Data Base Search: 407-446 (1981).

9. Shapiro, M., Beginners Guide to SAIL, Division Computer Research & Technology, NIH, Bethesda, MD, 20892 (1978).

10. Smith, N., SAIL Tutorial. Stanford Artif. Intell. Lab. memo AIM-290, or Computer Science Dept. Report #STAN-CS-76-575. Also available from NTIS.

Table 1. Some of the PSAIL built–in runtime functions in <sairun.c>/<sairun.h>

Note: functions prefixed with '*' return a pointer. Arguments with '*' prefix are passed by 'Reference'. Some SAIL functions were name–mapped by added a 'PS' or 'P' prefix to avoid conflicts with C/UNIX libraries. The $ is a type suffix for some generic functions and takes on the values: I=Integer, F=real, C=String etc.

Storage Allocation Procedures.

*aAdr$(<args>) – return datum pointer given array actual–arguments
*aChk$(<args>) – is same as aAdr$(), also check actual args–limits.
arrblt(*dstPtr,*srcPtr,wordCnt) – Array block transfer.
arrclr(dstPtr,value) – clear dynamic array to value.
arrinfo(aryPtr,parameter) – return information on dynamic array.
arrtran(dstPtr,srcPtr,cnt) – transfer dynamic array data.
aryfre(<ptr-list>,0) – release list dynamically allocated arrays.
*aryHdr$(<args>) – create dynamic header in static array, return pointer.
*arymk$(<args>) – make dynamic array and return pointer.
Carrclr(*dstPtr,value,cnt) – clear non–dynamic array to value.
Carrtran(*dstPtr,*srcPtr,cnt) – transfer non–dynamic array data.
chkversion(<ptr-list>,0) – consistency check module version list.
del_record(ptr) – delete record previously allocated with new_record.
*new_record(nBytes) – allocate a record (struct) of size nBytes.
Phash(s,t,m) – lookup/enter string s in table t of size m.
Pfree(*ptr) – return dynamic block previously allocated with Pmalloc().
*Pmalloc(nbyte) – allocate nbyte block dynamic storage from heap and return pointer.
*salloc(nbyte) – allocate nbyte dynamic string from GC'able string space and return pointer.
*str_GC() – invoke PSAIL dynamic string garbage collector.

Program Control Procedures.

callsys() – emulate SAIL 'CALL's to DEC10 system where possible.

String manipulation Procedures.

*catlist(<nbr-args>,<ptr-list>) – return dynamic string of concatenation of list of strings.
*concat(s1,s2) – return dynamic string pointer of (s1&s2).
length(s) – return length of string s.
*subsr(s,a,b) – return dynamic string s[a For b] substring.
*subst(s,a,b) – return dynamic string s[a To b] substring.

String Conversion Functions.

copstr(s) – return INTEGER first char of string s. 0 if empty.
*cv6str(i)/*N.P.*/ – convert sixbit INTEGER to 5 or 6 character string.
cvasc(s)/*N.P.*/ – convert (4 or 5 char.) string to 7-bit INTEGER word.
cvastr(s)/*N.P.*/ – convert (4 or 5 char.) string to 7-bit INTEGER word.
cvd(s) – return decimal float value of string s.
*cvf(f) – convert float f to decimal string, return dynamic string.
*cve(f) – convert float f to E–fmt string, return dynamic string.
*cvg(f) – convert float f to G–fmt string, return dynamic string.
cvo(s) – return octal INTEGER value of string s.
*cvos(i) – convert octal INTEGER i to string, return dynamic string.
*cvs(i) – convert decimal INTEGER i to string, return dynamic string.
cvsix(s)/*N.P.*/ – convert 5 or 6 character string to INTEGER.
*cvstr(i)/*N.P.*/ – convert sixbit INTEGER i to string, return dynamic string.
*cvxstr(i)/*N.P.*/ – convert 7-bit ASCII INTEGER i to string, return dynamic string.
equ(s1,s2) – return TRUE if s1[1:inf]==s2[1:inf] else FALSE.
getformat(*w,*p) – get current conversion format field width and precision values.
lop(*s) – return INTEGER value of and advance string past 1st char.
*makstr(n) – make a 2 character string from LSB 7-bits INTEGER n and return dynamic string.
setformat(w,p) – set number→string conversion format field width and precision.

Scan Procedures

`breakset(tbl,brkchars,mode)` – modify scan break table.
`getbreak()` – get a free break table number 1 to 54 if any.
`intscan(*s,*bChr)` – scan INTEGER # from *s put break char in *bChr.
`realscan(*s,*bChr)` – scan real # from *s put break char in *bChr.
`relbreak(tbl)` – release break table.
`*scan(*s,tbl,*bChr)` – return scanned string *s w/brk tbl, set *bChr.
`*scanc(*s,brkStr,omitStr,modeStr)` – return scanned string *s without using break table.
`setbreak(tbl,brkStr,omitStr,mode)` – define `scan()` break table.
`stdbrk(tbl)` – set standard break tables (1–18) as in page 38 [1].

Special Numeric Functions

`ash(v,n)` – return INTEGER v arithmetic shifted n bits (–right,+left) in 32-bits.
`lsh(v,n)` – return INTEGER v logically shifted n bits (–right,+left) in 32-bits.
`rot(v,n)` – return INTEGER v rotated n bits (–right,+left) in 32-bits.

I/O Emulation Procedures.

`arryin(chn,*buffer,wordCnt)` – read INTEGER array from binary channel.
`arryout(chn,*buffer,wordCnt)` – write INTEGER array to binary channel.
`chncdb(chn)` – return pointer to this PSAIL channel data block.
`PSclose(chn,<opt. arg>)` – close both input and output channels.
`closin(chn)` – close input channel if any.
`closo(chn)` – close output channel if any.
`enter(chn,file,*err)` – enter an output file on channel.
`PSgetchan()` – return next free I/O channel number.
`getprint()` – return last Setprint() mode.
`*inchwl()` – wait for CR then return string input from terminal.
`inchrw()` – wait for and return next character from terminal.
`inchrs()` – return –1 if no character typed else return character.
`*input(chn,brkTbl)` – input string from ASCII channel using brktable.
`intin(chn,brkChar)` – input integer from ASCII input channel.
`lookup(chn,file,*err)` – lookup file on input channel.
`PSopen(chn,dev,IOmod,nIb,nOb,*cnt,*brkChar,*eof)` – open I/O channel.
`out(chn,str)` – output string to channel.
`outchr(intVal)` – output character to terminal.
`PSprintf(<ctrl-str>,<ptr-list>)` – machine independent printf() function.
`realin(chn,brkChar)` – input real number from ASCII input channel.
`relchan(chan)` – release previous Getchan() channel number.
`release(chn,<opt. arg>)` – release (close first) I/O channel.
`PSrename(chn,fileName,protection,flag)` – rename file entered on channel.
`senter(chn,file,protection,*err)` – enter an output file on channel and set protection.
`setprint(file,mode)` – set Print() output to terminal and/or output channel.
`ttyup(flag)` – set tty input case conversion and return old value.
`usererr(value,code,msg)` – user error message processor.
`useti(chn,blkNbr)` – set input channel random I/O block address.
`useto(chn,blkNbr)` – set output channel random I/O block address.
`wordin(chn)` – read next INTEGER word from binary input channel.
`wordout(chn,data)` – write INTEGER word to binary output channel.