PLA

Bill Schottstaedt

Center for Computer Research in Music and Acoustics

Stanford University

Stanford, California 94305

June, 1983

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Preface

Pla is a high level Algol-like language intended primarily for the composition of computer music. It is patterned after SAIL with some borrowings from LISP and SCORE. The program Pla interprets Pla code. This document purports to be both a reference manual for experienced users of Pla, and a tutorial for beginners. Some knowledge of programming is assumed. Please refer errors in this document or bugs in Pla to BIL.

To run Pla, type R PLA<cr>. (<cr> is the RETURN key). You are asked first for the input files. These can contain Pla code, note list data, or Seg function definitions. The file TTY: opens Pla's interactive mode. Type just <cr> to stop inputting files. The last thing you are asked for is the output file name.

The distribution file for Pla is PlaDis.Dis[P,Doc]. A library of possibly useful Pla code can be found in Plalib.Txt[Pla,Bil]. The program EdPla, described in EdPla.Bil[Mus,Doc], is an editor-oriented extension of Pla.

Introduction to Pla

To get sound from the Samson box, we define our synthesizer patches (instruments), load them into Sambox, and call them from a note list (1). Pla (like SCORE) is a program to aid in the production of this note list.

Say we want to call an instrument named "Simp" whose parameters are:

```
Begin_Time Duration Frequency Amplitude Amplitude_Function.
(P1        P2       P3        P4        P5)
```

We want to play a scale from middle c up an octave with a constant amplitude of .1 and an amplitude function named "AMP". Pla can accept all its input from the TTY, but it is usually more convenient to use a command file. So, we create the file "TEST" and put in it the following:

```
Header("Func Norm.Fun[1,Bil]");

Voice Simp;
Begin
P2←1;
P3←note: c4,d,e,f,g,a,b,c5,end;
p4←.1;
p5←"amp";
End;
```

Now we exit the editor and

```
.R PLA<cr>
File 1 → TEST
File 2 → <cr>
Output file (<cr>=TEST.PLA) → <cr>
```

and Pla creates a file called TEST.PLA containing:

```
FUNC NORM.FUN[1,BIL];
PLAY ;
SIMP, .000, 1.000, 261.620, .100,AMP;
SIMP, 1.000, 1.000, 293.660, .100,AMP;
SIMP, 2.000, 1.000, 329.630, .100,AMP;
SIMP, 3.000, 1.000, 349.230, .100,AMP;
SIMP, 4.000, 1.000, 391.990, .100,AMP;
SIMP, 5.000, 1.000, 440.000, .100,AMP;
SIMP, 6.000, 1.000, 493.890, .100,AMP;
SIMP, 7.000, 1.000, 523.240, .100,AMP;
FINISH;
```

which is what Sambox wants to see. To compile and play the file:

```
R SAMBOX;TEST.PLA<cr>
```

The Header statement (2) in the Pla code tells Pla to place the string in the header portion of the note list. You can have any number of header statements, each of which can contain as many statements for Sambox as you like.

---

(1) Pla output is also compatible with Mus10

(2) (see Page 66)

To make our little composition clearer to another reader (or to ourselves at a later date), we can change the file to be:

```
Header("Func Norm.Fun[1,Bill]");

Define Dur←P2,
       Freq←P3,
       Amplitude←P4,
       AmpFunc←P5;

Voice Simp;
Begin
Dur←1;
Freq←Note: c4,d,e | f,g,a | b,c5,end;
amplitude←.1;
ampfunc←"amp";
End;
```

Here we use macros to give the parameters names and use the "|" delimiter to mark bar lines in the scale. The same play file is also created by:

```
Header("Func Norm.Fun[1,Bill];
        Common Simp SimpA SimpB");

Define Dur←P2,Freq←P3,Amplitude←P4,AmpFunc←P5;

Voice Simp (0:7);
Begin
Dur←rhythm:q;
Freq←[c,d,e,f,g,a,b,c×2];
IF Dur<2 THEN amplitude←.1 ELSE Amplitude←.2;
ampfunc←"amp";
End;
```

The numbers in parentheses after the voice name are the first and last times at which a note from Simp is allowed to begin. Because we are not telling the voice to quit when it gets to the end of the list of notes, we need some other method of telling it when to stop.

Now we will make the amplitude get gradually louder over the course of the scale by using an envelope. In this example we define an envelope named RAMP in the Pla code. We could also define it in a function file (like NORM.FUN), then use a FUNC statement to load the file into Pla. We will also make the notes overlap each other by .1 second.

```
HEADER("Func Norm.fun[1,Bill]");

SEG Ramp 0 0 1 10;
    ∂ create RAMP going from 0 at 0 to 1 at 10;

PARS: Name, Beg, Dur, Freq, Amp, AmpFnc;
    ∂ PARS statement is like the DEFINE above, but easier to type;

VOICE Simp(0:7);
    BEGIN
    REAL Ur_Dur;     ∂ note duration before overlap calculated;
    Ur_Dur←RHYTHM: Q;
    Beg←Beg+Ur_Dur; ∂ begin time is independent of overlap;
    Dur←Ur_Dur+.1;  ∂ add in .1 second overlap;
    Freq←[c,d,e,f,g,a,b,c×2];
    Amp←Ramp[Beg];
                     ∂ amp increases according to RAMP[P1];
    AmpFnc←"Amp";
    END;
```

Finally, we decide to make the note's amplitude function be dependent on the note's overall amplitude, and decide to cause the frequency go up by quarter tones:

```
PROCEDURE QuarterTone(INTEGER Pitch,Octave);
RETURN(16.351*(2↑(Octave+(Pitch/24))));
            ∂ return the "Pitch-th" quarter tone in "Octave";
            ∂ based on c/16;
HEADER("Func Norm.fun[1,Bil]");

SEG Ramp  0 0 1 10;
    ∂ create RAMP going from 0 at 0 to 1 at 10;

PARS: Name, Beg, Dur, Freq, Amp, AmpFnc;
    ∂ PARS statement is like the DEFINE above, but easier to type;

VOICE Simp(0:7);
    BEGIN
    REAL Ur_Dur;      ∂ note duration before overlap calculated;
    INTEGER i;        ∂ frequency index;
    Ur_Dur←RHYTHM: Q;
    Beg←Beg+Ur_Dur;  ∂ begin time is independent of overlap;
    Dur←Ur_Dur+.1;   ∂ add in .1 second overlap;
    Freq←QuarterTone(i,4);
    i←i+1;            ∂ initialized to 0--i counts the note number;
    Amp←Ramp[Beg];
                      ∂ amp increases according to RAMP[P1];
    IF Amp>.5 THEN AmpFnc←"Amp" ELSE AmpFnc←"Pna";
    END;
```

Although you don't need to become a programmer to use Pla (you can just enter lists of data as in SCORE), you may discover that you want to specify musical events in terms of algorithms. These can range from the very simple examples given above to very complex composition programs. Numerous examples of such algorithms have been included in this manual.

Expressions

Pla's basic syntax is borrowed from the Algol language SAIL. Variables can be declared to be **Real**, **Integer**, **Listvar**, **Function**, **String**, **Array**, **Variable** (i.e. real), or **Context**. To declare a variable, precede its name with the type declarator. For example, to declare the integer variables NOTECOUNTER, I, and ALLOCATIONSTATS,

```
INTEGER NoteCounter, I, AllocationStats;
```

Note that a semicolon terminates each statement (as in SAIL). All characters of a variable name are significant. All lower case characters are mapped into the corresponding upper case character. Only those characters used as operators cannot be used in identifier names.

All simple data types know about the assignment and print operators. Expression syntax is rather free. P0 through P64 are predeclared, and any number of P numbered variables can be used (see PField Page 63). There is also an array called "P" consisting of the same variables (P[1] is the same as P1). Pfields have both a real and a string value at all times. Pla uses the expression context to determine which value is intended.

All the normal frequency names (C, CS for C-sharp, CF for c-flat) are predeclared as real variables with the standard equal tempered tunings. BS is an octave above middle-c, CF is a half-step below middle-c.

To help users of external terminals, Pla accepts GEQ as a replacement for ≥, LEQ for ≤, NEQ for ≠, and := for ←. Most ordinary type conversions are done automatically. Expression syntax is that of SAIL. Any number of parentheses can be used. Operator precedence is:

```
highest    ¬ NOT : ∩ ∪
           ↑
           * / MOD &
           + -
           MIN MAX LSH LOR LAND LXOR ROT LNOT
           ← = ≠ < > ≤ ≥
           AND
lowest      OR
```

When in doubt, use parentheses.

TypeIt and Expr_Type

TypeIt returns the type of an identifier:

TypeIt("*identifier*")

TypeIt can be used in conditional compilation situations similar to SAIL's Declaration. For example, if we want to find out whether the procedure READSAV has been declared before we try to use it:

```
IF TYPEIT("READSAV")=0        ∂ has "READSAV" been defined yet?;
  THEN                        ∂ no--read "COMP";
    EVAL("REQUIRE Comp");     ∂ this causes "COMP" to be loaded;
```

TypeIt is also needed if you want to write generic routines. The values returned by TypeIt are

defined in the file PlaTab.Def[Pla,Bil]. TypeIt returns 0 if the identifier given has not yet been declared.

Expr_Type returns the type of the expression passed as its argument, similar to SAIL's **Expr_Type**.

Examples of some type conversions:

```
STRING S;
LISTVAR L;
REAL X;
INTEGER I;

L←{1 2 3};
S←NTH(L,1);          ∂ S="1"     -- convert 1st element of L to a string;
S←NTH(L,1)&" ";      ∂ S="{1} " -- here a listvar prints as a list;
I←NTH(L,1);          ∂ I=1;
X←NTH(L,1);          ∂ X=1.000;
I←65;
L←{};
X←3.14;
S←X;                 ∂ S=" 3.140" -- SAIL's CVF;
S←I;                 ∂ S="65"     -- SAIL's CVS;
S←ASCII(I);          ∂ S="A"      -- 65 = A in ASCII ordering;
L←INSERT(L,S,1);     ∂ L={A};
L←{};
L←INSERT(L,X,1);     ∂ L={3.140};
S←I&" "&X&" "&L;     ∂ S="65 3.140 { 3.140 }";
```

Reals and Integers

Operators

octal integer	'	"'12" = 10 decimal
assignment	←	"RealVar←3.14" assigns the value 3.14 to RealVar. Some type conversions are done automatically. Int_Var ← String_Var assigns the ASCII value of the first character of String_Var to the variable Int_Var. Int_Var←Real_Var is equivalent to using the Floor function (Int) on Real_Var.
addition	+	"2+3" is 5
subtraction	-	"2-3" is -1
multiplication	*	"2*3" is 6
division	/	"2/3" is 0.6666667. If integer division is indicated by the context, Pla rounds toward negative infinity. "IntVar←2/3" assigns 0 to IntVar and "IntVar←(-2)/3" assigns -1 to IntVar.
exponentiation	↑	"2↑3" is 8
greater than	>	"2>3" is false (0)
less than	<	"2<3" is true (1 or any non-zero quantity)
greater than or equal to	≥	"2≥3" is false (Geq can be used)
less than or equal to	≤	"2≤3" is true (Leq can be used)
equal	=	"2=3" is false
not equal	≠	"2≠3" is true (Neq can be used)
Not	Not	"NOT 2" is 0 (false) (~ can be used)
And	And or ∧	
Inclusive Or	Or or ∨	
modulo	Mod	"2 MOD 3" is 2
minimum	Min	"2 MIN 3" is 2
maximum	Max	"2 MAX 3" is 3

Bit-wise operators

	Lsh	logical shift
	Lor	logical Or
	Lxor	logical exclusive Or
	Land	logical And
	Lnot	complementation
	Rot	rotation
Print	**Print**	
Print to file	**Cprint**	

Built-In Functions

logarithm base e	**Log**	"LOG(8)/LOG(2)" is 3
sine	**Sin**	"SIN(180)" is approximately 0
floor (truncate)	**Int**	"INT(3.7)" is 3 (round toward -infinity)
absolute value	**Abs**	"ABS(-3)" is 3
arbitrary characters	**Ascii**	"ASCII('12)" is carriage return
sharp	n	"nc" is c-sharp
flat	u	"uc" is c-flat

Random Numbers

Random number sequences are generated by **Ran** and **RanF**. **Ran** has an optional argument which, if specified, is used as the seed for all further random numbers. **Ran**'s argument is meaningful only the first time it is called. Ran returns numbers between 0 and 1. **RanF** takes one argument also, but in this case it is the amount of correlation between successive random numbers (0=white noise, 1=a single number, .5=1/f noise (sort of), and so on). The random numbers produced by **RanF** fall between 0 and 1.

Predeclared Constants

The following constants are predeclared:

True	(any non-zero quantity)
False	0

Pi or π	3.14159265
∞	some very big number

Strings

A String is a sequence of almost any number of characters. The absolute maximum string length changes from day to day, but is guaranteed to be greater than 5000.

Operators and Functions

quote a string	'	"hi" returns "hi" whereas "hi" returns hi.
concatenation	&	"hi"&" there" returns "hi there"
print		PRINT("Hi there") prints "Hi there"
Scan for a word	**Scan**	SCAN("hi there") returns "HI" leaving "THERE" in the scanned string.
Scan for reals	**RealScan**	REALSCAN("1.0 2.1 3.2") returns the real value 1.0 leaving ("2.1 3.2") in the scanned string.
remove first character	**Lop**	LOP("Hi there") returns "H" leaving "I THERE" in the scanned string.
Print to file	**CPrint**	see Cprint, Page 57
return octal representation	**Octal**(<integer>)	

Predeclared Constants

carriage return—linefeed	↓
null string	**Null**
comment	∂

LowerCase

LowerCase takes a string argument and does non-obvious things to it. At one time, in the far distant past, Pla did not respect case distinctions in string constants, and LowerCase got around that. Pla still sometimes puts everything in upper case if it is feeling worried, so LowerCase can come in handy.

```
PRINT(LOWERCASE("Hi there"))
```

prints: Hi There.

Arrays

An Array is declared:

Array *ArrayName* [*Lower bound*:*Upper Bound*];

An array can be of any size, and is assumed to be an array of real numbers unless you explicitly give the array type.

```
ARRAY xx[0:100];
VARIABLE i,j,k;
FOR i←1 STEP 1 UNTIL 50 DO
    xx[i]←i;
```

fills XX with 1,2,3...up to 50, then leaves the rest 0. All real array elements are initialized to 0, string arrays are initialized to Nulls, and listvar arrays are initialized to empty lists.

Integer, String, Listvar, Context, and FlavorVar arrays can be declared by including the type before the word ARRAY:

```
STRING ARRAY Names[1:30];
LISTVAR ARRAY Lists[1:2];
```

Local arrays of all types are supported. (3) Arrays can be passed as procedure arguments. No matter what the actual array type, the procedure argument type should be given as simply: "Array".

```
PROCEDURE hi(ARRAY names);  ∂ pass a STRING ARRAY to Procedure Hi;
PRINT(names[1]);
```

The array bounds can also be integer variables (*not* expressions). If several arrays have the same bounds, the bounds need to be stated only at the end of the list.

Preset

Preset("*array-name*",*listvar*)

Preset stores the values found in *listvar* into *array-name*. If *array-name* has more elements than *listvar*, the extra array elements are initialized to 0, null, or {}.

```
Preset("hi",{1 2 3 4})
```

puts 1, 2, 3, and 4 into the first 4 locations of HI. String and Listvar arrays can also be preset. The elements of the list passed to Preset must be constants.

---

(3) Function arrays are currently unimplemented in any form

Arrinfo

Arrinfo(*array-name,index*)

Arrinfo returns information about the array passed as its first argument. The indices are

```
0       Return total number of elements in array
1       Return lower bound of array
2       Return upper bound of array
```

Functions

Functions are much like arrays with the addition of an interpolation along the x-axis. They are defined like Seg functions:

```
SEG F1   0 0
         1 25
         1 75
         0 100;
```

defines a squared-off pyramid shaped function named F1. Remember that SEG expects (y,x) ordered pairs!

A function is accessed by giving its name followed by the x-axis look up value in "[]". An interpolation is performed if necessary. Using F1 defined above, F1[0]=0, F1[-3]=0 (below the lower limit and above the upper limit, the first or final value, respectively, is returned), F1[25]=1, F1[50]=1, F1[10]=.4.

To scale and offset a function's values, just do the appropriate arithmetic (F1[x]*10+25). To apply a function over time, use P1 as the look up variable, scaled appropriately.

```
F1[(P1 MOD 3)*33]
```

repeatedly moves through F1 in 3 second cycles.

Func

You can read in a file of function definitions with the Func statement:

```
FUNC Fun.Fun;
```

This statement loads into Pla all the Seg functions defined in Fun.Fun.

Function

To declare a Function variable:

```
FUNCTION NewRamp;
NewRamp←F1;
```

The assignment causes NewRamp to be exactly the same function as F1. Functions can be passed as procedure arguments, and can be local variables. A function variable tries to be smart about assignments—if you have a string or pfield variable which contains the name of a function, you can assign that function to a function variable with "←". Upon assignment, the function is not copied.

MakeFunc

MakeFunc ("*Func-name*", *breakpoint-list*)

Use **MakeFunc** to create a function without having to deal with strings. MakeFunc has two arguments—the name of the function to create and a listvar containing the breakpoints (4). If the function already exists, MakeFunc redefines it.

```
MAKEFUNC("F1",{0 0 1 25 1 75 0 100});
```

is equivalent to the Seg call given above.

FuncList

func-list ← **FuncList** (*function*)

FuncList returns a listvar containing the breakpoints of the function passed as its argument. Rather than duplicate all the operators and procedures given with lists, Pla gives you function-listvar and listvar-function conversion routines. The format of the listvar returned is {y1 x1 y2 x2 ... yn xn}.

(4) In the form {Y1 X1 Y2 X2 ...}

Lists

A list is a linked list of elements. The elements can be reals, integers, strings or lists. A list is defined by enclosing the desired elements in "{}" and separating the elements with a delimiter (space, comma, etc). {2 3 4} is a list containing three elements. The curly brackets are used to make Pla's life a little easier—once inside a list "()" can be used as well as "{}". A list variable is called a Listvar. It can be used as an argument to a procedure or as an array element (see Page 15).

```
AListVar←BListVar;
```

assigns to AListVar the very same list that BListVar points to (this is a feature!). To get a copy of BListVar's list, use *Quote* (see Page 20). If a list is assigned to a non-list-variable, the first element of that list is used with (hopefully) appropriate type conversions.

```
AListVar←{1 2 3};
AnInteger←AListVar;
```

assigns the integer value "1" to AnInteger. To remove the current value of a listvar,

```
AListVar←{};
```

"{}" is the empty list.

Length

Len ← **Length** (*list*)

Length returns the number of elements in the given list. Because an element of a list can be another list,

```
LENGTH({1  2  (3 4)  5})=4
```

Insert

rtnlist ← **Insert** (*baseList*,
addedList, *location*)

Insert inserts *addedList* into *baseList* at the *location*'th element of *baseList*. If *addedList* is an expression (including a real or string constant), Insert evaluates the expression and places the result in the list at the indicated location. *BaseList* can be empty.

Delete

rtnlist ← **Delete** (**Reference** *baseList*, *location*)

Delete deletes the *location*'th element of *baseList*. Note that *baseList* is changed by Delete.

Reverse

rtnList ← **Reverse** (*baseList*)

Reverse reverses the elements (at the top level) in *baseList*.

Nth

rtnList ← **Nth** (*baseList*, *location*)

Nth returns the element or list found at the *location*'th entry in *baseList*. **Nth** can occur on the left side of an assignment operation:

```
NTH(LstV,3)←4
```

assigns the value 4 to the third element of LstV (which must exist).

Nth can also be used with a string argument:

```
NTH("HI",2)="I"
```

Quote

rtnList←**Quote**(*baseList*) returns a copy of *baseList*.

NullP, AtomP

NullP(*baseList*) returns true if *baseList* is a null list. *Baselist* can actually be of any type—for flavorvars, contexts, and functions **NullP** returns whether the variable has any instance, stored variables, or break points, respectively. If you Forget("All",context), that context will return true if passed to **NullP** until something is stored into it. All these variables are initialized to a null state.

AtomP(*list-element*) returns true if *list-element* is not itself a list.

Listify

rtnList ← **Listify** (*baseList*)

Listify returns *baseList* as a list of a list. **Listify** is needed because of **Insert**'s unfortunate (5) desire to flatten list representations. If you want to put a list in a list and have it remain a list, use **Listify** (sigh).

---

(5) Buggy

Append

rtnList ← **Append** (*list1*, *list2*)

Append returns a list made up of a copy of *List2* appended to a copy of *List1*.

Next

rtnList ← **Next** (*baseList*)

Next points *rtnList* at the element after the one that *baseList* is pointing at. The list is not copied, so you get two pointers into exactly the same list! Next is provided as an optimization.

Eval

val ← **Eval** (*list-or-string*)

Eval evaluates its argument as though it were an expression-- Eval({2 + 3}) returns 5, as does Eval("3+2").

You can use Eval to deal with a list of expressions by enclosing the expressions in quotation marks (this is often easier than building the expression into a list structure and evaluating that). For instance, you can fill the ListVar ListV with the following:

```
ListV←{"2+3" "i*2"};

NTH(ListV,1);                  ∂ is the quoted string "2+3";

EVAL(NTH(ListV,1));            ∂ is the unquoted string 2+3;

EVAL(EVAL(NTH(ListV,1)));      ∂ is the real or string value 5.000;
```

Evaluation of a quoted string returns the string within the quotes. Evaluation of a string returns the value of that string treated as an expression. If it is a variable name, for example, you get back the value of the variable.

Other List Functions

Using the functions given above it is possible to define APPLY, MAPCAR, and other useful LISP functions. Suggested definitions are given below.

```
PROCEDURE MapCar(STRING Fnc; LISTVAR Lst);
BEGIN                                   ∂ LST is changed;
INTEGER i;
FOR i←1 STEP 1 UNTIL LENGTH(Lst) DO
  NTH(Lst,i)←EVAL(Fnc&NTH(Lst,i));
RETURN(Lst);
END;

PROCEDURE Apply(STRING Fnc; LISTVAR Lst);
RETURN(EVAL(Fnc&Lst));

PROCEDURE Car(LISTVAR Lst);
RETURN(NTH(QUOTE(Lst),1));

PROCEDURE Cons(LISTVAR Lst1,Lst2);    ∂ or use APPEND;
RETURN(INSERT(QUOTE(Lst1),QUOTE(Lst2),LENGTH(Lst1)+1));

PROCEDURE Subst(LISTVAR NewExpr,OldExpr,BaseList);
BEGIN                                   ∂ BASELIST is changed;
INTEGER I;
FOR I←1 STEP 1 UNTIL LENGTH(BaseList) DO
  IF NTH(BaseList,i)=OldExpr
    THEN NTH(BaseList,i)←NewExpr;
RETURN(BaseList);
END;
```

If a macro name occurs within a list constant, it is expanded, but not evaluated. If a list constant element is in "[]", it is left exactly as is (this allows you to put anything into a list).

Cyclic Lists

Cyclic lists (or streams) are the default data structure throughout SCORE. They are defined in Pla in several ways (mostly to remain somewhat compatible with SCORE):

1) The stream can be placed in square brackets. In this case the elements of the stream can be expressions, and the elements of the stream should be separated by commas.

2) Streams can also be defined using the words List, Note, or Rhythm. In this case, most of SCORE's features work (motives, dotted rhythms, etc). The expected delimiter for this kind of stream used to be "/", but slash is now discouraged (and will eventually be obsolete). Use a comma or vertical bar ("|") instead.

The type of the stream is the type of the first element of that stream. A stream has an internal pointer which is pushed forward every time the stream is accessed. When the end of the stream is reached, the pointer is sent back to the start of the stream, so a stream cycles continually through its elements. Because the pointer is not accessible, the cycling cannot be turned off—use an actual list (Listvar) or an array for arbitrary list manipulations. A stream can occur anywhere in an expression that a variable of the same type is legal. For example, in

```
P4←[1,2,3]+2;
```

P4 is assigned successively the values 3, 4, and 5. Stream elements can also be strings: ["f1","hi"].

```
freq←[2,4,3]×[1,3,2,4,6];
```

cycles through the two streams with different periods, so *freq* is assigned the values 2, 12, 6, 8, 24, 3, 6, and so on.

To get a list of quoted strings (to include the quotation marks in the value of the expression), put a " ' " inside the open quote mark. *StrVar*←["amp","ind"] assigns *StrVar* the values AMP and IND, whereas *StrVar*←["'amp","'ind"] assigns it the values "AMP" and "IND".

List, Rhythm, Note

List, Note, and Rhythm are taken from SCORE and have very similar uses. The lists defined by these keywords can occur anywhere in an expression. The expected delimiter is "/", although "|" or comma are also recognized. The elements of the list following List, Note, or Rhythm cannot be arbitrary expressions.

List introduces a list of strings. The following SCORE constructs can occur in that list:

"X" after an element in the list, followed by a number, causes that element to be repeated that number of times:

```
P3←LIST: f2,f3×2,f1;
```

produces the list f2,f3,f3,f1.

A null element (an immediate repetition of a delimiting character) causes the previous element to be repeated:

```
P3←LIST: f3,,f2;
```

produces the list f3,f3,f2.

A Name (like "Z" or "List_Id") followed by a "(" denotes the definition of a motive. The motive is terminated with ")". It can be called later with ⍟Name.

```
P3←LIST|F3,Motive_Name(F2,G,A),F4,⍟Motive_Name;
```

produces F3 F2 G A F4 F2 G A.

"END" as an element of the list causes the voice using the list to be turned off (Killed, see Page 48) when that point is reached in the List. If more than one list has an END, the shortest list determines when the voice quits. MyNoteNum (see Page 45) overrides all ENDs.

Rhythm introduces a list of rhythm names that are turned into the appropriate durations.

S=16th note, E=8th, Q=quarter, H=half, W=whole, T before any of these means triplet. Q becomes 1, and the others are scaled accordingly. G=grace note. Any number is replaced with 4/number. This means that 8 = eighth note, 1 = whole note, 32 = thirty-second note, 20 = quintuplet sixteenth, and so on. A list can contain both numbers and letters.

⍟motive_name *real_number* causes all rhythms in motive_name to be multiplied by *real_number*.

A dot (period) after a rhythm causes that rhythm to be multiplied by 1.5 (a dotted rhythm). Any number of dots can be applied to a rhythm. Because the number given in rhythm can be a real number, the dots can become confusing. "Rhythm: .5, .5., 8.0, 8., 8.0." creates a double whole note, a dotted double whole note, an eighth note, then two dotted eighth notes.

Ties are handled using "+" and "-" in simple expressions. An eighth note tied to a dotted sixteenth is: "e + s.". Any number of durations can be tied together. As further examples, "Rhythm: h.. + te." creates a duration equalling a triply dotted half note tied to a triplet dotted eighth (the latter = e of course), and "Rhythm: 5..-g." gives a doubly dotted quintuplet quarter minus a dotted grace note.

Note introduces a list of note names using the convention that C4=middle C (octaves go from C to c so A4=440 Hz.) This is the only place in Pla where C4 is middle-c! In all normal expressions C is middle-c.

F=flat, S=sharp. EF3=e-flat a major sixth below middle c. N=natural (purely for SCORE compatibility). R=Rest (the note is UnQueued, see Page 48).

⍟motive_name *real_number* causes all notes in motive_name to be transposed *real_number* half steps up (or, if negative, down).

Motives can be defined within Motives. Any motive, once defined, is available globally from then on—its name should therefore be unique.

An example of the use of motives:

```
VOICE Simp (0:20);
  BEGIN
  I_ONLY:Mynotenum←19;
  p2←1;
  p3←LIST: f2, f3,, f4,
           Motive (gl1,gl2,gl3), f1, eMotive, f1;
  p4←NOTE: Frq (af4,af3),, b2 ,eFrq X 2, cs3,;
  p5←RHYTHM:e,Qrt (e,e,Trt (q,q),u),e,eTrt,eQrt;
  END;
```

The doubled delimiter after the definition of "Frq" repeats only the last element of "Frq" whereas "eFrq,," would have repeated all of "Frq". (I think that's the way SCORE works (?). "CS/2" is repeated because the line ended with ",". Clear as mud?

In Note, chords can be specified by separating the elements with ":". Thus Note: c:e:g,g:b:d5; alternates between the two three note chords. The chord feature works best if you are not setting P1 (the begin time) yourself. If you must set p1, you can find out when a chord is being played by looking at GetInfo(19) (See Page 65).

List declarations can occur within expressions:

```
VOICE Simp;
BEGIN
if p1 mod 2=0 then p2←rhythm:e,e,q;
p1←p1+1;
p3←(note: a4,b)*3;
END;
```

In this example, P2 is assigned the constantly recurring cycle: .5, .5, .5, .5, 1.0, 1.0. The internal pointer in the list is pushed forward only when the list is referenced which happens here only on every other note. P2 keeps its former value on the other notes. You can use cyclic lists in conjunction with the Case statement to step selectively through a group of motives:

```
CASE x of
  begin
  [0] p3←Note|a4,d,a;
  [1] p3←Note|bf4,d,bf;
  [2] p3←Note|g4,d,g;
  [3] p3←Note|bf4,g,bf;
  [4] p3←Note|bf4,,;
  else p3←Note|a4,e,;
  end;
x←(x+1) mod 5;
```

Rests

There are two simple ways to get rests: either use the "R" notation in Note, or UnQueue the voice (see Page 48). To imitate SCORE which thinks that a negative p2 value denotes a rest:

```
VOICE Simp(0:10);
    Begin
    p2←[-1,2,-1,2];
    if p2<0 then
        begin
        p2←abs(p2);
        unqueue(Self);
        end;
    End;
```

Score syntax is full of ambiguities, which Pla tries to sort out in some reasonable manner. Consider the following legal input:

```
List:X2X2(2X2),X2(2XX2),•X2X2;
```

In general, if you cannot tell what it means, neither can Pla.

If you would like to specify the note and rhythm values together but still use motives and whatnot, see the page "Note and Rhythm Combined" in PlaLib.Txt[Pla,Bil].

Contexts

A context is an open ended storage area in which the names and values of variables can be saved. SAIL uses the context mechanism for "backtracking". A context is a Pla data type which is declared:

Context *name*, . . . ;

just as integers or other variables are declared. Contexts can be returned from procedure and passed to procedures as arguments. Type Help *context-var* to print out the contents of a context. Once a context has been declared, any number of Real, Integer, String, Context, Listvar, or Array variables can be stored in it. The context remembers the name, type, and value(s) of the variable.

Store

Store ([*Variable-names*] , *context-variable*)

Store stores all the variables found in the set of *variable-names* into *context-variable*. If a given variable (name) already exists in the context, its value is updated. The square brackets are obligatory.

Restore

Restore ("ALL"-or-[*Variable-names*] , *context-variable*)

Restore assigns to the variables named in *variable-names* the values stored in the context *context-variable*. If the string "ALL" is found, every variable found in the context is restored.

Value

Val ← **Value** (*variable-name*, *context-variable*)

Value returns the present value of *variable-name* in the context *context-variable*. The value of the variable itself is not changed.

Forget

Forget ("ALL"-or-*variable-name*, *context-variable*)

Forget clears out all the contents of *context-variable* if the first argument is "ALL". Otherwise it deletes *variable-name* from the context.

In_Context

In_Context (*variable*, *context*)

In_Context returns true if *variable* is found in *context*.

For example:

```
Listvar L;
Array Hi[0:10];
Context C1;
l←{1 2 3};
Hi[3]←3;
Store([L,hi],C1);          ∂ put present state of L and Hi in C1;
l←{};                      ∂ now clobber the variables;
Hi[3]←2;
Print(Value(L,C1));        ∂ will print (1 2 3);
If L In_Context C1         ∂ is the listvar L in the context C1?;
  Then
    Restore([L],C1);       ∂ now L = {1 2 3} again;
Restore("All",C1);         ∂ nw HI[3] is 3 as well;
Forget("All",C1);          ∂ now C1 is an empty context;
```

There are several slight differences between Pla contexts and SAIL contexts: Forget can clear the entire context in Pla; Local variables can be Stored in a context from a nested scope and still be legally accessed outside that scope; only an entire array (not a subrange) can be restored in Pla; a Pla procedure can return a context (SAIL procedures must use Context ItemVars). When a context is assigned to another context, the first is copied. You end up with two distinct contexts, not two pointers to the same context.

Statements

Any amount of "white space" (spaces, tabs, crlfs, formfeeds) can be put between any portions of a statement—Pla accepts very free format input. Because Pla is an interpreter, there are few "musts" about statement order. Statements and declarations are executed immediately unless they are in a special block (see Header, Always, Finish, Voice, Mute, Procedure, Define).

Comment, Dont_Scan

Comments are preceded by "∂" or "COMMENT" and end with a semicolon. To comment out large sections of code, use Dont_Scan. Dont_Scan causes Pla to ignore everything until it sees the statement Scan;.

If_Then_[Else]

IF *boolean_expression* Then *statement* executes the statement after Then only if *boolean_expression* is true. Anything non-zero is true. The If statement can also appear within an expression:

```
X←(IF Y=3 THEN Y ELSE 2)
```

assigns the value of Y to X if Y is 3, and assigns 2 to X if Y is not 3. If the Else clause exists and the If clause is false, the Else statement is evaluated.

Case

Case *selector* Of BEGIN [*index1*] *statement*_ELSE *statement*; END;

The Case statement chooses a statement from a list of indexed statements and executes only that one. If there is an Else clause (which can occur anywhere in the list), it is executed if the case selector value does not match any of the indices. Pla's Case statement follows SAIL, not Pascal syntax, except that the last branch of the Case statement must end with a semicolon (like every other branch). The Else branch is optional and can occur at any point among the branches. A branch can have more than one index.

```
CASE i OF
  BEGIN
  [1]     PRINT("i must be 1");
  [2] [3] PRINT("i may be either 2 or 3");
  ELSE    PRINT("i don't know what i is");
  END;
```

While..Do

While *boolean_expression* **Do** *statement* executes *statement* while the *boolean_expression* is true. If *boolean_expression* is false from the start, *statement* is not executed.

For..Step..Until..Do

Another form of loop control structure is the **For** statement. **For** *ctr* ← *initial-value* **Step** *step-value* **Until** *final-value* **Do** *statement*. An example:

```
FOR i←1 STEP 3 UNTIL 12 DO PRINT("I is "&i);
```

Initial-value, *step-value*, and *final-value* can be expressions, but only *final-value* is evaluated on every iteration of the loop. *Ctr* can be a subscripted variable.

Done, Continue

At any point in a loop, you can jump out of the loop with the **Done** statement or jump back to the start of the loop with the **Continue** statement. These two statements apply only to the innermost loop in which they occur. Block names are ignored.

```
WHILE TRUE DO                    ∂ "forever do";
  BEGIN
  I←I+1;
  IF I>10 THEN DONE;             ∂ exit the loop;
  IF I MOD 2 = 0 THEN CONTINUE;  ∂ jump to start of loop;
  PRINT(i&" is odd!");
  END;
```

Begin..End

The Begin..End statement groups many statements into a single statement with optional local variables. **Begin** *local variable declarations statements* **End** can appear wherever *statement* appears above. A Begin..End block can have any number of local variables which are known only within that block.

```
IF I>3 AND J<2 THEN
  BEGIN
  INTEGER LocalCtr,K;
  LISTVAR Lst;
  LocalCtr←i+j;
  ∂ <and so on>      ;
  END;
```

Local variables are initialized at the start of the program to 0, NULL, {}, but thereafter are not reinitialized upon block entry. The BEGIN and END statements can be followed by an arbitrary block name (a quoted string), but all block names are ignored.

Procedures

A procedure is a named statment that is executed when its name is used. A procedure can have any number of arguments, and can return a value. As in SAIL, the argument types must be given. They can be Integer, Real, String, Listvar, Function, Array, or Context. The value returned can be of Real, String, Integer, Listvar, or Context type. Listvars, Arrays, Functions, and Contexts are passed by reference; the others are passed by value. To pass a procedure as a formal argument to a procedure, or to pass anything by reference, pass the name of that thing, then call Eval (see Page 21).

Procedures are defined with the word "Procedure" followed by the procedure name and arguments (if any) in parentheses, a semicolon, then the statement which is to be executed when the procedure is called. If this statement is a Begin..End block (the normal case), local variables are declared just after the "Begin". A procedure is exited either when it completes execution of its block or when it executes a Return statement. The word "Return" can be followed by an optional argument in parentheses. This argument is the value returned by the procedure. A Pla procedure can contain any number of Return statements which need not return values of the same type.

```
PROCEDURE SayArg(STRING Arg);
PRINT(Arg);
```

is a minimal procedure that simply prints its argument.

```
PROCEDURE RtnArg(INTEGER Arg);
BEGIN
REAL TwoPi;
TwoPi←3.14*2;
RETURN(Arg+TwoPi);
END;
```

returns its argument plus pi times two (approximately). "RtnArg" can appear anywhere in an expression that the type of the value returned is legal (for example, RealVar←RtnArg(3.0) assigns 9.28 to RealVar).

If a voice is created within a procedure, it should copy any procedure parameter values it wants saved into its own local variables at LONLY time.

Recursion

Recursion is supported. Local variables are not placed on the stack.

```
PROCEDURE SetTime(INTEGER SNo,TNo; REAL BaseNo,Len);
BEGIN
INTEGER BigNo;

comment return the length of the SNo-th piece of a section having Tno parts, where Len is
      the overall length and BaseNo determines the relative lengths of the parts;
```

```
IF Tno=1
  THEN RETURN(Len);
BigNo←INT(TNo*BaseNo+.5);
IF Sno>BigNo
  THEN
    RETURN(SetTime(Sno-BigNo,TNo-BigNo,BaseNo,Len*(1-BaseNo)));
RETURN(SetTime(Sno,BigNo,BaseNo,BaseNo*Len));
END;
```

comment An example: If we have a piece with 5 sections and want the section timings to reflect a ratio of .618, we call this procedure for each individual section with

```
SETTIME(Section_Number,5,.618,Total_Length_Of_Piece)
;
```

Examples

This section contains a variety of compsitionally useful procedures. The hope is that these examples will be valuable from a pedagogical as well as a practical viewpoint.

Factorial

Every computer-oriented text seems to have a definition of factorial lurking about. Far be it from me to flaunt tradition:

```
PROCEDURE !(INTEGER n);
IF n≤1
  THEN RETURN(1)
ELSE RETURN(n*(!(n-1)));
```

Random Choice from a List

One of the first things computer composers often want is a way to choose randomly from elements of a list. In the simplest case, we define our list of possible elements:

```
LISTVAR RList;
RList←{C E F G B};
```

then choose elements of that list using RAN (Page 10):

```
REAL Freq;
Freq←NTH(RList,1+RAN*LENGTH(RList));
```

Pitch Modes

We can build a similar system using the bit-wise operators (Page 10) to take an integer between 0 and 11 and return the nearest integer which fits in the current mode using normal musical definitions of mode such as Ionian.

```
PROCEDURE SetBit(INTEGER i,j);
RETURN(i LOR (1 LSH j));
                    ∂ set the j-th bit of i to 1;

PROCEDURE Bit(INTEGER i,j);
RETURN(i LAND (1 LSH j));
                    ∂ return the value of the j-th bit of i;

PROCEDURE SetFreqBits(LISTVAR L);
BEGIN               ∂ bit is 1 if note is not allowed;
INTEGER i,FreqInt;
FreqInt←0;
FOR i←1 STEP 1 UNTIL LENGTH(L) DO
  FreqInt←SetBit(FreqInt,NTH(L,i));
RETURN(FreqInt);
END;


INTEGER Aeolian Chromatic Dorian Phrygian Lydian Mixolydian Ionian
        Locrian Minor Octatonic Nonatonic Pentatonic WholeTone;

Aeolian←        SetFreqBits({1 4 6 9 11});
Dorian←         SetFreqBits({1 4 6 8 11});
Phrygian←       SetFreqBits({2 4 6 9 11});
Lydian←         SetFreqBits({1 3 5 8 10});
Mixolydian←     SetFreqBits({1 3 6 8 11});
Ionian←         SetFreqBits({1 3 6 8 10});
Locrian←        SetFreqBits({2 4 7 9 11});
Chromatic←      0;
Minor←          SetFreqBits({1 4 6 9 10});
Octatonic←      SetFreqBits({1 4 7 10});
Nonatonic←      SetFreqBits({1 5 9});
Pentatonic←     SetFreqBits({1 3 4 6 8 10 11});
WholeTone←      '52S2;
            ∂ define which pitches are not in the mode;


PROCEDURE SetMode(INTEGER FreqInt,Mode);
IF Bit(Mode,FreqInt)
  THEN      ∂ we assume that 0 is always in the mode;
    RETURN(SetMode(FreqInt-1,Mode))
ELSE RETURN(FreqInt);
            ∂ find nearest note to "FreqInt" in "Mode";
PROCEDURE SetFreq(INTEGER Pitch,Octave,Divisions);
RETURN(16.351*2↑(Octave+Pitch/Divisions));
            ∂ return "Pitch" in "Octave" divided into
              "Divisions" equal parts;
```

Now we can generate a stream of pitches and force them into whatever mode we like:

```
Freq←SetFreq(SetMode(Raw_Pitch,Our_Mode),Raw_Octave,12);
```

Filtered Random Selection

We often want to apply an envelope to a stream of random numbers. In Pla, we need only define the envelope we want, then:

```
SEG Amp_Env   0 0 1 100;
P4←RAN*Amp_Env[(TotalDur-P1)*100/TotalDur];
```

which gives us a sequence of random numbers whose amplitude gradually increases from 0 to 1 over the course of TotalDur. To get 1/f noise filtered in the same way:

```
P4←RANF(.5)*...
```

Almost as simple is the use of an envelope as a constraint. We define a lower and an upper bounds for some parameter using two functions, then force the value assumed by that parameter to fall inside the two envelopes:

```
SEG LowerOct   0 0 4 85 4 100;      ∂ low bound on numbers;
SEG HigherOct   8 0 4 95 4 100;     ∂ high bound on numbers;

PROCEDURE CheckEnv(REAL Val;
                   FUNCTION HighEnv,LowEnv;
                   REAL LookUp);
RETURN(LowEnv[LookUp] MAX Val MIN HighEnv[LookUp]);
P4←CheckEnv(Unchecked_P4,HigherOct,LowerOct,P1);
```

Weighted Selection

It is sometimes useful to produce only a certain group of numbers with some given weighting of probabilities. If we define the numbers we want and their associated weights as a list, we can build a function which, when looked up with RAN, will return the numbers desired. Say we want to have twice as many 3's as 1's and 10 times as many 3's as 5's, but only the values 1, 3, and 5 returned. Our list is then:

```
RList←{1 .5  3 1  5 .1}
∂ {number weight ...};
```

We then pass this list as an argument to the following Pla procedure:

```
INTEGER Keep_Old_Funcs; ∂ do we save intermediate functions;
Keep_Old_Funcs←FALSE;   ∂ this can be reset, of course;

PROCEDURE Weights(LISTVAR L);
BEGIN                   ∂ build function to produce weighted list;
STRING FName;           ∂ function name (generated internally);
REAL Sum,LastX,ThisX;
INTEGER i,FCtr;
LISTVAR FVals;          ∂ list built to describe new function;
Sum←0;
ThisX←0;
IF Keep_Old_Funcs THEN FCtr←FCtr+1;
FName←"FNC_"&Fctr;      ∂ generate new function name;
FOR i←1 STEP 1 UNTIL LENGTH(L)/2 DO
  Sum←Sum+NTH(L,i×2);   ∂ get total of all weights;
FVals←QUOTE({0});       ∂ QUOTE used to force FVALS to be {0};
FVals←INSERT(FVals,NTH(L,1),1);
                        ∂ now build list describing the function;
FOR i←1 STEP 1 UNTIL LENGTH(L)/2-1 DO
  BEGIN
  FVals←INSERT(FVals,NTH(L,i×2-1),i×4-1);
  LastX←ThisX;
  ThisX←LastX+NTH(L,(i×2))×100/Sum;
  FVals←INSERT(FVals,ThisX,i×4);
  FVals←INSERT(FVals,NTH(L,i×2+1),i×4+1);
  FVals←INSERT(FVals,ThisX,i×4+2);
  END;
FVals←INSERT(FVals,NTH(L,LENGTH(L)-1),LENGTH(FVals)+1);
FVals←INSERT(FVals,100,LENGTH(FVals)+1);
MAKEFUNC(FName,FVals);     ∂ build the new function;
RETURN(FName);
END;
```

Now we declare a function:

```
FUNCTION RFunc;
RFunc←Weights(RList);
```

To get our weighted random selection based on RLIST, we just:

```
P4←RFunc[RAN×100];
```

Now say we want the weight given a certain number in our list to be dependent on time. This allows us to gradually change from one mode to another, for example. In the implementation given here, we define a function that describes the behaviour over time of each of the values that appears in the list:

```
SEG Ramp_Up   0 0 1 100;
SEG Ramp_Down 1 0 0 100;
SEG Up_And_Down   0 0 1 50 0 100;

PROCEDURE RtnFncVal(STRING FName; REAL Val);
RETURN(EVAL(FName&"["&Val&"]"));

PROCEDURE ChangingWeights(REAL Time; LISTVAR L);
BEGIN        ∂ 0 ≤ time ≤ 100 -- describes how far along
             ∂ the X axis of the weighting functions we are;
LISTVAR L1;
INTEGER i;
L1←QUOTE({}); ∂ force it to be initialized to nil;
FOR i←1 STEP 1 UNTIL LENGTH(L)/2 DO
  BEGIN
  L1←INSERT(L1,NTH(L,i*2-1),LENGTH(L1)+1);
  L1←INSERT(L1,RtnFncVal(NTH(L,i*2),Time),LENGTH(L1)+1);
  END;
RETURN(L1);
END;
```

We define the changing weights in a list:

```
RList←{1 Ramp_Up  3 Ramp_Down  5 Up_And_Down}
```

On each note:

```
RFunc←Weights(ChangingWeights(P1,RList));
```

and over the course of time, the relative weighting of the values 1, 3, and 5 changes according to the functions given.

Weighted Interpolated Selection

Let us carry the investigation of weighted random selection one step further. We will now choose values from a continuum, not from a list of discrete quantities that the preceding procedures dealt with. In the method given here, we build and modify envelopes which represent the relative probability that a given value will fall at a given place. The Y axis is the probability that the X axis value will occur. Unlike WEIGHTS, here we interpolate within the value range, returning a value somewhere within the limits given, rather than exactly the one value given. It should be remembered that functions are specified as Y,X pairs.

```
{0 0  0 10  1 15  0 25  0 75  1 80  0 100}
```

as a probability function will return a function which, when looked up over the entire X axis (0:100) will return numbers between 10 to 25, and 75 to 100 with about 40% more likelihood of getting a number from the higher range (since the area under that portion of the curve is about that much larger). You can think of the generating function as an X,Y pairing where X is the probability that the value Y will occur (considering the entire function of course).

```
PROCEDURE ProbFunc(LISTVAR Probs);
BEGIN
comment Probs is the body of a function as a listvar;
STRING ProbF;              ∂ name of function created;
LISTVAR Vals,InnerVals;  ∂ new function's breakpoints;
INTEGER ProbFInt;
REAL Sum,PresentX;         ∂ area under envelope;
INTEGER Ctr,Len;
REAL XAmt,lastX;
IF Keep_Old_Funcs THEN ProbFInt←ProbFInt+1;
ProbF←"ENV_"&ProbFInt;
Vals←QUOTE({});            ∂ initialize to Nil;
InnerVals←QUOTE({});
Sum←0;
PresentX←0;
Len←LENGTH(Probs);
IF Len<3 THEN PRINT("Invalid argument to Probfunc");
FOR Ctr←3 STEP 2 UNTIL Len DO
  BEGIN                    ∂ make list of each area, get total area;
  REAL Area,X1,Y1,X2,Y2;
  X1←NTH(Probs,Ctr-1);
  X2←NTH(Probs,Ctr+1);
  Y1←NTH(Probs,Ctr-2);
  Y2←NTH(Probs,Ctr);
  Area←(X2-X1)*(Y1+((Y2-Y1)/2));
  Sum←Sum+Area;
  InnerVals←INSERT(InnerVals,Area,LENGTH(InnerVals)+1);
  END;
XAmt←0;


FOR Ctr←1 STEP 1 UNTIL (Len/2)-1 DO
  BEGIN
  LastX←XAmt;
  XAmt←NTH(InnerVals,Ctr)*100/Sum;
  IF XAmt>0
    THEN
      BEGIN
      INTEGER ILen;
      ILen←LENGTH(Vals);
      Vals←INSERT(Vals,NTH(Probs,Ctr*2),ILen+1); ∂ X0=>Y1;
      Vals←INSERT(Vals,PresentX,ILen+2);
      PresentX←PresentX+XAmt;
      Vals←INSERT(Vals,NTH(Probs,Ctr*2+2),ILen+3);
      Vals←INSERT(Vals,PresentX,ILen+4);
      END;
  END;
MAKEFUNC(ProbF,Vals);      ∂ create function and put on id-list;
RETURN(ProbF);             ∂ return name so we can refer to it later;
END;
```

We can now create and use the function returned by ProbFunc

```
FUNCTION Probs;
Probs←ProbFunc({0 0  0 10  1 15  0 25  0 75  1 80  0 100});
VOICE Simp;
    BEGIN
    P2←Probs[RAN*100];
    END;
```

As a final flourish, why not interpolate between two such systems?

```
PROCEDURE IntrpEnvs(FUNCTION IntrpF,Probs1,Probs2; REAL CurLoc);
Comment CURLOC is the current location in the interpolation (0-1).
        PROBS1 and PROBS2 are from PROBFUNC.
;
IF RAN>IntrpF[CurLoc*100]
  THEN                          ∂ IntrpF=0 => all from Probs1;
    RETURN(Probs1[RAN*100])
ELSE RETURN(Probs2[RAN*100]);   ∂ IntrpF=1 => all from Probs2;
```

Generic Procedures

It is possible to pass a procedure any number of arguments of any type without declaring these arguments in advance. To do this, we declare a *ListVar* argument, then use *Expr_Type* and streams to wrap up and decode everything correctly.

```
REQUIRE PlaTab.Def[Pla,Bil];          ∂ get type definitions;
PROCEDURE ExprsToo(LISTVAR l);
BEGIN
INTEGER i;
FOR i←1 STEP 1 UNTIL LENGTH(l) DO
  BEGIN
  CASE EXPR_TYPE(EVAL(NTH(l,i))) OF
    BEGIN
    [t_integer] PRINT("integer: ",EVAL(NTH(l,i)));
    [t_string]  PRINT("string: ",EVAL(NTH(l,i)));
    [t_variable][t_real]  PRINT("real:",EVAL(NTH(l,i)));
    [t_function] PRINT("function");
    [t_flavorvar] PRINT("flavorvar");
    [t_context] PRINT("context");
    [t_listvar] PRINT("listvar: ",EVAL(NTH(l,i)));
    [t_undefined] halt(NTH(l,i)&" is undefined.");
    ELSE HALT("illegal type");
    END;
  END;
END;
Exprstoo({1.0 [reverse({1 2 3})] [new_flavor(test)] "hi"});
```

Macros

A macro has a name, optional arguments, and a body. When the name occurs, Pla substitutes the body (with the arguments, if any) into the Pla code text at that point. Since the substitution is textual and happens before the text is parsed, just about anything can be put in a macro.

Define introduces a list of macro definitions separated by commas and ended with a semicolon. Each definition consists of the macro name (the word which is used in the file to call the macro) followed by the arguments (if any) in parentheses, then an "=", followed by the macro body (the code to be substituted for the macro name) in "⊂⊃". The "⊂⊃" delimiters are not needed if the macro has no arguments and the macro body does not contain semicolons.

```
DEFINE  Amp=P3,
        HighestFreq=440,
        AddTwo(Arg)=⊂Arg+2⊃,
        BigMac(Arg)=⊂SideEffect←Arg+3;
                    Print("BigMac is dangerous")⊃;
```

Perhaps the most common use of macros is shown in the first two examples above: more meaningful names can be given to special variables and constants. Once "Amp" has been defined to be "P3", Pla substitutes "P3" for every occurrence of "Amp" that it comes across. This substitution takes place before the resulting expression is parsed, so the action of "AddTwo" might be a little unexpected:

```
Intgr←AddTwo(2)
```

assigns 4 to Intgr (Pla actually parses Intgr←2+2) but

```
Intgr←4/AddTwo(2)
```

assigns 4 to Intgr (Pla parses Intgr←4/2+2). To avoid this ambiguity it is sufficient to place the body of the macro in parentheses (in this case):

```
DEFINE AddTwo(Arg)=⊂(Arg+2)⊃;
```

"BigMac" defined above consists of more than one expression. The variable "SideEffect" must, of course, be declared before "BigMac" is called.

Just Intonation

We can define alternate pitch names using macros to give us just intonation. The macros given here are based on middle c. Just intonation is based on combinations of the ratios (3/2) and (5/4), the fifth and major third.

```
DEFINE  c!(Oct)←c(c×(2↑(Oct-4)))⊃,
        df!(Oct)←c(c!(Oct)×16/15)⊃,
        d!(Oct)←c(c!(Oct)×9/8)⊃,
        ef!(Oct)←c(c!(Oct)×6/5)⊃,
        e!(Oct)←c(c!(Oct)×5/4)⊃,
        f!(Oct)←c(c!(Oct)×4/3)⊃,
        fs!(Oct)←c(c!(Oct)×25/18)⊃,
        g!(Oct)←c(c!(Oct)×3/2)⊃,
        af!(Oct)←c(c!(Oct)×25/16)⊃,
        a!(Oct)←c(c!(Oct)×5/3)⊃,
        bf!(Oct)←c(c!(Oct)×16/9)⊃,
        b!(Oct)←c(c!(Oct)×15/8)⊃;
```

Another fs is 45/32 (fs = e+d = (5/4)×(9/8)). The difference is the "syntonic comma", which plays a bigger role below. We can use these macros just like note names:

```
Freq←[c!(3),e!(4),g!(5)];
```

which is equivalent to the equal tempered SCORE-version:

```
Freq←NOTE: c3,e4,g5;
```

Pythagorean Tuning

In Pythagorean tuning all notes are derived from the not-quite-circle of fifths, so all ratios are a power of (3/2):

```
DEFINE  P_c(Oct)←c(c×(2↑(Oct-4)))⊃,
        P_df(Oct)←c(P_c(Oct)×256/243)⊃,
        P_d(Oct)←c(P_c(Oct)×9/8)⊃,
        P_ef(Oct)←c(P_c(Oct)×32/27)⊃,
        P_e(Oct)←c(P_c(Oct)×81/64)⊃,
        P_f(Oct)←c(P_c(Oct)×4/3)⊃,
        P_fs(Oct)←c(P_c(Oct)×729/1024)⊃,
        P_g(Oct)←c(P_c(Oct)×3/2)⊃,
        P_af(Oct)←c(P_c(Oct)×128/81)⊃,
        P_a(Oct)←c(P_c(Oct)×27/16)⊃,
        P_bf(Oct)←c(P_c(Oct)×16/9)⊃,
        P_b(Oct)←c(P_c(Oct)×243/128)⊃;
```

Mean-tone Tuning

Mean tone tuning is based on a fifth that is one fourth of the syntonic comma smaller than a perfect fifth. The idea here is that the major third is pure and the fifths are close. We also provide a few split tones. We use real numbers, since mean-tone tuning is a kludge anyway, and the fractions are cumbersome:

```
DEFINE  M_c(Oct)←(c*(2↑(Oct-4)))⊃,
        M_cs(Oct)←(M_c(Oct)*1.84501233)⊃,
        M_df(Oct)←(M_c(Oct)*1.86990725)⊃,
        M_d(Oct)←(M_c(Oct)*1.11806627)⊃,
        M_ds(Oct)←(M_c(Oct)*1.16839304)⊃,
        M_ef(Oct)←(M_c(Oct)*1.19622721)⊃,
        M_e(Oct)←(M_c(Oct)*5/4)⊃,
        M_f(Oct)←(M_c(Oct)*432/323)⊃,
        M_fs(Oct)←(M_c(Oct)*1.39766355)⊃,
        M_gf(Oct)←(M_c(Oct)*1.43095954)⊃,
        M_g(Oct)←(M_c(Oct)*323/216)⊃,
        M_gs(Oct)←(M_c(Oct)*1.56268047)⊃,
        M_af(Oct)←(M_c(Oct)*1.5999076)⊃,
        M_a(Oct)←(M_c(Oct)*1.67192317)⊃,
        M_bf(Oct)←(M_c(Oct)*1.78880273)⊃,
        M_b(Oct)←(M_c(Oct)*1.86932091)⊃;
```

EvalNow

There are occassions where we want the source code produced by a macro to be the result of some arbitrary computation based on the current state of the world. EvalNow causes Pla to evaluate the argument to EvalNow. The string result of the evaluation replaces the EvalNow statement and the result is then parsed as Pla-code. It is recommended that the argument be a string procedure.

Voices

A voice is the main music-producing entity in Pla.

Voice Declaration

A Voice is a process that runs whenever Pla thinks a new note from that voice is needed. A Voice is created by the word "Voice" followed by the voice name and by the expressions that define that voice. After the voice name you can also include the first and last begin times of the voice in parentheses, separated by a colon. A new begin time can be included outside the parenthese to tell the voice to shift the window some amount. The shift amount is used mostly to move note list portions around in a mix, but voices sometimes use it for an initial rest. For example:

```
Voice Simp;
```

creates a voice named "Simp" which starts execution at time 0 and goes forever, unless something else tells it to stop (there are many ways this can happen).

```
Voice Simp(0:12);
```

creates "Simp" which starts at 0 and goes until time 12, unless something else stops it first.

```
Voice Simp(0:12) 3;
```

creates "Simp" which now goes from seconds 3 to 15.

```
Voice Simp 3;
```

creates "Simp" which starts at time 3 and goes until someone stops it.

P1 (Begin Time)

The first and last begin times can be thought of as a frame placed around the voice's actions. If the voice does not set P1 itself, Pla starts the voice at the first begin time. If the voice sets P1 itself, Pla makes no further attempt to mess with it. Only those notes that occur within the (begin:end) window get to the note list. You can set P1 explicitly to have a voice play a chord, perhaps, or overlap itself.

```
Voice Overlap(10:15);          ∂ only output notes that occur;
  Begin                        ∂ between seconds 10 and 15;
  Real OldP1,NewP1,Dur;
  NewP1←OldP1+Dur;             ∂ P1 according to un-overlapped Dur;
  OldP1←NewP1;                 ∂ update notion of last P1;
  P1←NewP1+10;                 ∂ add in begin time explicitly;
  Dur←Rhythm: e,e,q,h;         ∂ "onset" rhythm (independent of P2);
  P2←Dur*1.2;                  ∂ 20% overlap;
  End;
```

Some care should be exercised when you set P1 yourself because Pla does not check that P1 makes sense in this case. If a voice uses the shift feature mentioned above and sets P1 itself, the shift will be ignored (the P1 assignment takes precedence). In general, if you want to set P1, use MYBEGIN and MYEND, rather than the window and shift features.

There is another, perhaps cleaner way to get overlaps and rests—use WaitUntil.

```
Voice Overlap;
  Begin
  P2←Rhythm: q,q,h,w;
  WaitUntil(P1+P2/2);
  End;
```

Voice Body

After the voice name declaration come the expressions that define what the actions of that voice are. The block of expressions must be within a single Begin_End statement. Pla parses all the expressions connected with all the voices, then sets up a queue of specific notes from each voice. Whenever a note is output, Pla asks the voice responsible for that note for another note. At that time all the expressions associated with the voice are again evaluated.

I_Only and F_Only

There are two exceptions to the evaluation order given above: I_Only and F_Only blocks are evaluated only at initialization time (I_Only) or at the very end of a voice (F_Only).

```
Voice Simp;
BEGIN
I_Only: MyNotenum←2;
p2←2;
F_Only: Print("All done.");
END;
```

creates a minimal voice named SIMP which outputs 2 notes, then prints, "All done". If "MyNoteNum←2" were not an I_Only statement, it would be evaluated every time the voice is called, and SIMP would go on forever, always thinking it had just 2 more notes to output (MyNoteNum is explained below).

Mutes

A Mute is a voice that does not directly output anything. It can be used to control other voices, for example, or keep track of statistics. A mute voice is declared and treated just like a normal voice except that the word "Mute" is used in place of the word "Voice". "MyParNum←0" also creates a mute voice.

Voice Environment

Any parameter of a voice can be accessed by: *Voice_Name:Parameter_Name*. (That is: the name of the voice, colon, the name of the parameter). If we have a voice named "Simp" which has a parameter named "Amp", we can get at or change the value of that parameter by the name "Simp:Amp". A string expression can be used for the voice name. The parameter can be a reference to the "P" array:

```
SIMP:P[i]←i+1;
```

Pla can become confused if several voices have the same name—Pla just scans its voice queue looking for the first occurrence of the voice name and uses that.

Each voice or file carries around with it some possibly useful information which you can access or change. The following words are predefined:

MyBegin	Begin time of file or voice
MyEnd	Voice End Time
MyDur	Voice Duration
MyStatus	Status of voice
MyType	Type of voice
MyNoteNum	Number of notes left to output
MyMessage	voice's message dictionary
MyParNum	number of parameters to output

MyStatus can have the following values:

Dead	voice is slated for garbage collection
Alive	voice is running normally
Stopped	voice is suspended
Removed	voice is temporarily suspended
Waiting	voice is waiting for its first begin time
IWait	voice is running in wait-status
Undefined	voice has no known state

MyType can have the following values:

Header	header block
Voice	normal voice
Mute	voice with no output
Section	a group of voices
File	a play file being mixed
User	a TTY: channel for interactive input
Finish	a block executed only at the end of compilation
Always	a block which is executed after each voice is run
Undefined	No type

PlaTab.Def[Pla,Bil] contains the definition of these macros. It can be loaded into Pla by **Require** PlaTab.Def[Pla,Bil]; The Voice type cannot be changed.

The voice status **Dead** signifies that the given voice is slated for destruction as soon as Pla's garbage collector gets around to it. **Stopped** means the voice has been stopped by someone. **Removed** means the voice has been UnQueued. **Waiting** happens during initial rests. **Undefined** occurs after a dead voice has been partially garbage collected.

A voice becomes Dead if:

1) It is KILLed,
2) NumNote is exceeded (MYNOTENUM=0),
3) MYEND is exceeded,
4) End of File or End of Section is encountered,
5) No voice is ALIVE,
6) MYSTATUS has been set to DEAD.

EndTime defaults to infinity (2↑30 right now).

If you assign a value to MyDur, be sure you first assign some reasonable value to MyBegin.

Dynamic Voices

Voices can be created within statements. When using dynamically created voices, it may be helpful to think in the following terms: the word "voice" signifies the creation of a completely independent entity (process). Each instantiaition of a voice has a pointer to the original template of expressions, its own state variables, and its own local variable context.

```
IF P3=3
  THEN                    ∂ every time P3=3, create a new SIMP;
    VOICE Simp (0:10);    ∂ SIMP runs from its creation time to 10;
        BEGIN
        p2←2;             ∂ set all of SIMP's parameters;
        p3←3;
        END;
```

When the If condition is true, causing evaluation of the statement following the Then, a totally independent voice is created which follows its own path from that moment on. A canon of 4 voices can be created by:

```
integer i;
for i←1 step 1 until 4 do
  voice ["soprano","alto","tenor","bass"] ([1,2,3,4]:[6,7,8,9]);
    begin
    p2←1;
    p3←[1,3,2,5,4];
    end;
```

Each time through the loop the expressions following the word "voice" are evaluated. Four independent voices are created, named SOPRANO, ALTO, TENOR, and BASS, with begin:end times of (1:6), (2:7), (3:8), and (4:9) respectively. The body of the voice is not executed at the time it is instantiated, but any I_Only block is. If we want to know later the order in which the voices were created, we can save the loop counter (using the example above) by declaring a local integer variable, and setting it to the value of the index at I_Only time:

```
for i←1 step 1 until 4 do
  voice ["soprano","alto","tenor","bass"] ([1,2,3,4]:[6,7,8,9]);
    begin
    integer Order;
    I_Only: Order←i;
    p2←1;
    p3←[1,3,2,5,4];
    end;
```

Because dynamic voices can be created within procedures or by other voices, Pla provides several ways, besides Sections, to organize a group of voices into a single event. In the following example, we use a procedure to pass parameters to a voice:

```
Procedure GraceNote(Real Freq,Beg);
Begin                       ∂ grace note at Freq and Beg-.05;
Voice Grace;                ∂ the voice producing the grace note;
  Begin
  Real Local_Freq,Local_Beg;
  I_Only: Begin
          MyNoteNum←1;      ∂ just one note in the ornament;
          Local_Freq←Freq;
          Local_Beg←Beg;
          End;
  P1←Local_Beg-.05;         ∂ BEG is calling voice's next begin time;
  P2←.05;
  P3←Local_Freq;
  End;
End;

Voice CallIt;               ∂ will get the ornamentation;
Begin
Integer GraceVar;           ∂ for the grace note array constant;
P2←[1,1,2,1,1,Kill];        ∂ after 4 notes, kill CALLIT;
P3←[a,b,c,d];
GraceVar←[GraceNote(ds,P1+P2),
          0,
          GraceNote(e,P1+P2),0,0,0];
End;                        ∂ 2nd and 4th notes get a grace note;
```

This code produces the following output play list:

```
PLAY ;
CALLIT,  .000,  1.000,  440.000;
GRACE,   .950,  .050,   311.130;
CALLIT,  1.000, 1.000,  493.890;
CALLIT,  2.000, 2.000,  261.620;
GRACE,   3.950, .050,   329.630;
CALLIT,  4.000, 1.000,  293.660;
CALLIT,  5.000, 1.000,  440.000;
FINISH;
```

A voice created in this way can reference variables local to the calling voice, and can continue to run after its parent has finished. It is an error, however, to continue to try to use the variables of the parent voice after it has been deallocated. To enable voices to keep track of the current environment, **Self** is a predefined function returning the name of the currently active voice, and **Caller**(*voice_name*) returns the name of the voice that called *voice_name*. If **Caller(Self):Mystatus=0**, the calling voice has been deallocated and its variables no longer exist.

Note that when a voice accesses a procedure's parameters, it should read the values of those parameters into variables local to the voice at L_ONLY time.

The Voice Queue

Pla treats each voice as a separate process, and has its own scheduling system to keep track of what voice to call next. This list of voices waiting for service is called the queue. You can get at any voice on the queue and any parameter of that voice. The voices are numbered according to their position (1 is the first). **QLen** is a global variable telling how many voices are waiting on the queue. If QLen is 0, do not try to mess with the queue! Each voice can be accessed by:

Queue [*voice_number*] : *parameter_name*.

This syntax is very similar to the voice "subfield" syntax (see Page 44).

To remove a voice from the queue, use UnQueue(*voice_Number*). To remove the current voice (which hasn't yet been put on the queue, but is being evaluated), use UnQueue(Self). When a voice is UnQueued, it becomes the next voice to be evaluated and reinstated, if possible on the queue. If you change a voice's begin time while it is on the queue, Pla tries to reposition it in the correct place, but don't expect miracles.

Stop, Start, and Kill

Stop, Start, and Kill give you a way to turn voices on and off according to some arbitrary condition.

Kill(*Voice_Name*) turns off voice *Voice_Name* permanently. Attempts to start him later are ignored. Kill without an argument kills the current voice (equivalent to Kill(Self)).

Stop(*Voice_Name*) stops *voice_name*. Stop without an argument suspends the current voice (equivalent to Stop(Self)). You can Stop a stopped voice (nothing happens). If only stopped voices are present on the queue, the queue is emptied and processing stops.

Start(*Voice_name*) awakens *Voice_Name* from suspension. You can start an active voice (nothing happens).

When a voice is stopped, even if it is in the midst of processing a note, it is shoved to the end of the queue and does not carry out any action until it is started. A stopped voice has no way to start itself. When a stopped voice is started, it becomes the very next voice to be serviced by the evaluator. If a voice starts several voices, they are serviced in reverse order because each is put at the head of the queue.

With UnQueue(Self), Stop(Self), and Kill(Self), it is of interest where the voice processing actually stops. In general, the entire body of the voice is evaluated before the voice is rescheduled to allow the voice to retract the scheduling action. To exit prematurely from anywhere within a voice body, use RETURN.

Message Passing

Each voice has a message dictionary (a list) called MyMessage. Each message consists of name, optional arguments, and an action to take when that name is called. A voice can cause that action to take place in the context of the called voice without causing rescheduling by sending that voice the appropriate message. Messages are declared:

Message *name*(*parameters*); *body*

All the components of the declaration are the same as in a procedure declaration. The message is called by passing the voice the message name (and arguments) using the ":" syntax.

```
MESSAGE No_Op(INTEGER N);
    RETURN(N);
```

if declared within the body of SIMP, can be called from another voice:

```
SIMP:No_Op(N);
```

or from SIMP itself:

```
SELF:No_Op(N);
```

The Flavor chapter contains a more complete discussion of message passing.

WaitUntil

When a voice is running normally, the scheduler first sends out its current play list data, then evaluates its body to get the next note's data and begin time. In a sense, the scheduler is always one note ahead in each voice. This behaviour can make it difficult to tell when a given voice will execute, making synchronization between voices unnecessarily complex. A voice can specify that its body is to be evaluated at the same time as the data is sent by calling the procedure WaitUntil. WaitUntil's argument is the time of the next note of the current voice. Unless you are being very clever, you probably want to call WaitUntil on every note of a voice that is using that scheduling status.

Pars, Defset, and Tempo

Pars

Pars can be used to give PFields names. It is slightly faster and more compact than the equivalent use of Define. The parameters' names starting with P0 are given in a list following the word Pars. If P0 is the instrument name, P1 the begin time, p2 the duration, p3 the frequency, and p4 the amplitude, we might want to name them InsName, Beg, Dur, Freq, and Amp respectively:

```
PARS: InsName, Beg, Dur, Freq, Amp;
```

These names can thereafter be used instead of P numbers. The Pars declaration can occur anywhere a Define can occur (it is just a special way of making macros). All names thus declared are known globally from the point of definition, so it is safer to keep the names unique.

DefSet

A variant of Pars is DefSet. It defines the identifiers following it to be macros with values increasing by one.

```
DEFSET: one, two, three;
```

defines one=1, two=2, and three=3.

Tempo

Tempo(*expr*)

Tempo can be used to change the tempo of all events in a file. Tempo(2) causes all events to happen twice as fast. If the Tempo statement occurs within a voice, it applies only to that voice and overrides any global tempo statement.

Tempo can bring about non-obvious behaviour in Pla. The problem is somewhat complex and might bear explication (feel free to skip this footnote):

> A voice can set P1 itself. Here the question of "what is P1?" becomes rather difficult to answer. Say we want to double the speed of all events in a file (Tempo(2)). P2 is easy: in every case we just divide P2 by the tempo and the event will take half as much time to complete. P1, however, has to be different from the previous P1 by half the amount that the present P1 (before the application of the tempo factor) was different from the previous (untempo-factored) P1. That is, if we have two events starting at times X and X+.5 when the tempo is 1, we want them to start at some time Y and Y+(.5)/2 when the tempo is 2. We can't just calculate X/2 and (X+.5)/2 because the tempo might be changing—we can't assume it has always been 2. "Y" has to depend on the previous post-tempo P1, the previous pre-tempo P1, and the present pre-tempo P1. The calculation for P1 becomes:

```
P1 ← PreviousPostTempo_P1 +
        (PresentPreTempoP1 - PreviousPreTempoP1) / Tempo.
```

This works for a simple case like a section or a file. But what if a voice has been calculating its own P1 according to some formula that involves the previous P1 value (as in "P1 ← P1 + .5")? It assumes that P1 is the Previous PRE-Tempo P1, but P1a gives it the real P1 (the Present-POST tempo P1). If P1a were to try to be very smart and insert the Previous Pre-tempo P1 whenever a voice asks for P1 if Tempo is being used and if P1 is being assigned a value, how would the voice (in this not impossible situation) ever be able to get at the REAL present P1? In other words, we have a dilemma—P1 is not what it seems because Tempo messes with it in non-intuitive ways.

We might next decide that auxilliary variables are the answer. We use them (not P1, which P1a is messing with), to calculate the value of P1 so that the voice is guaranteed to have a legitimate Previous-Pre-Tempo P1 to deal with. As an example, say we want the notes of a voice to be overlapping slightly, and want to apply a tempo change according to a function:

```
SEG TempoFnc  1 0 3 100;

TEMPO(TempoFnc[p1]);     ∂ goes from 1 to 3 in 100 seconds;

REAL dur,dur1,oldp1a,oldp1b,newp1a,newp1b;
            ∂ Dur=pre overlap duration.
              OldP1 = previous pre-tempo begin time.
              NewP1 = present pre-tempo begin time;

VOICE simp(0:100);
BEGIN
newp1a←oldp1a+dur;  ∂ Calculate begin time;
oldp1a←newp1a;      ∂ save it away for next time;
p1←newp1a;          ∂ set P1 ;
dur←rhythm:q,e,h;   ∂ Dur is the time until the next note;
p2←dur*1.2;         ∂ P2 > dur, so we get an overlap;
END;
```

Even this won't act quite as we hope, however. First, if two voices are being used, simultaneities in the pre-tempo version may not be simultaneities in the tempo-factored form. For example, say our TEMPO=1 output is:

```
SIMP    0 1;
TOOT     0 .5;
TOOT     .5 .5;
TOOT     1 1;
SIMP    1 1;
```

SIMP and TOOT come together at time 1. Now if we apply a function to the tempo, both start out together, but TOOT's second note is a different length (shorter if the tempo is going faster). Since each voice tracks his own P1 value (not the previous Pre-tempo-p1), TOOT's third note precedes SIMP's second note (6). To ensure that simultaneities in one version are also simultaneous in another after the application of a tempo function, we must integrate the tempo expression.

(6) Files and sections do the right thing here

As it turns out, the integration is not really so difficult and does allow you to have arbitrary functions on the tempo without losing anything from the previous version (no overlaps, gaps, or lost chords). In fact, if each voice had its own tempo factor, we can produce very complex and subtle tempo phrasing. The trick lies in making a map of the old time versus the new time (integrating the tempo function), then using variables in the voice to hold the sum of all previous P2's and the present P2, using that value to lookup the map, then getting the new P1 and P2 values by working backwards from the old P1 and the old P1 + P2.

```
P2←Map(OldP1+Our)-OldP1;
NewP1←OldP1+P2;
```

This method is used in the following example. Each voice keeps track of its own tempo. In all cases however, if the tempo functions meet at some point, or travel together, then at that time events in the separate voices coincide correctly, no matter what the preceding tempos looked like. There are no unexpected overlaps or gaps. The voice's tempo is the first derivative of the tempo function (its slope)—Simp1 always has a tempo of 1 in this example, but Simp2 starts at 1.5, then goes to .5.

```
Real simp1Tempo,Simp2Tempo,Our1,Our2,allOur1,allOur2;

Seg S1Tmp  0 0
           20 20
           100 100;

Seg S2Tmp  0 0
           10 15
           20 20
           100 100;


Voice Simp1 (0:20);
  BEGIN
  Our1←Rhythm: q;
  AllOur1←AllOur1+Our1;
  Our1←S1Tmp[AllOur1]-p1;
  P2←Our1;
  END;

Voice Simp2 (0:20);
  BEGIN
  Our2←Rhythm: q;
  AllOur2←AllOur2+Our2;
  Our2←S2Tmp[AllOur2]-p1;
  P2←Our2;
  END;
```

The tempo functions map the normal time against the tempo-factored time (remember that Seg functions are backwards, so the Y axis (the first of the pair) is the new time). The tempo functions should go past the last begin time to avoid infinite loops.

Experiments with changing tempos indicate that line segment tempo functions sound bad. A natural sounding tempo function has to have a different tempo for every note. It is possible, though tedious, to create Seg functions with hundreds of points, but we soon suspect that our data representation is the wrong one. What we really appear to need are mathematical functions like SINE or LOG. The cleanest way to apply such functions in Pla depends on what note production method we are using.

As currently implemented, TEMPO(<expr>) does the right thing in all cases for files and sections. For voices, it just divides P2 by the current value of <expr>. This will work if you are not setting P1 yourself. If you are, you should probably handle tempo changes by hand.

Files

Pla knows about three kinds of files: Pla code files, function definition files, and note lists. Any arbitrary file can be read or written using the File I/O procedures discussed later. A Pla code file can itself require other files (via Require) to any level of nesting. To open a note list, use **File** *filename*. If **File** occurs within an expression, *filename* is opened as a dynamic voice—*filename* is treated as an expression. When a function file is opened (via Func), the function definitions are read into Pla Functions, then the file is closed.

Note List Mixing and Editing

File *Filename* opens *FileName* and creates a voice node to handle it. Anything a voice can do, a file can do. For example,

```
FILE Test.Pla (3:21.2) 0; BEGIN END;
```

takes all the notes in TEST.PLA that begin between seconds 3 and 21.2 and shifts them back to fall between times 0 and 18.2. If any changes are desired in TEST.PLA while it is being mixed into the output file, the changes are given in the block following the word **File**:

```
FILE Test.Pla;
  BEGIN
  P4←P4*2;
  IF P3=440 THEN p3←660;
  END;
```

reads TEST.PLA into the output, doubling every P4 and changing any P3's that are 440 to 660. In this example the begin times are left alone.

Note lists are read just as Sambox reads them: everything between **Dont_Scan** and **Scan** is ignored, multiple **Play** or **Pass** statements cause all subsequent begin times to be reinterpreted, comments in "∂" and "<" are ignored, **Included** files are inserted directly into the note list.

When multiple note lists are read in, all the headers are saved before the output header is created so that multiple **Nptix**, **Common**, **Func**, **Record**, **Nutix**, **SRate**, **Ask**, **Output**, **Play**, **PFinish**, **IntDeb**, **Insert**, and **Pass** statements can be handled correctly. The output file should always be directly compilable by Sambox, even if the NODEFAULTS flag in Sambox is on. Any statement longer than the size of the current string expansion space is assumed to be a comment. This length is guaranteed to be greater than 4000 characters.

Pla assumes that you want only one call on a reverberator, and automatically deletes any extra calls it finds while mixing files. To turn off this feature, name your reverberator something other than "REVRB" or "REVERB".

Omit

To omit a voice call from a Note list, use UnQueue (Page 48). To omit a non-voice call (a **Print** statement for instance), use **Omit**. **Omit** followed by a list of tokens throws out any statement encountered in the note list which starts with one of those tokens.

```
OMIT("Print Toot");
```

omits all PRINT statements and all calls on TOOT. Omit can occur anywhere in an expression that a procedure call can occur.

Function Files

A function file consists of a sequence of Seg and Synth function definitions in a truly bizarre format (7). You can create function files using the FUNC, TFUN, or EDFUN programs, but I recommend the editor. When using the editor just remember that Pla only knows about Seg functions (line segment envelopes) which are defined:

SEG *name* y1 x1 y2 x2 ...

Name can be have any number of characters. To read a function file into Pla, use Func *fileName*; The keyword Record is identical to Func.

(7) This format dates from the beginning of recorded history

Input/Output

Print sends output to the console, and **CPrint** sends output to a text file. **RdNum** gets keyboard input, and **Get** and **Seek** perform text file input. Output files other than the note list need to be explicitly opened. Pla automatically closes all files. If you want to write a new output file and read it back in during the same compilation, you must write the data, close the file, then reopen it.

Print

Print (*text*)

Print sends *text* to the console. Use the "&" operator (concatenation), to include several pieces of information in a single **Print** statement.

```
PRINT("I is "&i)
```

prints out "I is 5" (or whatever the value of I is). When a listvar is printed, the "{}" are printed as "()". Print can take any number of arguments.

RdNum

value ← **RdNum**;

RdNum waits for input from the keyboard, then handles it as a string and a real number. The expression context determines which value is used. Eval can be called on a string, so you can input an expression to Pla in a string and have it evaluated. RdNum is similar to Inchwl in SAIL.

CPrint

CPrint (*file-listvar*, *text*)

Any number of separate output files can be open during a compilation. Use Cprint to write data to these files. *file-listvar* is the listvar returned by **Open**. **CPrint** sends *text* to that file. CPrint, like Print, can have any number of arguments.

To write directly to the note list being compiled, use 0 as the first argument to CPrint. Pla cannot assume that something being CPrinted is in a format in which P1 and P2 have the normal meanings, so it cannot mix CPrinted data as it would voice calls. It is up to you to ensure that *text* makes sense to the music compiler. CPrinted data follows the voice call in the note list. To put a string just before the voice call, use P0.

Open

file-listvar ← **Open** (*file-name*, *access-mode*) ;

Use Open either to get arbitrary access to a section or note list, or to open a file for output. *Access_mode* can be either Read (1), Write (2), or Read-Alter (3). Open returns a listvar pointing to the file or section. This listvar's value should *not* be changed, nor should it be used in any expression. Use it only as an argument to CPrint, Seek, or Get. Any number of files can be open at once.

Close

Close (*file-listvar*) ;

Close closes the file that *file-listvar* points to. Pla automatically closes any open files at the end of a compilation.

Get

data-list ← **Get** (*file-listvar*) ;

Once a file or section has been opened, Get. can be used to march through the file sequentially. Get returns a listvar, *data-list*, containing the next line of the file or section. The "next line" is considered to be everything up to the next semicolon.

Seek

data-list ← **Seek** (*file-listvar*, *line-number*) ;

Once a file or section has been opened, you can access any "line" of it with Seek. Seek returns a listvar containing the data on the line specified. *Line_number* is a logical note list line: everything up to the next semi-colon is considered to be one line. Each call on Seek causes the file marker to be put back to the first record, then stepped forward until the desired line is found, so Seek is not super-fast.

Sections

A group of voices can be packaged as a Section which can thereafter be treated as a voice. We define a section by:

```
SECTION name;
BEGIN
statement list;
END;
```

where *statement list* can be empty, or contain any number of legal statements. At least one of the statements must be a voice, file, or section call. A section can be instantiated any place a voice statement is legal by using the word CallSection with the section name followed by any operations that are desired:

```
CallSection name (optional window) optional shift;
    BEGIN                  ∂ (name should be unique).
    p4←p4*.5;              ∂ make entire section softer (p4=amp);
    p3←p3*2;               ∂ transpose up an octave (p3=freq);
    if p0="toot"
      then p4←.5;          ∂ make all TOOT notes louder;
    if p0="simp"
      then unqueue(0);     ∂ omit all SIMPs;
    if p0="hithere"
      then p0←"toot";      ∂ change all HITHEREs to TOOTs;
    END;
```

Once a section has been defined, it is precalculated by Pla and handled as though it were a note list. The voice window and shift syntax (see Page 43) also applies to sections:

```
CALLSECTION SectA(X:Y) Z;
```

takes only those notes in the section which fall between seconds X and Y and offsets them by Z-X seconds (that is, the section window starts at time Z).

A section or a file can be called from within an expression. The syntax is exactly the same as that for voices. For example:

```
section hi;      ∂ define a section containing 4 notes of voice HO;
begin
  voice ho(0:3);
  begin
  p2←1;
  p3←p3+1;
  end;
end;

integer i;       ∂ now get three copies of that section;
for i←1 step 1 until 3 do
  callsection hi;
    begin
    p4←p4+1;
    end;
```

Sections are implemented as temporary files. If you are calling a large number of voices, it may help keep the core size reasonable to break the code into several sections, then call these sections at the end.

Section editing

If you need arbitrary access to a section's data (CallSection has some limitations), you can keep track of each section and convert its data into lists with the following procedures:

```
LISTVAR SectionNames;  ∂ list of {section-name file-name} pairs;
INTEGER SectNum;       ∂ section counter;

PROCEDURE AddName(STRING SectName; REAL Duration);
BEGIN                  ∂ keep track of section file names;
LISTVAR ThisPair;
SectNum←SectNum+1;
ThisPair←QUOTE({0 0});
NTH(ThisPair,1)←SectName;
NTH(ThisPair,2)←Duration;
ThisPair←INSERT(ThisPair,"PLA"&SectNum&".TMP",3);
SectionNames←APPEND(SectionNames,QUOTE(ThisPair));
END;


PROCEDURE GetArg(STRING SectName; INTEGER Which);
BEGIN
INTEGER i;
i←1;
WHILE (i<LENGTH(SectionNames)) DO
  BEGIN
  IF NTH(SectionNames,i)=SectName
    THEN RETURN(NTH(SectionNames,i+which));
  i←i+3;
  END;
PRINT("Can't find "&SectName&↓);
END;

PROCEDURE GetName(STRING SectName);
RETURN(GetArg(SectName,2)); ∂ return Section file name;

PROCEDURE GetDur(STRING SectName);
RETURN(GetArg(Sectname,1)); ∂ return section duration;


PROCEDURE ConvSectToList(STRING SectName; INTEGER WhichWay);
BEGIN       ∂ convert section data to list of lists;
LISTVAR SectList,SectData,FilPtr;
SectList←QUOTE({});
FilPtr←OPEN(GetName(SectName),1);
SectData←GET(FilPtr);
WHILE NOT NULLP(SectData) DO
  BEGIN                        ∂ read in section data;
  SectList←INSERT(SectList,LISTIFY(SectData),WhichWay);
  SectData←GET(FilPtr);
  END;
CLOSE(FilPtr);
RETURN(SectList);
END;
```

In each section, the FINISH block should contain a call on AddName with the section's name and its total duration.

```
FINISH: AddName("One",p1);       ∂ Section One;
```

Reverse Section

Given the procedures above, we can easily produce the retrograde of a section:

```
PROCEDURE ReverseSection(STRING SectName);
RETURN(ConvSectToList(SectName,1));
```

This list can be traversed by a voice, or we can open a file and write it out ourselves:

```
PROCEDURE ListToSect(LISTVAR L; STRING SectName);
BEGIN
LISTVAR FilPtr;
STRING S;
INTEGER i;
FilPtr←OPEN(GetName(SectName),2);
FOR i←1 STEP 1 UNTIL LENGTH(L) DO
  BEGIN
  S←NTH(L,i);
  CPRINT(FilPtr,S&";"&↓);
  END;
CLOSE(FilPtr);
END;
```

Pla Directives

PField

PField (*num*)

Pla creates an array to hold each voice's parameters. The array's default size is 128. If you use more than 128 parameters, you must use PField to get the needed room. If you do not need 128 pfields, you can save yourself a substantial amount of core space by calling PField with the maximum number of fields you need.

SetFormat

SetFormat (*num*)

If you want more (or less) precision than Pla's default, call SetFormat. *Num* is the number of digits printed after the decimal point, in most cases. Pla's default is three.

Exit

Exit (*filename*)

Exit names the output note list file. If you specify the name with Exit, Pla does not ask for any more input or output files after finishing with the present file, but, instead, it goes immediately to the compilation of the output file.

CallSambox

CallSambox causes Pla to exit automatically to Sambox—your own, if it exists, otherwise to the system version, once the note list compilation is complete.

Swap

Swap (*String*)

At the end of the compilation, after the output note list has been closed, you can specifiy that some arbitrary string be typed to the TTY input buffer. Swap sends *string* to your terminal, causing *string* to be treated as a monitor command.

```
SWAP("RU SAMBOX[PC,BIL];NC2"&ASCII(13));
```

Pty

Pty fires up a subjob to which you can issue arbitrary monitor commands whilst inside Pla. Type Ctrl-Q to return to release the subjob.

Save

Save (*filename*)

Save saves the complete state of your executing Pla image in *filename*. If no extension is given, ".SAV" is used. To restart Pla from the point of the Save command, type RU *filename*. It is a little dangerous to change or delete files that were open at the time of the save.

GetInfo and SetInfo

val ← **GetInfo**(*index*)
SetInfo(*index*, *new-value*)

These two procedures provide access to a number of special purpose Pla variables. *Index* is the index of the entity to deal with. **GetInfo** returns its value, **SetInfo** sets its value to *New_Value*. *Index* currently has the following meanings:

1	Tempo
2	RealBeg (current Pla-P1)
3	PMax (maximum pfield number)
5	QLen (length of scheduler's queue)
7	Octave divisions for ∪ and ∩
9	Queue display flag (true=on, default=false)
17	GogTab['250] (RecGC, True=Off)
18	RecGC trigger minimum (1024 is Pla default)
19	True if voice is currently in a chord

Index 7 sets the number of octave divisions assumed by the sharp (∩) and flat (∪) operators (see Page 10). The default value is 12. The "queue display" flag is discussed on Page 97. Don't expect RealBeg to have any obvious relation to an instrument's P1.

Global_PFields

Normally, Pla keeps voice parameter values insulated from other voices. **Global_Pfields** turns off this feature.

Always

Always: *statement*

The **Always** statement causes Pla to execute *statement* after each voice call. *Statement* can perform bookkeeping or debugging functions, for example. Only one **Always** block is allowed per compilation.

```
ALWAYS: BEGIN
        PRINT("p1 is now "&P1&↓);
        IF p1>2000 THEN HALT("oops...");
        END;
```

Finish

Finish: *Statement*

The statement following the word **Finish** is executed only at the end of the Pla compilation, just before the word "FINISH;" is sent to the output note list. Only one **Finish** statement is allowed per compilation.

Header

Header: *string*

Header statements get put in the header portion of the note list. There can be any number of **Header** statements in a compilation.

```
HEADER( "NPTIX←256;
        COMMON SIMP SIMPA SIMPB;
        RECORD NORM.FUN[1,BIL]");
```

The argument to header can be an expression.

Require

Require *filename*

Require suspends compilation of the current Pla code file and continues with *filename*. Pla code files can be nested to any depth.

Call

Call (*expr*, *string_expr*)

Call is akin to SAIL's CALL function. It is provided for the UUO experts among us.

CALL(0,"DSKPPN") returns (as a string) the PPN you are aliased to.
CALL(0,"GETPPN") returns the PPN you are logged in under.
CALL(0,"EXIT") exits Pla entirely.
CALL(0,"RUNTIM") returns the run time your job has consumed so far
CALL(0,"PJOB") returns your job number
CALL(0,"MSTIME") returns the current time in UUO format
CALL(0,"DATE") returns the current date
CALL("200000,"SHOWIT") turns off display of the current file state
CALL(-1,"BEEP") flashes your wholine.

and so on... See the UUO manual if this kind of thing interests you.

Graphics

Pla's graphics routines provide some access to the GrnLib library of display routines. The routines are described in GrnLib.Bil[Mus,Doc].

The available GrnLib routines are:

```
ClearScreen
DpyStr(S,AddCRLF,WriteIt,X0,Y0,dX,dY,Lx,Mode)
DpyVct(X0,Y0,dX,dY,Mode)
InitID(n,X0,Y0,X1,Y1)
Style(n,Styl)
Show(n)
Empty(n)
Line(n,dX,dY)
LineTo(n,X0,Y0)
Box(n,dX,dY)
BoxTo(n,X0,Y0)
Hop(n,dX,dY)
HopTo(n,X0,Y0)
Text(n,S,X0,Y0,dX,dY)
GrnRd(n,Filename,FileDevice)
GrnWrt(n,Filename,FileDevice)
TurtlX(n)
TurtlY(n)
LptGrn(n)
GrnMap(n,m)
```

where "n" and "m" are INTEGER ARRAYs declared in Pla. **ClearScreen** clears the screen; **DpyStr** does immediate text output; **DpyVct** performs immediate vector and box output; **InitID** initializes a graphics buffer (an integer array); **Style** sets the current write mode for the buffer; **Show** displays the buffer; **Empty** empties the graphics buffer (erases the picture); **Line** adds a line to the graphics buffer; **LineTo** does the same but goes to a specific point whereas Line goes a specific distance from the current point; **Box** adds a box to the buffer; **BoxTo** does the same as Box, but goes to a specific point; **Hop** hops (without drawing) to a new point in the display; **HopTo** also hops, but goes to a point rather than a distance from the current point; **Text** adds text to the buffer; **GrnRd** reads in a .PLT file; **GrnWrt** writes a .PLT file; **TurtlX** returns the current x drawing position in the display; **TurtlY** returns the current y drawing position in the display; **LptGrn** and **GrnMap** are currently unimplemented.

A Pla-readable form of Grn.Hdr can be found in PlaLib.Txt[Pla,Bil]. It also sometimes exists as GrnPla.Hdr[1,Bil]. The example programs in GrnLib.Bil[Mus,Doc] can be used with very few changes. For example, the program given under DPYVCT becomes in Pla:

```
REQUIRE GrnPla.Hdr[1,Bil];
INTEGER j,i;

ClearScreen;

COMMENT First make  big box;
DpyVct(10,10,0,400,LWMV);
DpyVct(10,410,400,0,LWMV);
DpyVct(410,410,0,-400,LWMV);
DpyVct(410,10,-400,0,LWMV);

COMMENT Then add a bunch of dots;
j←RAN(8);
FOR i←1 STEP 1 UNTIL 100 DO
    DpyVct(10+RAN(j)*400,10+RAN(j)*400,0,0,0);
COMMENT Now a couple rectangles;
DpyVct(250,250,50,50,0);
DpyVct(400,400,10,10,0);
COMMENT and punch a hole in the first rectangle;
DpyVct(275,275,10,10,LWMB);
RONUM;
```

Pla does not support defaulted arguments, so those have been filled in above, and DPYBIT has been replaced with DPYVCT with dX and dY set to 0.

In the example below we have defined NoLin and NoBox (as in GrnLib). The example in GrnLib/D uses two dimensional arrays which we have to implement in one dimension here.

```
REQUIRE grnPla.hdr[1,bil];
DEFINE  Theta=0,
        Phase=1,
        ArmLen=2,
        ArmOff=3,
        NumArms=6;
INTEGER ArrBounds;
PROCEDURE NoLin(ARRAY n; INTEGER dX,dY);
  BEGIN
  Style(n,LWMB);
  Line(n,dX,dY);
  Style(n,0);
  END;
PROCEDURE NoBox(ARRAY n; INTEGER dX,dY);
  BEGIN
  Style(n,LWMB);
  Box(n,dX,dY);
  Style(n,0);
  END;
ArrBounds←NumArms*ArmOff;
INTEGER ARRAY Arms[0:ArrBounds];
INTEGER i;
INTEGER ARRAY n[0:100];
InitID(n,0,0,511,511);
ClearScreen;
FOR i←0 STEP 1 UNTIL NumArms-1 DO
  BEGIN
  Arms[i*ArmOff+Theta]←(i+1)*2;
  Arms[i*ArmOff+ArmLen]←[13,8,5,3,2,1,1];
  Arms[i*ArmOff+ArmOff]←i MOD 4;
  END;
WHILE TRUE DO
  BEGIN
  REAL dX,dY;
  INTEGER k;
  IF CALL(0,"SNEAKS") THEN DONE;
  HopTo(n,125,250);
  IF Arms[Phase] MOD 360 = 0
    THEN
      FOR k←0 STEP 1 UNTIL NumArms-1 DO
        Arms[k*ArmOff+ArmOff]←((Arms[k*ArmOff+ArmOff]+1) MOD 4);
  FOR i←0 STEP 1 UNTIL NumArms-1 DO
    BEGIN
    dX←SIN(Arms[i*ArmOff+Phase]+90)*Arms[i*ArmOff+ArmLen]*8;
    dY←SIN(Arms[i*ArmOff+Phase])*Arms[i*ArmOff+ArmLen]*8;
    Arms[i*ArmOff+Phase]←Arms[i*ArmOff+Theta]+Arms[i*ArmOff+Phase];
    CASE Arms[i*ArmOff+ArmOff] OF
      BEGIN
      [0] NoLin(n,dX,dY);
      [1] Line(n,dX,dY);
      [2] Box(n,dX,dY);
      [3] NoBox(n,dX,dY);
      ELSE ; END;
    END;
  Show(n);
  Empty(n);
  END;
```

Flavors

A **Flavor** is yet another Pla type, this time borrowed from Lisp Machine Lisp (Zetalisp). A number of the more esoteric Zetalisp options have not been implemented, some because I can see no use for them, and some because the bugs they fix in Lisp do not exist in Pla. Interested readers are referred to the Lisp Machine Manual for further details. If some feature described there and not implemented here actually is useful, send me (BIL) a note.

A great deal has been written about "object oriented programming", "modularity", "contracts", "abstract types", and the infinite joys of doing things in the newest possible way, but all that seems irrelevant here. No grandiose claims are being made for Pla. Flavors exist in Pla because I think the facility they provide will prove useful to composers.

Introduction to Flavors

A flavor is a description of an object, much like a record_class declaration in SAIL. Each object of that flavor has its own local variables and can be the value of a flavor variable (something like a record and its associated record_pointer). Unlike a record, however, a flavor object has associated code which is invoked by "sending the object a message", to use the SmallTalk terminology. In Pla the message syntax uses the colon:

```
Obj:Msg(Args)
```

This statement invokes the code associated with MSG in the object OBJ, passing it the parameters ARGS. The code, sometimes called the method, is essentially defined as a procedure body where the procedure name is the name of the message used to invoke the procedure.

```
MESSAGE PrintName;
PRINT(SELF);
```

This little piece of code defines a message named "PrintName" which, when sent to an object, causes that object to print whatever it thinks its name is. **Self** is the name of the currently active object.

The descriptions that define objects can be combined in a variety of ways to form ever more complex and specialized types. The combinations are not constrained to be hierarchical (as they are in SmallTalk), and the flavor descriptions need not be complete. An incomplete flavor can be viewed as a partial description of a behavior that other flavors may want to be able to incorporate. New messages can be added and old messages can be redefined at any time, and all the instances of the modified flavor immediately know about the modifications. The entire flavor description can even be changed at any time, and within certain limitations, existing instances of that flavor immediately begin to behave as if they had always been defined according to the new definition.

The handle on a particular instance of a flavor, a flavor variable, is called a **FlavorVar** in Pla. FlavorVars can be local variables or procedure arguments, and procedures can return flavorvar values. Arrays of flavorvars are supported.

The Flavor Descriptor

A new flavor is defined as a set of local variables, messages, and a list of other flavors to be combined into the new flavor. A variety of flavor options control how the combined flavors are incorporated. In any case, all local variables of the included flavors become local variables of the new flavor. In the most straightforward case, if the new flavor defines a message, that definition overrides any definition of that message in the included flavors. All messages of the included flavors that are not defined in the new flavor are automatically incorporated into the new flavor. See below for exceptions and qualifications.

Flavor

The syntax of a flavor definition is:

```
FLAVOR name (options);
BEGIN
  local variables
  messages
  component flavor names
END;
```

If a flavor by the same name already exists, its definition is replaced by the new one. If the local variables of the old flavor do not exist in the new one, you can probably confuse Pla. The flavor options are, of course, optional. The current options are:

Gettable_Instance_Variables
Settable_Instance_Variables
Included_Flavors
Component_Flavors
No_Vanilla_Flavor
Method_Combination
Instance_Variables

The use of these options is discussed below.

The rest of the flavor definition is perhaps best defined by some examples.

```
                INTEGER Major,Minor;
                Major←'2512;
                Minor←'3122;
FLAVOR Modes;                    ∂ declare a flavor named "modes";
BEGIN
INTEGER Mode;                    ∂ which has one local variable;

  I_ONLY: Mode←Major;            ∂ "mode" is initialized to Major;

  MESSAGE Set_Mode(INTEGER i);   ∂ message to change the mode to i;
    Mode←i;

  MESSAGE Get_Mode;              ∂ message to get the value of mode;
    RETURN(Mode);
END;
```

This is a very simple flavor. It is somewhat equivalent to a SAIL record class description in that we have described a class of objects that contains an integer (Mode) with procedures for getting and setting the value of that integer. Instances of this flavor can also respond to the

message Describe which comes for free with all "vanilla" flavors. To anticipate a little, we can declare and create instances of this flavor as follows:

```
FLAVORVAR CurMode;            ∂ variable whose value is a flavor object;
CurMode←NEW_FLAVOR(Modes);∂ instantiate object of the Modes flavor;
PRINT(CurMode:Get_Mode);  ∂ prints value of CurMode's Mode variable;
```

If an I_Only message has been declared, it is executed (sent to Self) automatically when an instance of the flavor is instantiated. F_Only messages also exist and may even do something someday.

We can add messages to an existing flavor or modify existing methods with **DefMethod**. Since we did not specify any options, Modes is a "vanilla" flavor. Currently vanilla flavors get an extra message for free named "Describe" which returns a string containing a description of the object. We did not combine any flavors into the Modes flavor.

Flavor options

Because we often want to be able to get and set the values of the local variables of a flavor instance, two options exist to provide these messages automatically:

Gettable_Instance_Variables
Settable_Instance_Variables

Our Modes definition could have been:

```
FLAVOR Modes (Gettable_Instance_Variables
              Settable_Instance_Variables);
BEGIN
INTEGER Mode;
END;
```

In this case, the Set_Mode and Get_Mode messages are generated automatically by the flavor system. Gettable_Instance_Variables generates a message named "GET_"&var_name for each local variable. Settable_Instance_Variables generates a message named "SET_"&var_name for each variable. The SET_var message takes one argument: the new value for the local variable.

To turn off the generation of the vanilla flavor messages, use the option No_Vanilla_Flavor.

The remaining two options, Included_Flavors and Method_Combination, require some preliminary discourse. To avoid bringing our exposition of flavors to a halt, we will put off these discussions until later.

Flavor Instances and Flavor Variables

A flavor variable is a variable whose value is a flavor instance. Flavor variables are declared by the word **FlavorVar**. A flavor variable gets its values through assignment, just like any other variable. A new flavor instance is created by the procedure New_Flavor. The argument to New_Flavor is the name of the flavor type to be instantiated.

FlavorVar

FlavorVar *name;*

declares the flavor variable *name*. A flavor variable's value can be any type of flavor (similar to an any_class record_pointer in SAIL). A procedure can return a flavorvar as its value, and can take flavorvar arguments. Local Flavorvars are legal, as are flavorvar arrays. The following example shows some of the operations that can be performed on a flavorvar

```
FLAVOR Node {Gettable_instance_variables
             Settable_Instance_Variables};
BEGIN
FLAVORVAR Nxt,Lst;  ∂ to create a doubly linked list of flavorvars;
INTEGER i;          ∂ a key for each node in the list;
END;
FLAVORVAR First,Cur;
                    ∂ "First" is the top of the list;
First←NEW_FLAVOR(Node);
Cur←NEW_FLAVOR(Node);
                    ∂ create a couple instances of Node;
First:Set_Nxt(Cur); ∂ First:Nxt => Cur;
Cur:Set_Lst(First); ∂ Cur:Lst => First;
First:Set_i(1);     ∂ First:I = 1;
Cur:Set_i(2);       ∂ Cur:I = 2;
Cur←First;          ∂ now Cur = First;
PRINT(Cur:Get_i);   ∂ prints "1";
Cur←Cur:Get_Nxt;    ∂ Now Cur = First:Nxt;
PRINT(Cur:Get_i);   ∂ prints "2";
```

New_Flavor

New_Flavor(*Flavor-type*)

New_Flavor creates a new instance of the flavor *flavor-type*. If an I_Only message has been declared for the flavor, it is executed at this time. If an I_Only message is added later, already existing instances of the flavor are not re-initialized.

Messages and Methods

Message

Message *name (optional-arguments); Body;*

Message declares a message name that the flavor can respond to, specifies what arguments the message expects (if any), and gives the code to be executed when the message is received. This code is called the method, for no particular reason. In layman's terms, a message declaration declares a procedure which can be invoked by using the message name in conjunction with the name of an instance of the flavor. There are four special kinds of messages: **BeforeMessage**, **AfterMessage**, **I_Only**, and **F_Only**.

I_Only and F_Only

If an I_Only message has been declared for a flavor, it is executed when a new instance of that flavor is created (by New_Flavor). F_Only messages are currently ignored.

BeforeMessage and AfterMessage

Oftentimes a flavor merely wants to modify the action of an included flavor's message. Two special kinds of messages are provided for these cases: BeforeMessage and AfterMessage. The declaration of these messages looks just like a Message declaration, but the message behavior is different. When flavors are combined, all beforemessages are run, then the primary message, and finally all the aftermessages in reverse order. For example:

```
FLAVOR One;
BEGIN
MESSAGE msg(INTEGER i);      ∂ our main message;
  BEGIN
  PRINT("MAIN 1 ");
  RETURN(TRUE);
  END;
END;
FLAVOR Two;
BEGIN
  BEFOREMESSAGE msg(INTEGER i); PRINT("before 1 ");
  AFTERMESSAGE msg(INTEGER i);  PRINT("after 2 ");
  MESSAGE msg(INTEGER i);       PRINT("MAIN 2 ");
END;
FLAVOR Three;
BEGIN
  BEFOREMESSAGE msg(INTEGER i); PRINT("before 2 ");
  AFTERMESSAGE msg(INTEGER i);  PRINT("after 1 ");
  MESSAGE msg(INTEGER i);       PRINT("MAIN 3 ");
END;
FLAVOR Four;
BEGIN
(one two three);
END;
FLAVORVAR x;
x←NEW_FLAVOR(four);
x:msg(32);
```

Flavor four is thus a combination of the first three flavors. Because flavor Four has not specified differently (via Method_Combination), only the first "Msg" method is used (that of flavor One), but all the before and after messages are concatenated. The statement

```
x:msg(32);
```

therefore causes the following to be printed:

```
before 1 before 2 MAIN 1 after 1 after 2
```

We may not want to override the other primary messages. That is where method combination comes into Pla:

Method Combination

The Method_Combination flavor option determines how the messages of included flavors are combined to form a new composite message. The default combination method is the blank type (that is, it is not specified directly) which we have seen in action above. This combination method just uses the first method it finds and ignores all other primary methods. The search order is defined by the order in which the flavors are given in the flavor list, and by the order of the flavor lists of the included flavors. To quote the Lisp Machine documentation: If Flavor1's immediate components are Flavor2 and Flavor3, and Flavor2's components are Flavor4 and Flavor5, and Flavor3's component is Flavor4, then the complete list of the components of Flavor1 is

Flavor1 Flavor2 Flavor4 Flavor5 Flavor3

The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including non-terminal nodes before the trees they head [obviously], and eliminating duplicates: End-of-quote.

The other available combination methods are "Or", "And", "ProgN", and "List". The Or-combination executes each primary method until one returns true. The And-combination executes each until one returns false. ProgN-combination executes each method, then returns the value of the last one. Finally, List-combination executes each method and returns a list containing the values returned by the methods in order of execution.

Continuing with our example given above, we can define several new minimal flavors to illustrate each of these options:

```
FLAVOR Five ((method_combination (Or Msg)));
BEGIN (one two three) END;
```

If "Msg" is sent to an instance of this flavor, we get:

```
before 1 before 2 MAIN 1 after 1 after 2
```

The first Msg executed returns true (see Flavor One above), so only MAIN 1 gets printed out. And-combination gives:

```
before 1 before 2 MAIN 1 MAIN 2 after 1 after 2
```

Because the first Msg returns true, the second also gets executed. It does not return a value, which is equivalent to returning false, so the Msg of Flavor Three does not get executed. ProgN-combination gives:

```
before 1 before 2 MAIN 1 MAIN 2 MAIN 3 after 1 after 2
```

All Msg methods are executed in order. The value returned in this case is false. List-combination gives:

```
before 1 before 2 MAIN 1 MAIN 2 MAIN 3 after 1 after 2
```

as before, but the value returned is a list of the values returned by each method. The commonest use for method combinations is probably {ProgN L_Only} which insures that all initialization methods of all included flavors are executed when a new instance is created. The option Method_Combination should be placed in its own list. Each combination choice then follows as a list of the form:

```
{type msg-name}
```

For example:

```
{Method_Combination (ProgN I_Only) (Or Msg)}
```

Component_Flavors

You can specify the component flavors of a flavor as a part of the flavor options.

```
FLAVOR Combine {(Component_Flavors Flavor1 Flavor2)}; BEGIN END;
```

is equivalent to

```
FLAVOR Combine;
BEGIN
(Flavor1 Flavor2);
END;
```

but is a little clearer and more consistent with the rest of the flavor syntax.

Included_Flavors

We sometimes need to create a flavor that contains only a partial description of an object and that does not enforce any particular ordering on the flavors included in it. This flavor is then used by others who presumably do define the order of the flavors. To declare that a flavor needs access to variables and messages of other flavors, but does not want to specify an ordering that will confuse matters when it itself is included in a flavor, use the Included_Flavors option. This option is followed by a list of the names of the flavors to be included.

```
FLAVOR InclFlav {(Included_Flavors Flavor1 Flavor2)}; BEGIN END;
```

DefMethod

The messages that a particular flavor responds to need not be defined at the same time that the flavor is defined. **DefMethod** can be used both to add a new message to a flavor's repertoire, and to modify (redefine) an existing message. The syntax of Defmethod is:

DefMethod *flavor-name MsgWrd MsgName (args); Body;*

Flavor-name is the name of the flavor to which the new method is to be added. All local variables of that flavor are available to the method being defined. *MsgWrd* is either *Message*, *BeforeMessage*, or *AfterMessage*. *MsgName* is the name of the message being defined. If a message by that name already exists, DefMethod overwrites the old definition and all current instances of the flavor use the new definition rather than the old. The arguments are optional. *Body* is the method (procedure body) to be executed when an instance of *flavor-name* is sent *MsgWrd*.

If you are willing to use only DefMethod to define messages, you can also use the flavor option Instance_Variables to declare the flavor instance variables. Instance_Variables exists mostly to be more compatible with ZetaLisp.

```
FLAVOR Test ((Instance_Variables (Integer I J) (Real X)));
```

is equivalent to

```
FLAVOR Test;
BEGIN
INTEGER I,J;
REAL X;
...
```

except that the variables named cannot be accessed inside the body of the flavor. Instance variables declared this way and variables accessed from included_flavors are not available within the body of the flavor descriptor (the parser sees the body before it evaluates the options). Use Defmethod to get around this misfeature.

FlavorVars respond to the message MYMESSAGE by returning a list containing the names of all the messages which that FlavorVar knows.

Error Handling and Help

It is an error to send an object a message it does not know about already, but it is possible for a flavor to tell the flavor system that it wants to handle errors of this sort itself. If a message named "Handle_Unhandled_Messages" exists, that method is executed when this error occurs, rather than the system error handler. The message takes one string argument which is the name of the offending message. Lisp Machine lisp also has a default-handler flavor option which would be trivial to implement if anyone wants it.

A variety of help facilities are provided to aid debugging. **Help** *flavor-name* gives a list of the local variable names, message names, and included flavors for the given flavor. **Help** *flavor-name:message-name* describes the message referred to. If it is a normal message, the method is pretty-printed. If it is a combined method, a list of lists is displayed showing the names of the flavors searched for the Before, Primary, and After portions of the method. **Help** *Flavor-Var* prints the actual name of the flavorvar (its location in memory), the flavor it is an instance of, and the names and values of its local variables.

Just for the fun of it, we continue here with the very first example given, wherein we defined a flavor named "Modes". We now define a flavor which creates a voice to play a scale:

```
FLAVOR Scales;
BEGIN
INTEGER Octave;
  I_ONLY: Octave←3;
  MESSAGE Play_Scale(REAL Beg,Dur; INTEGER Mode);
    BEGIN
    VOICE Simp (Beg:(Beg+Dur));
      BEGIN
      INTEGER NoteN,MyOctave,MyMode;
      I_ONLY:   BEGIN
                MYBEGIN←beg;
                MYDUR←Dur;
                MyOctave←Octave;
                MyMode←Mode;
                END;
      P2←MYDUR/7;
      WHILE (MyMode LAND (1 LSH NoteN)) DO NoteN←NoteN+1;
      P3←16.351*2↑(MyOctave+NoteN/12);
      NoteN←NoteN+1;
      END;
    END;
  MESSAGE Handle_Unhandled_Message(STRING MsgName);
    BEGIN
    PRINT("Scale maker cannot respond adequately to "&MsgName&↓);
    END;
END;
```

Next we create an instance of this flavor and tell it to play a scale.

```
FLAVORVAR ScaleMaker;
ScaleMaker←NEW_FLAVOR(Scales);
ScaleMaker:Play_Scale(0,10,Major);
```

We now have a voice on the scheduler's queue ready to play a major scale from c3 to b3 in the Major mode. The two flavors used here do not make use of anything very fancy. In the next flavor, we combine the previous two, thus creating a flavor that knows about modes and about scales. We also want all the various I_Only blocks to run when a new instance is created, so we specify that the method combination scheme for I_only messages is to be "ProgN".

```
FLAVOR Scale_In_Mode { Gettable_Instance_Variables
                       Settable_Instance_Variables
                      (Method_Combination (Progn I_Only))};
BEGIN
(Modes Scales);
END;
```

A Scale_In_Mode object can now understand any of the Scales or Modes messages, and runs both I_Only blocks when initialized. We can, of course, add local variables and messages to a flavor as well as combining others in.

```
FLAVORVAR SetOctaveOfScale;
SetOctaveOfScale←NEW_FLAVOR(Scale_In_Mode);
SetOctaveOfScale:Set_Octave(4);
```

Now the octave to be used by the voice in the Play_Scales of the SetOctaveOfScale instance of the Scale_In_Mode flavor starts at middle c, rather than the default location at an octave below middle c. We decide, however, that we want SetOctaveOfScale and any other instances of Scale_in_Mode to use the mode it got from the Mode Flavor mixed in, without needing to be

told. So, we define a new message for this flavor, and immediately all instances "know" about the addition:

```
DEFMETHOD Scale_In_Mode MESSAGE Play_A_Scale;
  BEGIN
  SELF:Play_Scale(10,10,Mode);
  END;
SetOctaveOfScale:Play_A_Scale;
```

Sail Procedures

Pla provides a very versatile interface to SAIL procedures. It is unlikely that you will ever be forced to use SAIL due to some inadequacy in Pla, but the mechanism is provided in case the need ever arises. You can, in fact, write everything in SAIL, then use Pla purely as an interpretive SAIL environment. SAIL procedures can be precompiled and linked into Pla much as instruments are loaded into Sambox. The SAIL procedures can have any number of String, Real, Integer or Record_Pointer(Any_Class) arguments, and can return a String, Real, or ListVar value. In addition, all of Pla's innards can be mangled from SAIL through three entities:

```
RECORD_POINTER(ANY_CLASS) ARRAY PN[0:128]

INTEGER PROCEDURE CheckList(STRING Name;
                    REFERENCE RECORD_POINTER(ANY_CLASS) rp)

RECORD_POINTER(ANY_CLASS) PROCEDURE AddList(
                  STRING Name;
                  INTEGER Kind;
                  BOOLEAN UserCheck(TRUE))
```

All the present Pfield values are kept in the PN array. Addlist adds identifiers to Pla's own table of identifiers. Checklist returns the address of an identifier and its type. PlaTab.Def[Pla,Bil] contains the type definitions.

AddList takes two arguments: the name of the entity to be put on the table and its type. CheckList also takes two arguments. The first argument is the name of the entity to look for, and the second is a reference record_pointer which ends up pointing to the desired entity, if it exists. CheckList returns 0 if it can not find the desired entity; otherwise it reurns the type of the identifier.

Any SAIL procedure can be linked into Pla and can use these three entities to access whatever it likes. The following little procedure redefines a macro (equivalent to SAIL's Redefine):

```
PROCEDURE ReDefMac(STRING Name,NewBody);
BEGIN        comment redefine macro Name to be NewBody;
RECORD_POINTER(ANY_CLASS) Tmp;
IF CheckList(Name,Tmp)=#Mac
  THEN PRINT(Name," is not defined yet")
ELSE
  R:S[R:Nxt[Tmp]]←NewBody;
END;
```

ReDefMac uses CheckList to find the address of the macro *Name*. If it can find *Name* and *Name* is a macro, *ReDefMac* redefines the macro body to be *NewBody*. To return a value from a procedure, however, you must use:

Itemvar Procedures

An ITEM is a SAIL data type which can contain a DATUM which itself can have some SAIL type. An ITEMVAR is a variable containing an ITEM as its value. Because of the way linkages are handled in SAIL, to return a value from a SAIL procedure inside Pla without recompiling Pla, we must use an itemvar which contains that value. If we want to return an integer value from a SAIL procedure called from Pla:

```
P3←Int_Proc(freq);          ∂ Int_Proc is a SAIL procedure;
```

we must make that procedure an INTEGER ITEMVAR procedure, and put the value to be returned into an INTEGER ITEMVAR (8). REAL, STRING, INTEGER, and RECORD_POINTER(ANY_CLASS) ITEMVAR procedures are supported. An example itemvar procedure is:

```
INTEGER ITEMVAR PROCEDURE Int_Proc(REAL Freq);
BEGIN
INTEGER ITEMVAR rtnVal; comment returned value of Int_Proc;
RtnVal←GetItm;
DATUM(RtnVal)←Freq/220; comment put a new item in rtnval;
RETURN(RtnVal);
END;
```

This procedure can be called in Pla anywhere an integer is legal. It returns an integer.

```
Freq←220;
P3←Int_Proc(Freq);
```

assigns 1 to P3.

Once you have produced a file containing the desired procedures, you must add to that file the linking mechanism, then compile it as "MYPLA.REL" and load it with Pla ("LOA PLA.REL[NEW,MUS]/SAV"). Thereafter, your own version of Pla exists on your area, where it can be run by typing "RU Pla". Because it is a rather large program, it is probably best to delete the DMP file when you are through, and recreate it the next time it is needed.

The linkage mechanism to Pla is very similar to that of SAMBOX and SAMINS. The procedures must be compiled into a file named "MYPLA.REL". Unless you know what you are doing, it is best to make the source for MYPLA.REL begin as follows:

```
ENTRY;
BEGIN
REQUIRE "Pla.HDR[1,BIL]" source_file;
```

After this prologue, any number of SAIL procedures can follow. Finally, **DefProc** links the procedures into Pla.

DefProc

DefProc has three arguments:

```
DefProc(t,s,n)
```

DefProc links procedure *n* into Pla using the string *s* as its Pla-style representation (using "LISTVAR" in place of "RECORD_POINTER(ANY_CLASS)") where *n* returns a value of type *t* (no value = #NotAType). The type macros are defined in PlaTblSai[Pla,Bil] required by Pla.Hdr.

Suppose we have a network of nodes built from listvars and we want to crawl through that network one node at a time, doing something at each node. We can define a process to move to the next node:

---

(8) An integer itemvar is an itemvar containing an item whose datum is of integer type

```
RECURSIVE PROCEDURE Traverse(RECORD_POINTER(ANY_CLASS) Top;
    REFERENCE RECORD_POINTER(ANY_CLASS) NextNode);
BEGIN
IF Top=NULL_RECORD THEN RETURN;
IF VarPtr:T[Top]=#ListPtr
  THEN Traverse(VarPtr:Spf[Top],NextNode)
ELSE                  comment found one--point to it with NEXTNODE;
  BEGIN
  NextNode←Top;
  SUSPEND(MYPROC);  comment and wait for caller to resume;
  END;
Traverse(VarPtr:Nxt[Top],NextNode);
END;
```

#ListPtr signifies that we have not found an actual node. Traverse needs to ignore a #ListPtr node and go on. In the next procedure, we "flatten" the list (that is, we remove all internal structure). The *TransRec* procedure mentioned in *Flat* merely copies the node passed to it.

```
RECORD_POINTER(ANY_CLASS) PROCEDURE Flat(
    RECORD_POINTER(ANY_CLASS) Top);
BEGIN
RECORD_POINTER(ANY_CLASS) Base,Tmp,NxtNode;
ITEMVAR Nxt;
IF Top=NULL_RECORD
  THEN
    RETURN(NULL_RECORD);
SPROUT(Nxt←Getitm,Traverse(Top,NxtNode),PSTACK(4));
Base←TransRec(NxtNode);      comment copy the top node of the list;
Tmp←Base;
WHILE TRUE DO
  BEGIN
  RESUME(Nxt,HiThere,READYME);
  IF PSTATUS(Nxt)=TERMINATED THEN DONE;
  VarPtr:Nxt[Tmp]←TransRec(NxtNode);
  Tmp←VarPtr:Nxt[Tmp];
  END;
Relitm(Nxt);
RETURN(Base);
END;
```

Relitm releases the item used as the process item of *Traverse*. *PStatus* returns the current status of the process whose process item is passed to it. *Flat* resumes *Traverse* and puts itself in *ReadyMe* state so that it is resumed whenever *Traverse* suspends itself. A similar procedure rotates the elements of a listvar:

```
PROCEDURE Rotate(RECORD_POINTER(ANY_CLASS) Top);
BEGIN
RECORD_POINTER(ANY_CLASS) Ahead,Behind,SaveTop;
ITEMVAR first,second;
IF Top=NULL_RECORD THEN RETURN;
SPROUT(First←GetItm,Traverse(Top,Ahead),PSTACK(4));
SaveTop←TransRec(Ahead);
SPROUT(Second←Getitm,Traverse(Top,Behind),PSTACK(4));
RESUME(First,HiThere,READYME);
WHILE PSTATUS(First)≠TERMINATED AND PSTATUS(Second)≠TERMINATED DO
  BEGIN
  CopyRec(Behind,Ahead);
  RESUME(First,HiThere,READYME);
  RESUME(Second,HiThere,READYME);
  END;
IF Behind≠NULL_RECORD
  THEN
    CopyRec(Behind,SaveTop);
Relitm(First);
Relitm(Second);
END;
```

CopyRec copies the value of one node to another. The itemvar *HiThere* in these examples is just a dummy argument to *Resume*. We now link all the procedures given in this section:

```
PROCEDURE MakeProc;
BEGIN
DefProc(#ListVar,"ListVar Procedure Flatten(ListVar Top)",Flat);
DefProc(#NotAType,"Procedure Rotate(ListVar Top)",Rotate);
DefProc(#Integer,
        "Integer Procedure Int_Proc(Real Freq)",Int_Proc);
DefProc(#NotAType,
        "Procedure ReDefMac(String Name,NewBody)",ReDefMac);
END;
REQUIRE MakeProc INITIALIZATION;
```

We compile this file and load it into Pla, and the four procedures are immediately available for use. If we have a listvar L with the value, {1 2 (3 4) 5},

```
PRINT(Rotate(L))
```

prints "(5 1 2 (3 4))", and

```
PRINT(Flatten(L))
```

prints "(1 2 3 4 5)".

Sprout and Resume

You can also sprout and resume SAIL procedures from Pla. As an example, say we have defined a process named *Hi*:

```
PROCEDURE hi(STRING arg);
BEGIN
RECORD_POINTER(ANY_CLASS) Tmp;  ∂ communicate through lists;
RECORD_POINTER(ANY_CLASS) ITEMVAR It,Py;
It←GetItm;                      ∂ put an item in IT;
Tmp←NEW_RECORD(R);
R:T[Tmp]←#ListElement;          ∂ DO NOT say #RealType!;
R:X[Tmp]←3.14;
R:S[Tmp]←"3.14";                ∂ S is the printed value;
DATUM(It)←Tmp;                  ∂ IT is sent to Pla by Hi;
Py←RESUME(CALLER(MYPROC),It);
    ∂ Py is sent to Hi when Pla resumes it;
Tmp←DATUM(Py);                  ∂ R:Spf[Tmp] is top of list;
RESUME(CALLER(MYPROC),It);      ∂ CALLER(MYPROC) was Pla;
END;                            ∂ Pla terminates everything;
```

In Pla we can now create an instantiation of this process with:

```
IntVar ← SPROUT(Hi("There"));
```

The process_item is handled by Pla. The sprout defaults are PSTACK(4)←SUSPME, so the process has a little more stack space than the default, and Pla is suspended while the sprouted process runs. The process should RESUME Pla (CALLER(MYPROC)) when it is done. It can send Pla a list in the item in resume (IT above). Pla can then resume the process:

ListVar1 ← **Resume**(*IntVar*, *ListVar2*);

Sprout returns an integer which is used thereafter to call that particular instantiation of that process. The integer becomes the first argument to Resume. Resume's second argument is the list (record_pointer itemvar) you want communicated to the process being resumed. When Pla is once again resumed by the process, it receives *ListVar1*. The Resume defaults used in Pla are those of SAIL: Pla is suspended, and the resumed process runs. WrapUp, declared in Pla.Hdr, handles the resume and the translation from itemvars to record_pointers for you. WrapUp has two arguments, the itemvar of the procedure to resume (CALLER(MYPROC) in most cases), and the record_pointer you want sent to it. It returns a record_pointer to the calling process when it is resumed.

To increase stringstack space, set DEFSSS in the SAIL environment. To increase arithmetic stack space, set DEFPSS.

The following is an example of the use of sprout and resume. We want to sprout multiple versions of a composition program (a complex score-generating program) and remap its parameters to those of a new voice. The composition program assumes that various parameters mean various things, but the voice has things in a different order.

```
LISTVAR map1,map2,vals;          ∂ Maps tell COMP what to do;
INTEGER i,Spr1,Spr2;             ∂ SPRs keep track of sprouts;

DEFINE  ok=7,
        AllDone=6;

∂ (1 time color talea div tuning mode tempo form seed insNum);

map1←(1 60 64 64 12 1 t 1.75  a 1024  8);
map2←(1 60 64 64 12 1 t 1.0   a 1024  8);
```

```
PROCEDURE SetFields(LISTVAR LVal; INTEGER Num);
BEGIN            ∂ all voices use same parameter set-up;
P1←Nth(LVal,3);
P2←Nth(LVal,4);
P0←"Cel_One_Sum";
P3←Nth(LVal,5);
IF P3>2500 THEN P3←P3/4;
P4←(Nth(LVal,6)/8) MAX .002;
P5←NTH(LVal,7);
END;


SECTION One;
BEGIN
Spr1←Sprout(Comp(Ok));
VOICE Gull1;
  BEGIN
  Vals←RESUME(Spr1,Map1);
  IF NTH(Vals,1)=AllDone THEN KILL(SELF)
  ELSE SetFields(Vals,1);
  END;
END;

SECTION Two;
BEGIN
Spr2←Sprout(Comp(Ok));
VOICE Gull2;
  BEGIN
  Vals←RESUME(Spr2,Map2);
  IF NTH(Vals,1)=AllDone THEN KILL(SELF)
  ELSE SetFields(Vals,1);
  END;
END;


CALLSECTION One;
BEGIN
P5←P5*4;
P6←"RevSet: .1";
END;

CALLSECTION Two;
BEGIN
P5←P5*4;
P6←"RevSet: .1";
END;
```

Initialization

If the SAIL procedure !INIT! has been linked into Pla, it will be called automatically at initialization time. The procedure must not take any arguments. Do not use the SAIL Initialization feature—you will get ill mem refs.

Technical Information for SAIL linkages

For the complete use of SAIL, you need to know the exact nature of Pla's representation of the world. All variables are stored in VARLOC or VARPTR records where S is the string representation of the value, X is the real representation, and T is the type of the variable. When a name is passed to ADDLIST, it finds the variable's record on the hash table, and returns a pointer to it. To add an INTEGER to Pla named "ALIVE" with the value that Pla uses internally to indicate a running process, you can do the following in SAIL:

```
RECORD_POINTER(ANY_CLASS) Tmp;
Tmp←AddList("ALIVE",#Integer);
VarLoc:X[Tmp]←#Alive;
```

To change the value of a variable in Pla, get a pointer to it from CHECKLIST, then assign the new value to VARLOC:X[Ptr] and/or VARLOC:S[ptr]. It is even possible to change the variable's type dynamically. If the variable is a LISTVAR, SPF points to the top of the linked list of VARPTR class records that represent the list value of that LISTVAR. The list is structured as follows: In a given list, each element has both a string (VARPTR:S) and a real value (VARPTR:X). Its type (VARPTR:T) is either #ListElement (it is an atom, not another list), or #ListPtr (the list element is itself a list). The next element in the list is pointed to by VARPTR:NXT. If the given element is itself a list, the top of the list is merely a pointer to the list itself, which can be found at SPF.

The internal representation of (a b (c d)) is:

```
a-(nxt)-b-(nxt)-L
            |
          (Spf)-->c-(nxt)-d
```

The PN array is an array of record pointers. Pfields not in use are NULL_RECORDS. VARLOC:X and VARLOC:S are as above, but VARLOC:T is #Pn. These identifiers are defined in PlaTbl.Sai[Pla,Bil] which is required by Pla.Hdr in MyPla.Sai.

To pass a Pla variable by reference to a SAIL procedure (SAIL's APPLY mechanism doesn't make this automatic), pass the string name of the variable, then use CheckList to get a pointer to the variable's location. Remember that the variable's value is stored in R:X[PtrCheckListReturned] and R:S[SameLocation], unless it is a List variable (LISTVAR). R:SPF[Ptr] is the top of the listvar's list.

One last thing to note: to pass a string result back to Pla, you have to declare a global STRING ITEM, then assign that item to the STRING ITEMVAR used in the Pla-SAIL procedure. You cannot call GETITM in this case due to a bug in SAIL (uninitialized string itemvars give ill mem refs).

Differences Between SAIL and Pla

Pla is not SAIL! The following differences between the two languages are worth remembering:

1. Case statements: (Pla does not have Case expressions) The final branch of a Pla Case statement should end in a semicolon.

2. Block names (affecting Done, Continue, Begin, and End): Pla ignores Block names completely—it does not check that they match and does not notice when Done or Continue try to refer to an outer loop from within a nested loop.

3. Cprint takes the identifier returned by Open, rather than a channel number.

4. Pla does not support SAIL's FOR WHILE construct, nor the FOR statement where the control variable gets a list of values, as in

```
FOR i←1,3,5,6 DO
```

5. Pla's Sin function is SAIL's SIND function.

6. Pla's Ran function does not need an argument.

7. Pla's Scan function is actually SAIL's SCAN with following scan table:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789_!#$"
```

scanned in "XNR" mode.

8. Multiple assignments in Pla are evaluated from left to right (SAIL goes the other way).

9. The only legal macro delimiters in Pla are "⊂⊃".

10. See under Contexts for the differences between SAIL and Pla contexts.

Realtime Use of Pla

It is possible to drive the Samson box directly from Pla without creating a .PLA file or a .SAM file. Pla or SAIL procedures can control both the on-going composition and the real time inputs from the keyboard. The necessary support routines are found in RLTLIB.REL[RLT,MUS] and are described in some detail in RLTLIB.BIL[MUS,DOC] and SIXSYS.JAM[UP,DOC]/11P (SAM Messages).

The only portion you need be concerned with is the instrument definition file (an example is RLTINS.SAI[RLT,MUS]). The instrument code uses the LOWER assembler (see SAMLIB.BIL[MUS,DOC]/4P), so the code should be familiar to users of the Samson box. We do not use SAMLIB itself, however, because it has far more complication than we need. All the routines you need should be available from RLTPE.SAI[RLT,MUS]. Once you have defined how Pla talks to the patches (via the internal procedures GetNextNote and InitPatches), all you need to do is write normal Pla-code to define the voice behaviour and the way in which TTY: input is handled. Pla was not designed with real time interaction in mind, so certain "features" can get in the way. If anything galls you, send BIL a note.

As a suggestion, the following header exits straight to SAM, allows you to type "QUIT" to cause the composition to be exited, and otherwise handles everything you type as normal Pla-code:

```
EXIT Sam:Hi;

PROCEDURE Quit; BEGIN GivSAM; CALL(0,"EXIT"); END;

PROCEDURE Hi(STRING S);
BEGIN
EVAL(S)
END;

SetUserProc("hi");
```

After this header, you can include whatever is of interest. There should always be at least one voice (a mute perhaps) on the queue so that Pla does not decide the composition is complete prematurely.

An example MYPLA.SAI file can be found on [RLT,BIL]. PLA.REL is currently on MUSIC←[1,BIL]. MUSIC←PLA.DMP[RLT,BIL] has all the default files loaded in. You can run it with RAND[RLT,BIL] or SIMP[RLT,BIL]. I suggest SIMP, then type

```
SetPcRev(.25)
HiAtt←.05
SetRanAmp(.007)
Temp←2
```

These statements turn the reverb way up, give more noise than normal, set the attack time of the high notes to be .05 seconds, and slows the entire process down to half speed.

Pla Processing Order

I hope that Pla's actions will seem natural and unsurprising. There are cases however where it is useful to know exactly what order things occur in while Pla is receiving and processing commands.

Everything in Pla is done with "parse trees". The expression P3←3+X; is turned into a list:

```
      ←
    /   \
  P3     +
        / \
       3   X
```

The evaluator is given the top of this tree (the node containing the "←"), then recursively traverses the tree, handling each node according to what it finds. The assignment operator tells it to evaluate the right branch and assign the result to the thing found on the left side. So the evaluator travels down to the next node on the right side, the "+" sign, which it translates into "evaluate both branches and add the results", then hops back up with that result. The "3" node just returns its value, the "X" node returns the value of X (which we assume is a declared variable), and 3+X gets assigned to P3.

As Pla receives commands, it parses them into trees. Until Pla encounters a voice declaration (9), it simply evaluates the tree immediately and throws it away. Given the file:

```
Integer I;
I←3;
Real X;
X←I*2;
```

Pla makes room for an integer named I, assigns it the value 3, then makes room for X and assigns it 6.0.

Voices, Mutes, and Files (note lists) parse their associated statements, but do not evaluate them. A Section is treated as a recursive call of the entire Pla program, so it results in a stream of voice calls saved in a temporary file (named PLAnnn.TMP where nnn is the section number).

Once all input is complete, the list of voices is scanned for header statements. These are removed as they are encountered, evaluated, and passed on to the note list. Each voice in the list is evaluated once (the parsed tree is traversed causing all statements to be executed). If an L_Only block is found, it is evaluated and thrown away. F_Only blocks are not evaluated until later.

Once evaluation of the tree is complete, Pla assumes that the voice has now determined its begin time. The voice is inserted into the queue according to the begin time (lowest time is at the top of the list) if its status has not become #Dead and its notenum is greater than 0. Stopped voices are given a begin time of infinity, so they end up sitting at the back of the queue. During compilation of the note list, Pla is continually evaluating the tree of the voice at the top of the queue and reinserting the voice into the queue. Suppose we have a queue consisting of 5 voices:

Voice	A	B	C	D	E
BeginTime	3	5	20	100	infinity
Status	Alive	Alive	Waiting	Dead	Stopped

(9) Voice, Mute, Section, Always, Header, Finish, or File

Voice C is waiting for the termination of an opening rest ("VOICE C(20:100)" or something similar). Voice D has been Killed by someone, but hasn't percolated to the top of the queue, so Pla hasn't garbage collected him yet. Voice E has been Stopped by someone. Now Pla evaluates Voice A. Let's say that P2←3 on this note, so the new begin time is 6. Voice A is reinserted and our queue is:

Voice	B	A	C	D	E
BeginTime	5	6	20	100	infinity
Status	Alive	Alive	Waiting	Dead	Stopped

Voice B decides to Stop Voice A and Kill itself. Since its status is Dead, Pla does not reinsert it on the queue. If Voice B has an F_Only block, that block is evaluated as the last act of Voice B. The queue is now:

Voice	C	D	E	A
BeginTime	20	100	infinity	infinity
Status	Waiting	Dead	Stopped	Stopped

Now Pla changes Voice C to Alive, and evaluates its statements which we'll pretend set Voice C's begin time to 101.

Voice	D	C	E	A
BeginTime	100	101	infinity	infinity
Status	Dead	Alive	Stopped	Stopped

Voice D is discarded finally (since its status is Dead), Voice C is evaluated again (for brevity let's assume it's finished too), and our queue ends up as:

Voice	E	A
BeginTime	infinity	infinity
Status	Stopped	Stopped

Because the only voices left on the queue are stopped, Pla flushes the queue and finishes compilation. It evaluates the Finish block (if any), and closes all open files. F_Only blocks in Voice E and A are ignored because these voices never really finished.

When a voice occurs inside an expression (a "dynamic" voice), each instantiation of that voice gets a pointer to the template of the voice and a block of space for all its local variables. All instantiations of the voice talk to the same global variables (if any). To save the state of the world at the time of instantiation, save the desired variables in local variables in the I_Only block. The I_Only block is executed at the time of instantiation, but the template of the voice is not executed until the current voice has finished. For example, say we have a voice which creates another voice each time it creates a note, and the created voice needs to know the note its father was on at the time it was created.

```
VOICE Father;
BEGIN
P2←1;
P3←NOTE:a4,b,c;
VOICE Son;
  BEGIN
  REAL Dads_Pitch;
   I_ONLY: Dads_Pitch←P3;
   P2←3;
   P3←(1,2,3)×Dads_Pitch;
   END;
END;
```

Each note of the "Father" voice creates a string of notes in the Voice "Son" whose tune is transposed according to the "Father" voice.

Some further things to note: Mynotenum is set initially to infinity (2↑30 I guess). If a list ends with the word **End**, mynotenum is immediately reset to the lesser of its value and the number of elements in that list (see Page 24).

Access to a voice parameter via the subfield syntax (SIMP:P3 for example) refers to the array of parameters presently associated with SIMP. It does not cause anything to happen within SIMP.

Until a voice is on the queue (until it has been evaluated at least once), attempts to **Stop**, **Start**, or **Kill** it are ignored. If you want one voice to stop another voice at the beginning of time, put the voice to be stopped first in the command file (this insures that it is evaluated first). To refer to voices that have not yet been declared (a forward reference), put the voice name in quotation marks.

The **Finish** block (if any) is evaluated after all voices have completed execution, but before the word "FINISH" is placed in the play file. This lets you include a last directive for Sambox before it quits.

Some workarounds (these reflect current bugs in Pla):

To be sure a list is initialized to NIL, *ListVar*←QUOTE({}). Believe it or not, *ListVar*←{} is not sufficient in every case.

To make sure the "shift" and "window" settings work for a note list, handle them explicitly using **Kill** and **UnQueue**.

Debugging and Error Handling

Upon encountering an error, Pla prints an error message with a line feed where the error was noticed. There are several responses to the error: ?, E, <cr>, <alt>, QUIT, and *statement*.

"E" at this point exits directly to the editor to the file Pla thinks the error occurred in. If Pla doesn't know what file to go to it asks you for the file name. Once in the editor, •XRUN and •XRSYS return to Pla with the edited file name loaded automatically into Pla.

<cr> tries to continue compilation, after flushing the offending statement.

<Alt> exits to the monitor. Output buffers are not flushed, the output note list and temporary files are deleted, and all other files are closed.

Quit also exits to the monitor, but flushes output buffers and saves the output file. It is a cleaner, albeit slightly slower way to get out of Pla from the error handler.

Statement: The error handler is really the evaluator in disguise. Any expression typed to the error handler is parsed and evaluated. You can inquire about the state of the world, set variables, and so on. You have access to variables local to the immediately enclosing begin_end block. If you want more information about where the error occurred, type Help Eval_Tree, and the pretty-printer will print its idea of where in the program the error occurred. The notation used is LISP-like with user-declared entities in upper case.

Pla also has a modest automatic error correction routine that tries to detect and correct common spelling errors. You will be warned if any such corrections are made.

Halt

A break point can be set anywhere by including the word "HALT" with an optional string argument:

```
HALT("I is greater than 3");
```

The string is printed out when the Halt statement is executed. Upon halting, you are dropped into the error handler.

Esc-I

If Pla seems to be in an infinite loop, or you want to interrupt a compilation, type Esc-I (ESCAPE, then I). Pla flushes whatever it was doing and prompts for input. Pla interrupts are usually serviced at safe places in Pla, so the interrupt should rarely cause you to lose anything. The interrupt drops you into the error handler.

Help

To get a complete listing of all defined identifiers, type "HELP;". The names, types, and values are listed. To get information about any identifier, try typing "Help" followed by the identifier's name.

Help also knows about voice names, "QUEUE" (to print out what's on the queue), and perhaps more: don't be too surprised if a great deal of strange information comes pouring out. For example, say we have the following voice declaration in the file EX:

```
Voice simp(0:4);
  begin
  p2←1;
  p3←200+2;
  end;
```

We can run Pla and ask for help about SIMP:

```
.r Pla
File 1 → ex
Command file: EX[1,BIL]
File 2 → tty:
Command file: 1
>help simp;
        SIMP            SIMP (.000 : 4.00)
        Status: Alive
        Mail:()
        NoteNum:1073741824
(Begin
  (← P2  1.00     )
  (← P3  202      )
```

Pla provides various information about the state of the voice, then prints a LISP-like definition of the voice's body. The same kind of help is available for Pla procedures. Constant expressions are evaluated at parse time, as shown in the example above. If the voice status is "dead", the reason for the voice's demise is included in the Help. You can put F_ONLY:HALT; into a voice, then ask for help when the F_ONLY message is executed to find out why that voice is quitting.

Help Eval_Tree causes Pla to attempt to print out a representation of where the evaluator was just before Pla was stopped.

Help Stack prints out all the procedures currently on the SAIL stack (from the top down) and the values of all their parameters. Record_Pointers are printed as *Class_Name.Address*. If you type HELP *address*, Pla prints out the class name, field names, and field values of the record in question. HELP *number* assumes that *number* is a record address.

Help PtrValsToo sets a flag that tells HELP to henceforth print out the actual SAIL designation of the records traversed during HELP. You can poke around to your heart's content, following the pointers wherever they lead. The record pointer help is mainly a debugging aid for the developer-maintainer of Pla.

Queue Display

If you are mystified by the behaviour of the voice scheduler, try turning on the display of the scheduler's queue: SETINFO(9,TRUE). The topmost voice in the queue is currently running.

Pla Bugs

If you suspect that there is a bug in Pla and BIL isn't responding to gripes, and if you feel like messing with Pla's innards yourself, read page 2 of Pla.Sai[1,Bil] and have at it. The code does contain comments which may be of some assistance. Pla.Sai[Sys,Mus] cannot be compiled directly (it is an archival source). Once you've finished making changes, it might be a good idea to run the set of test files on [Pla,Bil]—these have been designed to check nearly every aspect of Pla. To run the tests, alias to [Pla,Bil], put the new version of Pla on [1,Bil] and BATCH/NOW/DO ●TEST.DO. Batch.Log[Pla,Bil] will have a complete record of what happened. Batch.Cor[Pla,Bil] has a correct version to compare it with. Please send BIL a note describing the bug and the changes made.

Index

Reserved Words

A	A above middle c
Abs	Absolute value
AF	a-flat
AfterMessage	Code that follows a message
Always	Execute a statement after every voice call
And	Boolean operator
Append	Append to a list
Array	Variable type declaration
Arrinfo	Array bounds information
AS	a-sharp
Ascii	Get arbitrary characters
Ask	Ask for file name
AtomP	Is argument an atom
B	B above middle c
BeforeMessage	Code that precedes a message
Begin	Compound statement
BF	b-flat
Box	Draw box
BoxTo	Draw a box
BS	b-sharp
C	Middle C
Call	Call a UUO
Caller	Voice Message function
CallSambox	Exit to Sambox
CallSection	Instantiate a section
Case	Case statement
CF	c-flat
ClearScreen	Clear the screen
Close	Close an output file
Comment	Ignore text up to semicolon
Common	instrument declaration (note list)
context	Variable type declaration
Continue	Loop control statement
CPrint	Write argument to a text file
CS	c-sharp
D	D above middle c
Define	Define a macro
DefMethod	Define a method for a flavor
DefSet	Define and preset macros
Delete	Remove element from list
DF	d-flat
Do	Do statement
Done	Loop control statement
Dont_Scan	Ignore text up to SCAN
DpyStr	Immediate text output
DpyVct	Immediate vector output
DS	d-sharp

E	E above middle c
EF	e-flat
Else	Conditional statement
Empty	Empty the graphics buffer
End	Compound statement close
ES	e-sharp
Eval	Evaluate a list or string
EvalNow	Evaluate now and replace with string result
Exit	Name output file
Expr_Type	Return type of expression
F_Only	Voice block executed as final voice act
F	F above middle c
False	Boolean constant
FF	f-flat
File	Open a note list
Finish	Block executed at end of compilation
Flavor	Abstract type descriptor
FlavorVar	Flavor variable
For	For statement
Forget	Remove variable from a Context
FS	f-sharp
Func	Load functions from file
FuncList	Create listvar from function data
Function	Declare function variable
G	G above middle c
Geq	Greater than or equal
Get	Get data from a file
GetInfo	Return Pla internal data
GF	g-flat
Global_Pfields	Make pfields global
GrnMap	unimplemented
GrnRd	Read a plot file
GrnWrt	Write a plot file
GS	g-sharp
Halt	Set breakpoint
Header	Put statements in note list header
Help	Request Help
Hop	Hop to a new point
HopTo	Hop to a new point
I_Only	Voice block executed at initialization time
IF	Conditional Statement
In_Context	Returns whether a variable is in a context
Include	Insert Note list
InitID	Initialize graphics buffer
Insert	Add element to list
Int	Truncate
IntDeb	Note list debug flag
Integer	Variable type declaration

Kill	Voice scheduling call
Land	Logical AND
Length	Length of list
Leq	Less than or equal
Line	Draw a line
LineTo	Draw a line
List	Score-like list notation
Listify	Put a list in a list
Listvar	A list variable
Lnot	complementation
Log	Log base e
Lop	Remove first character
Lor	Logical OR
LowerCase	Force string to lower case
LptGrn	unimplemented
Lsh	Logical shift
Lxor	Logical exclusive OR
MakeFunc	Create function from listvar data
Max	Return maximum of two args
Message	Voice message declaration
Min	Return minimum of two args
Mod	modulo
Mute	Voice with no output
MyBegin	Begin time of file or voice
MyDur	Voice Duration
MyEnd	Voice End Time
MyMessage	voice's message dictionary
MyNoteNum	Number of notes left to output
MyParNum	number of parameters to output
MyStatus	Status of voice
MyType	Type of voice
Neq	Not equal
New_Flavor	Create a new instance of a flavor
Next	Next element of a list
Not	Boolean operator
Note	Score-like Note notation
Nptix	number of processing elements (note list)
Nth	Return element of list
Null	Null String
NullP	Is argument a null list
Nutix	update ticks (note list)
Octal	Convert octal integer to string
Of	Part of Case statement
Omit	Note list editing
Open	Open an output file
Or	Boolean operator
Output	output file (note list)
P	Parameter array
Pars	Define parameter names
Pass	Same as Play
PField	Set maximum parameter number
PFinish	Note list header flag

Pi	3.14159265
Play	Note list Begin
Preset	Preset array elements
Print	Write argument to screen
procedure	Give a statement a name
Pty	Launch another job from Pla
QLen	Length of the queue
queue	The voice scheduler
Quote	Copy a list
Ran	white noise
RanF	Random numbers
RdNum	Get TTY input
Real	Variable type declaration
RealScan	Scan for reals
Record	same as Func
Require	Continue compilation at new file
Restore	Get variable value from a Context
resume	Resume a SAIL process
Return	Return value from procedure
Reverse	Reverse a list
Rhythm	Score-like rhythm notation
Rot	rotation
Save	Save current Pla state
Scan	Return word from string
Section	A group of voices
Seek	Random file access
Seg	Function definition
Self	Message handle
SetFormat	Set number printing precision
SetInfo	Change Pla internal data
Show	Display the buffer
Sin	Sine in radians
sprout	Sprout a SAIL process
SRate	sampling rate (note list)
Start	Voice scheduling call
Step	Part of For statement
Stop	Voice scheduling call
Store	Put variable into a Context
String	Declare string variable
Style	Set write style for buffer
Swap	Call some other program at end of Pla
Tempo	Tempo declaration
Text	Add text to graphics buffer
Then	Conditional statement
True	Boolean constant
TurtlX	Current display x-location
TurtlY	Current display y-location
TypeIt	Return type of identifier

UnQueue	**Remove voice from queue**
Until	**Part of For statement**
Value	**Return value of a variable in a context**
Variable	**Declare Real variable**
Voice	**Pla process**
WaitUntil	**Schedule voice by argument**
While	**While statement**