ABSTRACT

SAL is a new infix language for designing algorithmic compositions in Common Music (CM) [1], a composition environment implemented in Common Lisp [2] and Scheme [3]. SAL provides an easy-to-learn, command-style syntax that lets a composer learn algorithmic design without requiring that they first become familiar with Lisp notation. The name SAL stands for Simple Algorithmic Language -- or Secretly Another Lisp -- and is a hommage to Sal Martirano, a brilliant and influential composer who was a professor at the UIUC School of Music for many years. SAL incorporates many features of Lisp while avoiding the two most confusing aspects of Lisp programming style: prefix notation and the use of Quote to block evaluation. The result is a highly expressive language that is easy to learn. SAL also takes ideas from shell languages, and its (extendable) command set provides a consistent, clean way of interacting with the CM environment. SAL commands are typically easier (shorter, less arcane) to express than equivalent Lisp expressions. The SAL system consists of two software modules: a lexer/parser that is loaded into Common Music and an editing mode that provides syntax highlighting (code colorization) and evaluation services for designing and executing SAL programs. The separation between runtime and development components allows a single CM runtime to “serve” a number of concurrent SAL sessions. Using SAL the composer does not interact directly with Lisp and so does not need to learn anything about Lisp’s Read/Eval/Print loop (REPL) or how to recover from Lisp error breaks. Since SAL is parsed from its string representation it is able to report better (more helpful) program error messages back to the composer than can the underlying Lisp evaluator.

1. INTRODUCTION

Lisp is a powerful, dynamic and expressive language. It is also difficult for many people to learn! In order to understand the benefits of Lisp a student must first master its special (prefix) notation style and learn about an invisible process called Lisp evaluation. This is steep hill for anyone to climb. SAL is a language that replaces Lisp as a langage for algorithmic design while still leveraging Lisp’s powerful programming environment and interactive nature. While SAL replaces Lisp parenthetical syntax it still supports many of the expressive features of the Lisp runtime environment such as lists, keywords, and loop iteration. Above all, SAL avoids the two most difficult and arcane features of Lisp programming style: the use of prefix notation and expression quoting that blocks Lisp evaluation in the REPL. As a result the SAL language is expressive but easy to learn so composers are able to start designing musical algorithms almost immediately. This, in turn, allows the newcomer to learn about the power of algorithmic composition at a very early point in the learning curve, without having to first master Lisp syntax.

2. THE SAL SYSTEM

SAL consists of two software modules: a lexer/parser that is loaded into Common Music and an editing mode for developing SAL programs. The lexer/parser implements the language and the editing mode supports the design and execution of SAL programs. The fact that these modules are distinct and separate components means the runtime and development environments can be imbedded in separate software environments (Figure 1), each of which can be customized and optimized to support their specific purposes. This flexibility also allows a single CM runtime to “serve” a number of different SAL sessions running at the same time. In such a configuration, multiple users can be designing SAL programs on different computers but share a common algorithmic execution environment. Currently the SAL editing environment is implemented in X/Emacs [4]; a new cross-platform GUI environment in JUCE [5] is already under development. The Lisp side (lexer/parser) can run in any common Lisp and can communicate with the development environment(s) via several different methods: inter-process communication, socket reading/writing, and (in ECL Lisp [6]) with a direct foreign function interface.
3. THE SAL LANGUAGE

SAL borrows ideas from Lisp, Dylan [7], shell languages like bash [8], and the music languages SuperCollider [9], and PLA [10]. The essential characteristics of SAL are:

- All actions, including assignment are accomplished by executing statements.
- All statements begin with a command (reserved word).
- Statements may incorporate infix expressions that express math, logic, and function calling.
- Expressions sequences are delimited by "."

Several of these features are presented and compared to equivalent Lisp expressions in Figure 2.

SAL:
print "my random MIDI: ", 20 + random(60)

Scheme:
(begin (display "my random MIDI: ")
 (display (+ 20 (random 60)))
 (newline))

Common Lisp:
(format T "my random MIDI: ~S~%" (+ 20 (random 60)))

Figure 2. Equivalent statements in SAL, Scheme and Common Lisp. Each prints a text message and a random MIDI key number to the Console.

3.1. Expressions

SAL statements typically utilize expressions to accomplish their tasks. An expression consists of operators and operands combined using infix rules (operators between operands). An operand can be a number (integer, float, or ratio), variable reference var, an array or list reference arry[n], object slot reference obj.slot, a string within "", the booleans #t and #f, constant symbols :sym, lists {...}, function calls func(a,b) and conditional expressions #? (true, false).

Infix operators (Table 1) connect operands to form expressions.

<table>
<thead>
<tr>
<th>Math</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>% (mod)</th>
<th>^ (expt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relations</td>
<td>=</td>
<td>!=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Logic</td>
<td>!</td>
<td></td>
<td></td>
<td></td>
<td>&amp;</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Table of infix operators. "~" is general equality, i.e. true if its operands “look the same”. For example this relation of two lists: \{c e g\} =~ \{c e g\} is true.

Spaces or parentheses must delimit operators from operands. Commas delimit expression sequences used in function calls and some statements.

3.1.1. List notation

Lists of constant data are notated by grouping elements inside \{\} with spaces separating the individual elements. Lists can be nested to any level. For example, a list representing two chords could be notated:

print \{\{c e g\} \{d f a\}\}

Of course, Lisp lists can also be created using Lisp’s list function as well:

print list(1, 2 * 3, list(4, 5))

3.1.2. Function calls

A function call consists of the name of a function followed by its arguments inside \(\). Individual arguments within a call are separated by comma and may involve nested function calls. Some functions support named arguments, i.e. arguments whose values are preceded by a colonized name:

sprout make(<midi>, time: 33, keynum: 66)

All functions defined by Lisp are available in SAL. Users can add their own functions using the define command (See section 3.2.1).

3.2. Statements

A statement is a language construct that accomplishes some task. All statements start with a reserved name; some statements have additional (clausal) names associated with them. SAL provides general programming statements that implement command sequencing, iteration, assignment, as well as explicitly compositional statements that (generally) result in musical input or output occurring (open, sprout, plot, receive, etc.)

Any group of statements can be turned into a single statement (code block) by enclosing them within begin - end statements. Code blocks allow local variables to be defined using the with clausal:

begin
  with chord1 = \{c4 e g\},
  chord2 = transpose(chord, random(12))
  print chord1, “ and “, chord2
  sprout mymusic(chord1,chord2)
end

The remaining sections discuss several representative commands. Documentation of the complete SAL language is available in CM’s on-line documentation at http://commonmusic.sourceforge.net

3.2.1. Define

The define command adds top-level definitions to the runtime environment. Definition templates exist for creating global variables, functions and musical processes. Definitional forms for creating musical patterns, audio instruments and tuning systems will be added in the very near future. A global variable definition consists of the name of the variable followed
by an optional value clause introduced by =. If the variable does not have a value cause it is automatically set to boolean false.

**define** variable mytempo = 60

A function definition consists of the name of the function, a series of zero or more parameters enclosed within () and followed by a statement to execute when the function is called. The **return** command can be use to return a value back to the caller:

**define** function random-row (knum)
    return transpose(shuffle({0 1 2 3 4 5 6 7 8 9 10 11}), knum)

Musical process definitions are very similar to functions except that they contain **run** blocks instead of **return** statements. A **run** block is syntactically identical to the **loop** iteration statement (see below) except that it supports several additional statements specific to musical process definitions: **output** and **wait**. See Section 5 for an extended process example.

3.2.2. Assignment

The **set** command assigns values to locations: variables, array and list positions, and slots of objects. More than one location can be set in the same command using commas to delimit individual assignment clauses. Clauses proceed in left-to-right order so later assignments can depend on earlier ones:

**set** a = 60, b[2] = a * 3, c = list(a,b[2])

Each **set** clause consists of a location, operator and value (expression). The = operator performs full assignment, i.e. any value in location is replaced with the new value. The remaining operators affect an (existing) value in location and are provided to make common “assignment jobs” easy to accomplish. For example += adds value to the existing value in location, <= replaces the old value only if it is greater than value, and & appends value to the end of a list (Table 3).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>+=</td>
<td>increment</td>
</tr>
<tr>
<td>*=</td>
<td>scaling</td>
</tr>
<tr>
<td>&lt;=</td>
<td>minimization</td>
</tr>
<tr>
<td>=&gt;</td>
<td>maximization</td>
</tr>
<tr>
<td>&amp;</td>
<td>list collecting</td>
</tr>
<tr>
<td>@</td>
<td>list prepending</td>
</tr>
<tr>
<td>^</td>
<td>list appending</td>
</tr>
</tbody>
</table>

Table 3. Table of operators for the set command.

3.2.3. Iteration

SALs **loop** statement perform iteration. It is almost identical to Lisp’s powerful loop macro [11], except that (1) it executes SAL statements and (2) it is terminated by an **end**. Musical processes define an iteration over time using the **run** statement, which is syntactically identical to **loop**. Thus, if a composer can express iteration she can compose algorithmically. A complex example of a **run** iteration is included in the last section of this paper.

4. Lexing and Parsing

The purpose of SAL’s lexer/parser is to take a SAL language statement (string) and transform it into a valid Lisp program to be compiled and executed. The lexer is responsible for turning the string input into a stream of tokens for the parser to subsequently analyze. Part of the lexing process involves checking that the stream contains balanced input for () {} [] “” pairs, and that tokens are valid identifiers or constants (symbols, numbers, booleans and class variables). If an error is encountered, the lexer displays a contextual error message and points to the offending location in the input. This allows the composer to identify the source of the problem and the editor to position its cursor at the offending line in the editing buffer.

Once the lexer has tokenized and classified all the terms in the input the resulting token stream is passed to the parser. The parser determines if the contents are valid expressions and then emits “backend code” (Lisp expressions) for compilation and execution. The parser performs left to right pattern matching on the stream of tokens using declarative grammars, or rules that define legal syntax. More specifically, the SAL language consists of collection of grammars, each grammar contains a set of rules whose left side contains the rule’s “name” and whose right side contains a pattern that the token stream must match for the rule to be true, or matching. The pattern in the right hand side of the rule consists of a series of literals and rule names, each of which is designated as required, optional, zero-or-more or one-or-more in the matching. Elements that have no right hand expansion are terminals, i.e. they must match explicit token classes in the stream (numbers, identifiers, operator names, and so on). For example, Figure 1 shows three simplified rules for math expressions, where <= encloses rule names, <= marks the left and right hand side of a rule, {} is grouping, | is exclusive choice and {}* means zero or more of.

<expr> → <term> { <operator> <term>}*
<term> → { - <term>} | (! <term>) | (<expr>) | <funcall> | <identifier> | <number>
<operator> → + - * / ^ | & !
<funcall> → ...
...

To parse SAL input the parser matches the input stream against the combined rules of all the grammars; the first rule that matches “consumes” all its matched tokens and returns an object called a code-unit in place of those tokens in the stream. If the stream is successfully parsed the resulting collection of code-units is passed to an emit method that (recursively) translates units into valid
Lisp expressions. Certain emit methods also perform code analysis, for example registering variable declarations, examining slot names in object classes to insure the slots exits, determining if list data is constant or not, and so on.

Since the SAL language is declaratively defined it is straightforward to add new grammars to the parser in order to implement new features in the language. In addition, an ad hoc subset of SAL statements is bundled together to form a “top level command set”. Users can add their own “top level” simply by adding their rules to this collection.

5. SAL EDITING MODE

SAL-mode is an Emacs editing mode that supports designing and executing SAL programs. An equivalent application is currently being implemented in JUCE to provide a cross-platform, drag-and-drop GUI to Common Music. The SAL editing mode currently provides three basic services: (1) Syntax highlighting, or colorization of SAL program code based on its syntactic content, (2) menus for configuring the environment and controlling the Common Music runtime system, and (3) evaluation services for executing SAL programs. SAL is extremely easy to work with, and its Eval and Help commands are taken from the SuperCollider system. To evaluate a SAL command, the cursor is placed at the end of the expression and the keypad Enter key is pressed. Any terminal printout that occurs as a result of executing a command is sent to the SAL mode’s Console window. Contextual help is available by placing the cursor on a SAL command and pressing the Help key. The help text for the Command is then opened in the user’s web browser. The SAL menu can be used to start and stop a Common Music session, load optional software modules, play files, and so on. A special menu item Translate Expression provides an important pedagogical function: it translates a selected SAL expression into an equivalent Lisp expression. This is an easy and interactive tool for an interested composer to learn Lisp notation. For example, to learn prefix math notation, the user simply selects a SAL (infix) math expression in the buffer, chooses Translate Expression and then can study the equivalent Lisp expression displayed in the Console.

6. SAL PROGRAM EXAMPLE

The following code implements a complete SAL program that generates MIDI events to a file called ‘test.mid’. The algorithm takes two inputs, the number of events to create, and a list of possible rhythms, and generates a random melodic line that lies between the key numbers 40 and 70. If the current key number is an even number the algorithm adds a note one octave above it, else (the key number is odd) a note is added 1 octave below. In this case an additional tone is also added somewhere within the lower octave 40% of the time.

```
open "test.mid", tempo: 90

define process random-midis (num, rhys)
  run repeat num
    for k = between(40, 70)
    for r = pick(rhys)
      output make(<midi>, time: now(),
                   keynum: k)
      if even?(k) then
        output make(<midi>, time: now(),
                     keynum: k + 12)
      else
        begin
          output make(<midi>, time: now(),
                      keynum: k - 12)
          if odds(.4) then
            output make(<midi>, time: now(),
                         keynum: k - random(12))
        end
      end
    wait r
  end
sprout random-midis(20, {.1 .2 .4})
```

7. REFERENCES