SMURF Language Reference Manual

Serial MUsic Represented as Functions

Richard Townsend, Lianne Lairmore, Lindsay Neubauer, Van Bui, Kuangya Zhai
{rt2515, lel2143, lan2135, vb2363, kz2219}@columbia.edu

October 28, 2013

Contents

1 Introduction

SMURF is a functional language that allows a composer to create serialist music based on the twelve-tone composition technique. In general, serialism is a musical composition method where a set of values, chosen through some methodical progress, generates a sequence of musical elements. SMURF is based on the functional syntax and semantics set forth by Haskell. The backend of SMURF generates MIDIs corresponding to the composition defined by the user's initial program in SMURF.

2 Syntax Notation

The syntax notation used in this manual is as follows. Syntactic categories are indicated by *italic* type. Literal words and characters used in the SMURF language will be displayed in `typeset`. Alternatives are listed on separate lines.

Regular expression notations are used to specify grammar patterns in this manual. $r*$ means the pattern r may appear zero or more times, $r+$ means r may appear one or more times, and $r?$ means r may appear once or not at all. *r1*|*r2* denotes an option between two patterns, and *r1 r2* denotes *r1* followed by *r2*.

3 Lexical Conventions

SMURF programs are lexically composed of three elements: comments, tokens, and whitespace.

3.1 Comments

SMURF allows nested, multiline comments in addition to single line comments.

Comment Symbols	Description	Example
`/* */`	Multiline comments, nesting allowed	`/* This /* is all */ commented */`
`//`	Single-line comment	`// This is a comment`

3.2 Tokens

In SMURF, a token is a string of one or more characters that is significant as a group. SMURF has 6 kinds of tokens: *identifiers*, *keywords*, *constants*, *operators*, *separators* and *newlines*.

3.2.1 Identifiers

An identifier consists of a letter followed by other letters, digits and underscores. The letters are the ASCII characters `a-z` and `A-Z`. Digits are ASCII characters `0-9`. SMURF is case sensitive.

letter → [‘`a`’-‘`z`’ ‘`A`’-‘`Z`’]

digit → [‘`0`’-‘`9`’]

underscore → ‘`_`’

identifier → *letter* (*letter* | *digit* | *underscore*)*

3.2.2 Keywords

Keywords in SMURF are identifiers reserved by the language. Thus, they are not available for re-definition or overloading by users.

Keywords	Descriptions
`Bool`	Boolean data type
`Int`	Integer data type
`Note`	Atomic musical data type
`Beat`	Note duration data type
`Chord`	Data type equivalent to `[Note]` type
`System`	Data type equivalent to `[Chord]` type
`True, False`	Boolean constants
`let, in`	Allow local bindings in expressions
`if, then, else`	Specify conditional expression, else compulsory
`random`	Generate random numbers
`print`	Print information to standard output
`main`	Specify the value of a SMURF program

3.2.3 Constants

In SMURF, constants are expressions with a fixed value. Integer literals and Boolean keywords are the constants of SMURF.

digit → [‘`0`’-‘`9`’]

constant → -? [‘1’-‘9’] *digit**
0 *digit**
True
False

3.2.4 Operators

SMURF permits arithmetic, comparison, boolean, list, declaration, and row operations, all of which are carried out through the use of specific operators. The syntax and semantics of all of these operators are described in sections 5.3, 5.4, and 5.5, except for declaration operators, which are described in section 6.

3.2.5 Newlines

SMURF uses newlines to signify the end of a declaration, except when preceded by the \ token. In the latter case, the newline is ignored by the compiler (see example below). If no such token precedes a newline, then the compiler will treat the newline as a token being used to terminate a declaration.

3.2.6 Separators

separator → ,
&
\

Separators in SMURF are special tokens used to separate other tokens. Commas are used to separate elements in a list. The & symbol can be used in place of a newline. That is, the compiler will replace all & characters with newlines. The \ token, when followed by a newline token, may be used to splice two lines. E.g.

```
1 genAltChords (x:y:ys) = [(x,Time 4,1)]   \
2                         :[(y,Time 4,-1)]:genAltChords ys
```

is the same as

```
1 genAltChords (x:y:ys) = [(x,Time 4,1)]:[(y,Time 4,-1)]:genAltChords ys
```

3.3 Whitespace

Whitespace consists of any sequence of *blank* and *tab* characters. Whitespace is used to separate tokens and format programs. All whitespace is ignored by the SMURF compiler. As a result, indentations are not significant in SMURF.

4 Meaning of Identifiers

In SMURF, an identifier is either a keyword or a name for a variable or a function. The naming rules for identifiers are defined in section 3.2.1. This section outlines the use and possible types of non-keyword identifiers.

4.1 Purpose

4.1.1 Functions

Functions in SMURF enable users to structure programs in a more modular way. Each function has at least one argument and exactly one return value, whose types need to be explicitly defined by the programmer. The function describes how to produce the return value, given a certain set of arguments. SMURF is a side effect free language, which means that if provided with the same arguments, a function is guaranteed to return the same value.

4.1.2 Variables

In SMURF, a variable is an identifier that is bound to a constant value or to an expression. Any use of a variable within the scope of its definition refers to the value or expression to which the variable was bound. Each variable has a static type which can be automatically deduced by the SMURF compiler, or explicitly defined by users. The variables in SMURF are immutable.

4.2 Scope and Lifetime

The lexical scope of a top-level binding in a SMURF program is the whole program itself. As a result of this fact, a top-level binding can refer to any other top-level variable or function on its right-hand side, regardless of which bindings occur first in the program. Local bindings may also occur with the `let` *declarations* `in` *expression* construct, and the scope of a binding in *declarations* is *expression* and the right hand side of any other bindings in *declarations*. A variable or function is only visible within its scope. An identifier becomes invalid after the ending of its scope. E.g.

```
1 prime = [2,0,4,6,8,10,1,3,5,7,9,11]
2 main = let prime = [0,2,4,6,8,10,1,3,5,7,9,11]
3            p3 = (head prime) + 3
4        in print (p3)
```

In line 1, `prime` is bound to a list of integers in a top-level definition, so it has global scope. In line 2, the `main` identifier (a special keyword described in 6.4) is bound to a `let` expression. The `let` expression declares two local variables, `prime` and `p3`. In line 3, the `head` function looks for a definition of prime in the closest scope, and thus uses the binding in line 2. So the result to be printed in line 4 should be 3. After line 4, the locally defined `prime` and `p3` variables will be invalid and can't be accessed anymore.

4.3 Basic Types

There are three fundamental types in SMURF: `Int`, `Bool` and `Beat`.

- `Int`: integer type
- `Bool`: boolean type

- `Beat`: beat type, used to represent the duration of a note. A constant of type `Beat` is any power of 2 ranging from 1 to 16.

4.4 Structured Types

Structured types use special syntactic constructs and other types to describe new types. There are two structured types in SMURF: *list* types and *function* types.

A *list* type has the format `[`t`]` where t is a type that specifies the type of all elements of the list. Thus, all elements of a list of type `[`t`]` must themselves have type t. Note that t itself may be a list type.

A *function* type has the format t_1 `->` t_2 `->` ... `->` t_n `->` t_{ret} which specifies a function type that takes n arguments, where the kth argument has type t_k, and returns an expression of type t_{ret}. Any type may be used to define a function type, except for a function type itself. In other words, functions may not be passed as arguments to other functions, nor may a function return another function.

4.5 Derived Types

Besides the basic types, SMURF also has several derived types.

Expressions of type `Note` are used to represent musical notes in SMURF. The note type can be written as

`(Int,Int)$Beat`[`.`]*

The first expression of type `Int` must evaluate to an integer in the range from -1 to 11, representing a pitch class or a rest. When this expression evaluates to -1, the note is a rest, otherwise it represents the pitch class of the note. The second expression of type `Int` must evaluate to an integer in the range of 0-3, representing the register of the note, where the integer values and corresponding registers are given below.

- 1: Bass clef, B directly below middle C to first C below middle C
- 0: Bass clef, next lowest B to next lowest C
- 2: Treble clef, middle C to the first B above middle C
- 3: Treble clef, first C above middle C to next highest B

The expression of type `Beat` refers to the duration of the note, and may be followed by optional dots. The dot is a postfix operator described in section 5.3. Using this format, a quarter note on middle C could be written as `(0,2)$4`.

The `Chord` type is used to represent several notes to be played simultaneously. It is equivalent to the list type `[Note]`. The compiler will check to make sure all the notes in a chord have the same time duration.

The `System` type is used to represent a list of chords to be played sequentially. It is equivalent to the list type `[Chord]`.

5 Expressions

This section describes the syntax and semantics of *expressions* in SMURF. Expressions in SMURF use prefix, infix, or postfix operators. Unless otherwise stated, all infix and postfix operators are left-associative and all prefix operators are right-associative. Some examples of association are given below.

Expression	Association
`f x + g y - h z`	`((f x) + (g y)) - (h z)`
`f g x`	`((f g) x)`
`let { ... } in x + y`	`let { ... } in (x + y)`
`~ <> [0,1,2,3,4,5,6,7,8,9,10,11]`	`(~ (<> [0,1,2,3,4,5,6,7,8,9,10,11]))`

5.1 Description of Precedence in Expressions

Precedence describes the order of evaluation for subexpressions in a given expression. The order of precedence for expressions and their operators mirrors our ordering of the following subsections i.e. the first subsection describes the expressions with highest precedence, and the last subsection describes the expressions with lowest precedence.

5.2 Primary Expressions

primary-expr → *variable*
constant
`(` *expression* `)`
`[` (*expression*?|*expression* (`,` *expression*)*) `]`
`(` *expression*`,` *expression*`)$`*expression*

variable → *identifier*

5.2.1 Variables

A variable x is a primary expression whose type is the same as the type of x. When we evaluate a variable, we are actually evaluating the expression bound to the variable in the closest lexical scope.

5.2.2 Constants

An integer or boolean constant, as described in section 3.2.3, is a primary expression with type equivalent to the type of the constant.

5.2.3 Parenthesized Expression

An expression surrounded by parentheses is a primary expression.

5.2.4 Lists

A list is a primary expression. Lists can be written as:

`[`*expression*$_1$`, ..., `*expression*$_k$`]`

or

expression$_1$`:(`*expression*$_2$`:( ... (`*expression*$_k$`:[]))`

where $k >= 0$. These two lists are equivalent. The expressions in a list must all be of the same type. The empty list `[]` is typeless until it is applied to a non-empty list of some type `[`t`]`, at which time it is assigned the same type.

5.2.5 Notes

A note is a primary expression, and is written as a tuple of expressions of type `Int` followed by a `$` symbol and an expression of type `Beat`. The values of each of these expressions must follow the rules outlined in section 4.5.

5.3 Postfix Operator Expressions

postfix-expression $\rightarrow$ *beat-expr* `.`

The only expression in SMURF using a postfix operator is the partial augmentation of an expression of type `Beat`, which uses the dot operator. We say "partial augmentation" because a dot increases the durational value of the expression to which it is applied, but only by half of the durational value of that expression. That is, if *expr* is an expression of type `Beat` that evaluates to a duration of n, then *expr*`.` is a postfix expression of type `Beat` that evaluates to a duration of $n/2$. The dot operator may be applied until it represents an addition of a sixteenth note duration, after which no more dots may be applied. For instance, `4..` is legal, as this is equivalent to a quarter note duration plus an eighth note duration (the first dot) plus a sixteenth note duration (the second dot). However, `8..` is not legal, as the second dot implies that a thirty-second note duration should be added to the total duration of this expression. Our compiler will check the number of dots and return an error if too many are applied.

5.4 Prefix Operator Expressions

prefix-expression → *prefix-op expression*

Prefix Operator	Description	Example
∼	Tone row inversion	∼ `row` (returns the inversion of `row`)
`<>`	Tone row retrograde	`<>` `row` (returns the retrograde of `row`)
`!`	Logical negation	`if !(a == 5) then True else False`

SMURF has three prefix operators: logical negation, tone row inversion, and tone row retrograde. There is another row transformation operator, but it takes multiple arguments and is described in section 5.5.5. The logical negation operator can only be applied to expressions of type `Bool`, and the two row operators can only be applied to expressions of type `[Int]`. The compiler will check that all of the integers in a list are in the range $0-11$ if the list is passed to either of the tone row operators. All three operators return an expression of the same type as the expression the operator was applied to.

5.5 Binary Operator Expressions

binary-expression → *expression*$_1$ *binary-op expression*$_2$

The following categories of binary operators exist in SMURF, and are listed in order of decreasing precedence: list, arithmetic, comparison, boolean, tone row.

5.5.1 List operators

List Operator	Description	Example
`++`	List Concatenation	`[1,2,3] ++ [4,5,6]` (result is `[1,2,3,4,5,6]`)
`:`	List Construction	`1 : [2,3,4]` (result is `[1,2,3,4]`)

List operators are used to construct and concatenate lists. These two operators are `:` and `++`, respectively. The `:` operator has higher precedence than the `++` operator. Both of these operators are right-associative. List operators require that *expression*$_2$ be an expression of type `[`t`]`, where t is the type of *expression*$_1$. In other words, *expression*$_1$ must have the same type as the other elements in *expression*$_2$ when doing list construction. When doing list concatenation, both *expression*$_1$ and *expression*$_2$ must have type `[`t`]`, where t is some non-function type.

5.5.2 Arithmetic operators

Arithmetic Operator	Description	Example
`+`	Integer Addition	`a + 2`
`-`	Integer Subtraction	`5 - a`
`*`	Integer Multiplication	`5 * 10`
`/`	Integer Division	`4 / 2`
`%`	Integer Modulus, ignores negatives	`14 % 12`
`%+`	Pitch Class Addition (addition mod 12)	`14 %+ 2 == 4`
`%-`	Pitch Class Subtraction (subtraction mod 12)	`14 %- 2 == 0`
`$+`	Rhythmic Addition	`2 $+ 2 == 1`
`$-`	Rhythmic Subtraction	`1 $- 2 == 2`
`$*`	Rhythmic Augmentation	`8 $* 4 == 2`
`$/`	Rhythmic Diminution	`2 $/ 8 == 16`

There are three types of arithmetic operators (listed in descending order of precedence): basic, pitch class, and rhythmic. Basic arithmetic operators are those found in most programming languages like `+`, `-`, `*`, `/`, and `%`, which operate on expressions of type `Int`. It should be noted that the modulus operator ignores negatives e.g. `13 % 12` is equal to `-13 % 12` is equal to `1`. The pitch class operators are `%+` and `%-`. These can be read as mod 12 addition and mod 12 subtraction. They also operate on expressions of type `Int`, but the expressions must evaluate to values in the range $0 - 11$. The built-in mod 12 arithmetic serves for easy manipulation of pitch class integers. Lastly, there are rhythmic arithmetic operators (both operands must be of type `Beat`). These include `$+`, `$-`, `$*`, and `$/`.

Out of the basic arithmetic operators, `*`, `/`, and `%` have higher precedence than `+` and `-`. Both pitch class operators have the same precedence. Rhythmic arithmetic operators follow the same form as basic arithmetic operators, with `$*` and `$/` having higher precedence than `$+` and $-.

5.5.3 Comparison operators

Comparison Operator	Description	Example
`<`	Integer Less than	`if a < 5 then True else False`
`>`	Integer Greater than	`if a > 5 then True else False`
`<=`	Integer Less than or equal to	`if a <= 5 then True else False`
`>=`	Integer Greater than or equal to	`if a >= 5 then True else False`
`$<`	Rhythmic Less than	`4 $< 8 == False`
`$>`	Rhythmic Greater than	`4 $> 8 == True`
`$<=`	Rhythmic Less than or equal to	`4 $<= 4 == True`
`$>=`	Rhythmic Greater than or equal to	`1 $>= 16 == True`
`==`	Structural comparison	`if a == 5 then a = True else a = False`

SMURF allows comparison operations between expressions of type `Int` or `Beat`. Structural comparison, however, can be used to compare expressions of any type for equality. All of the comparison operators have the same precedence except for structural comparison, which has lower precedence than all of the other comparison operators.

5.5.4 Boolean operators

Boolean Operator	Description	Example
`&&`	Logical conjunction	`if b && c then True else False`
`||`	Logical disjunction	`if b || c then True else False`

Boolean operators are used to do boolean logic on expressions of type `Bool`. Logical conjunction has higher precedence than logical disjunction.

5.5.5 Tone row operators

The only binary tone row operator is the transposition operator, $\wedge\wedge$. $expression_1$ must be an expression that evaluates to a list of pitch classes, and $expression_2$ must have type `Int`. The result of this operation is a new tone row where each pitch class has been transposed up by n semitones, where n is the result of evaluating $expression_2$.

5.6 Conditional expressions

conditional-expression $\rightarrow$ `if` $expression_{boolean}$ `then` $expression_{true}$ `else` $expression_{false}$

When the value of $expression_{boolean}$ evaluates to true, $expression_{true}$ is evaluated, otherwise $expression_{false}$ is evaluated. $expression_{boolean}$ must have type `Bool`. Conditional expressions do not have newline restrictions.

5.7 Let Expressions

let-exp → `let` *decls+* `in` *expression*

Let expressions have the form `let` *decls* `in` *e*, where *decls* is a list of one or more declarations and *e* is an expression. The scope of these declarations is discussed in section 6.

The declarations in a let expression can be separated by an & symbol and also by a newline character. For example:

```
let x = 2 & y = 4 & z = 8
in x + y + z
```

The previous code is equivalent to the following:

```
let x = 2
    y = 4
    z = 8
in x + y + z
```

If the first code snippet were written without the & symbol and no newlines in between the declarations, a compile-time error would be raised.

5.8 Function application expressions

function-app-expression → *identifier expression+*

A function gets called by invoking its name and supplying any necessary arguments. Functions can only be called if they have been declared in the same scope where the call occurs, or in a higher scope. Functions may be called recursively. Function application associates from left to right. Parentheses can be used to change the precedence from the default. The following evaluates function *funct1* with argument *b* then evaluates function *funct2* with argument *a* and the result from evaluating (*funct1 b*):

funct2 a (funct1 b)

If the parentheses were not included, a compile-time error would be generated, as it would imply that *funct2* would be called with *a* as its first argument and *funct1* as its second argument, which is illegal based on the description of function types in section 4.4.

A function call may be used in the right-hand side of a binding just like any other expression. For example:

```
let a = double 10
in a
```

evaluates to 20, where `double` is a function that takes a single integer argument and returns that integer multiplied by two.

6 Declarations and Bindings

This section of the LRM describes the syntax and informal semantics of declarations in SMURF. A program in SMURF, at its top-most level, is a series of declarations separated by newline tokens. Declarations may also occur inside of `let` expressions (but still must be separated with newline tokens). The scoping of such declarations is described in this section. There are three types of declarations in SMURF: type signatures, definitions, and function declarations.

Declaration Operator	Description	Example
`::`	Type specification	`number :: Int`
`->`	Argument and function return type specification	`isPositiveNum :: Int -> Bool`
`=`	Variable or function binding	`x = 3`

6.1 Type Signatures

type-sig → *identifier* `::` (*type*|*function-type*) *newline*

function-type → *type* `->` *type* (`->` *type*)*

type → `Int`
`Bool`
`Beat`
`Note`
`Chord`
`System`
identifier
`[` *type* `]`

A type signature explicitly defines the type for a given identifier. The `::` operator can be read as "has type of." Only one type signature for a given identifier can exist in a given scope. That is, two different type signatures for a given identifier can exist, but they must be declared in different scopes. There are three categories of types in SMURF: basic, structured, and derived types; types are described in sections 4.3-4.5.

6.2 Definitions

definition → *identifier* = *expression newline*

A definition binds an identifier to an expression. All definitions at a given scope must be unique and can be mutually recursive. For example, the following is legal in SMURF:

```
let x = 4
   z = if y == 7 then x else y
   y = let x = 5
        in x + 3
in x + z + y
```

The x in the nested let statement is in a different scope than the x in the global let statement, so the two definitions do not conflict. z is able to refer to y even though y is defined after z in the program. In this example, the three identifiers x, y, and z in the global `let` will evaluate to values 4, 8, and 8, respectively, while the identifier x in the nested let statement will evaluate to 5.

A type signature may be given for a definition but is not required.

6.3 Function Declarations

fun-dec → *identifier args* = *expression newline* (*fun-dec*)*

args → *pattern*
pattern args

pattern → *pat*
pat `:` *pattern*
`[` *pattern-list*? `]`
`(` *pattern* `)`

pattern-list → *pat* (`,` *pat*)*

pat → *identifier*
constant
_

A function declaration defines an identifier as a function that takes some number of patterns as arguments and, based on which patterns are matched when the function is called, returns the result of a given expression. Essentially, a function declaration can be seen as a sequence of one or more bindings, where each binding associates an expression with the function identifier. The binding associated with the identifier depends on the patterns matched in the function call. There must be at least one pattern listed as an argument in a function declaration. All bindings defining a function must be contiguous and separated by newlines, and the number of patterns supplied in each binding must be the same.

As with definitions, only one set of bindings outlining a function declaration for a given name can exist in a given scope. However, the same function name can be declared multiple times if each instance is in a different scope.

If a function declaration for some identifier x occurs in scope n, then a type signature for x in scope $k >= n$ is required. That is if a function has been declared but its type has not been explicitly stated in the same or a higher scope, a compile-time error will be generated. The type of the arguments passed to a function are checked at compile-time as well, and an error is issued if they don't match the types specified in that function's type signature.

A *pattern* can be used in a function declaration to "match" against arguments passed to the function. The arguments are evaluated and the resultant values are matched against the patterns in the same order they were given to the function. If the pattern is a constant, the argument must be the same constant or evaluate to that constant value in order for a match to occur. If the pattern is an identifier, the argument's value is bound to that identifier in the scope of the function declaration where the pattern was used. If the pattern is the wildcard character '_', any argument will be matched and no binding will occur. If the pattern is structured, the argument must follow the same structure in order for a match to occur.

Below, we have defined an example function `f` that takes two arguments. The value of the function call is dependent on which patterns are matched. The patterns are checked against the arguments from top to bottom i.e. the first function declaration's patterns are checked, then if there isn't a match, the next set of patterns are checked, and so on. In this example, we first check if the second argument is the empty list (we disregard the first argument using the wildcard character), and return False if it is. Otherwise, we check if the second argument is composed of two elements, and, if so, the first element is bound to `x` and the second is bound to `y` in the expression to the right of the binding operator `=`, and that expression is evaluated and returned. If that match failed, we check if the first argument is zero and disregard the second. Finally, if none of the previous pattern sets matched, we bind the first argument to `m`, the head of the second argument to `x`, and the rest of the second argument to `rest`. Note we can do this as we already checked if the second argument was the empty list, and, since we did not match that pattern, we can assume there is at least one element in the list.

```
f :: Int -> [Int] -> Bool
f _ [] = False
f _ [x, y] = if x then True else False
f 0 _ = True
f x l = if x == (head x) then True else False
f m x:rest = f m rest
```

6.4 `main` Declaration

Every SMURF program must define the reserved identifier `main`. This identifier may only be used on the left-hand side of a top-level definition. The expression bound to `main` is evaluated and its value is the value of the SMURF program itself. That is, when a SMURF program is compiled and run, the expression bound to `main` is evaluated and the result is converted to our bytecode representation of a MIDI file. As a result, this expression must evaluate to a value of type `[]`, `Note`, `Chord`, `System`, or `[System]`. If a definition for `main` is not included in a SMURF program or if the expression bound to it does not have one of the types outlined above, a compile-time error will occur.

7 Library Functions

Below are the library functions that can be used in the SMURF language. These functions are implemented in SMURF but are available to all users for their convenience. Each library

function will include a description and its SMURF definition.

Head

The function `head` takes a list and returns the first element. This function is commonly used when working with lists. Although we do not currently support polymorphic typing in SMURF, the head function can nonetheless be thought of as a function that takes an expression of type `[`t`]` and returns an expression of type t, where t may be any non-function type.

```
head_note :: [a] -> a
head (h:tl) = h
```

Tail

The function `tail` takes a list and returns the end of the list. This function is commonly used when working with lists. The typing of this function is the same as the typing of `head`.

```
tail_note :: [a] -> [a]
tail (h:tl) = tl
```

MakeNotes

The function `makeNotes` takes in three lists and returns a list of notes. The first list consists of expressions of type `Int` representing pitches or rests. The second list consists of expressions of type `Int` representing the register that the pitch will be played in. The third list is a list of expressions of type `Beat` representing a set of durations. It is common in 12 tone serialism to use lists of notes. This function allows the user to easily turn their modified rows and columns into a list of notes to add to their system.

```
makeNotes :: [Int] -> [Int] -> [Beat] -> [Note]
makeNotes [] = []
makeNotes (h1:tl1) (h2:tl2) (h3:tl3) = (h1,h2)$h3:(makeNotes tl1 tl2 tl3)
```