SAMPL Reference Manual

Mike Haskel Mike Glass Morgan Rhodes
Navarun Jagatpal

March 5, 2007

1 Lexical Conventions

Tokens consist of identifiers, keywords, numbers, the '=' sign, and operators. Whitespace may include spaces, tabs, newlines, and carriage returns, and is ignored except in that it separates tokens.

1.1 Identifiers

Identifiers are strings of alphanumeric characters starting with an alphabetic character. They are case sensitive and may be of unlimited length.

1.2 Keywords

The keywords have special syntactic meaning and may not be used as identifiers:

```
be else end hz if let lfs rad sec start then until
```

1.3 Numbers

The convention for numbers is based upon the convention for floating constants in C. They consist of an integer part, a decimal point, a fraction part, an `e` or `E`, an exponent sign, and an integer exponent. The integer part, fraction part, and integer exponent each consist of a sequence of digits. The exponent sign consists of either a `+` or a `-`. Any part may be missing, so long as the integer exponent is present exactly when the `e` or `E` is, the exponent

sign is present only when the integer exponent is, the decimal point is present whenever both the integer and fraction parts are present, and at least one of the integer and fraction parts is present. Numbers denote `double` floating numbers in C via the usual translation.

2 Behavior of Generated Programs

The programs generated by the SAMPL compiler read and write audio data streams from standard input and output, respectively. The read and write operations occur synchronously in that there is a one-to-one correspondence between input and output samples points, which are strictly interleaved.

The audio itself is encoded as 16-bit two's-complement little-endian single-channel raw linear PCM data, sampled at 44.1 kHz. It may be possible to configure many of these parameters via command-line options to the generated programs—run the generated programs with the `--help` flag for details.

3 Expressions

Expressions in SAMPL have a type and a value. The type of an expression is implicit and does not change throughout the program's execution. The value of an expression may change as the program scans new input, but is otherwise static. Due to the possibility of infinite recursion, an expression's value may at any point be undefined, and the program may loop indefinitely when trying to evaluate it.

3.1 Types

An expression's type determines how it may be used, and informally specifies the interpretation of the expression's value. The various mechanisms for forming expressions determine the type of the resulting expression and impose restrictions on the types of their components. Type restrictions are checked at compile-time.

3.1.1 Base Types

The following are base types:

```
time intensity frequency angle scalar boolean
```

3.1.2 Functional Types

Given types a and b, $(a.b)$ is a type. These correspond to functions from type a to type b. Functions in multiple arguments are implemented as curried functions, that is, a function from a and b to c has type $(a.(b.c))$.

3.2 Values

Values of expressions of type scalar are a `double` in C. These values cannot be infinite or `nan`. Values of expressions of type boolean can either be true or false. The implementation of values in other base types is not defined. We assume the existence of a single canonical value for each; further behavior is specified with rules for how they interact with other values in following sections.

3.3 Unit Values

As discussed in the previous section, the values of expressions of base types (excepting scalar and boolean) are based upon some canonical unit value. The unit values are described here. One linear full-scale (`lfs`) is the maximum intensity of the input and output streams. Behavior is undefined upon a value greater than `1 lfs`. One second (`sec`) is a conceptual second as determined by the sample frequency. One radian (`rad`) is a conceptual radian—2π radians form a complete wave period. One Hertz (`hz`) represents the *angular* frequency of a 1 Hz wave. That is, a component with an angular frequency of one Hz has a period of 2π seconds.

3.3.1 Functional Values

Values of functional types are abstract. The value of an expression of type $(a.b)$ consists of a mechanism which, when provided a value of type a, produces a value of type b.

4 Definitions

*program : definition**
definition : let *name parameters* = *expression*

name : type ID
*parameters : (type ID)**

A program in SAMPL consists of a series of definitions. Each of these definitions binds a new name whose scope consists of the entire program, and optionally a collection of formal parameters whose scope consists of the body of the definition. Behavior is undefined when a definition attempts to bind an identifier twice or attempts to bind an identifier bound as a name elsewhere in the program.

There are two special names in SAMPL, `input` and `output`. Input is an expression of type intensity whose value is the intensity of the input signal at any particular point in the scanning proccess. The name `input` must not be bound elsewhere.

One of the program's definitions must bind the name `output` with no arguments as type intensity. After scanning a sample of input and setting the value of the `input` name accordingly, a program outputs the value of the `output` name. If the value of the `output` name is not well defined (i.e. infinite recursion), the program may hang indefinitely.

4.1 Interpretation of Definitions

A name bound by a definition may be used in an expression anywhere in the program.

Supposing identifiers in the parameter list to be expressions of their specified type, the expression on the right size of '=' (which may contain the parameter identifiers) must have the same type as the type specified in *name*. Given that the type specified in *name* is b and the types specified in the parameters are, from left to right, $t_1 \dots t_n$, the identifier in *name* is an expression which may be used anywhere in the program, and has type $(t_1.(t_2.(\dots(t_n.b)\dots)))$.

The identifier mentioned in *name* is an expression which may be used anywhere is the program. If it has no formal parameters, its value is the value of the expression on the right side of its definition. If it has a single formal parameter, then the value is a mechanism which, given a value of appropriate type, yields the value of the expression on the right side of the definition when taking the value of the parameter identifier to be the provided value.

If the definition has multiple formal parameters, we use the method of curried functions. That is, the value of the defined identifier is a mechanism

which is provided the value of the first parameter and returns a mechanism which is provided the value of the second parameter and returns another mechanism, et cetera. The method which is provided the value of the final parameter returns the value of the expression given that the parameter identifiers assume the appropriate provided values.

The need for type specifiers is a last-minute realization, and details are forthcoming.

5 Notation of Expressions

This section describes the various means of constructing expressions, their types, and their values.

5.1 Sequential Expressions

expression : sequential | *conditional*
sequential : be *(seq-continuation* | start *ID atom*)*
*seq-continuation : conditional (*until *conditional sequential)?*

An *expression a* which is a bare *conditional* is equivalent the *sequential* expression denoted `be` *a*.

A sequential expression describes a value which changes upon designated conditions. The structure consists of several instances of the keyword `be` followed by a value expression, separated by the keyword `until` followed by a transition expression. Note that a programmer may only assign a sequential expression directly to a name; sequential expressions do not nest inside other expressions.

A sequential expression may contain the keyword `start` followed by a tail specifier and a number of arguments. The tail specifier must be an identifier defined in the present scope and must not be a formal parameter. The number and types of the arguments must exactly match those of the formal parameters in the definition of the tail specifier. The use of the `start` keyword denotes that the value and transition expressions of the present sequential expression are considered conjoined with those in the definition of the tail specifier, after variable substitution. The definition of the tail specifier may in turn end with a tail specifier, and so on.

The type of all value expressions in a sequential expression must be the same—this is the type of the entire expression. The type of each of the

transition expressions must be boolean.

At any time, exactly one of the value expressions is considered current. The value of the expression is the value of the current value expression. If at any time the value of the transition expression following the current expression becomes true, the next value expression (after the current expression) which is followed by a false transition expression becomes the current expression. If no transition expression follows the current expression, the current expression remains current until the program terminates.

5.2 Conditional Expressions

conditional : logical | if *conditional* then *conditional* else *conditional* end

A conditional expression consists of a condition (following `if`), a positive part (following *then*), and a negative part (following *else*).

The condition must have boolean type. The positive and negative parts must have the same type—this is the type of the expression.

Whenever the value of the condition is true the value of the expression is the value of the positive part. Whenever the value of the condition is false the value of the expression is the value of the negative part.

5.3 Logical Operators

logical : comparison | *logical LOP comparison*

Logical operators consist of '|' (logical or) and '&' (logical and). Note that these have the same precedence, unlike C and most sane languages. The value of all operator expressions depends only on the values of their arguments.

The type of each argument to the operator must be boolean. The type of the entire expression is boolean.

The value a logical "and" expression is true if and only if the value of each of the arguments is true. The value of a logical "or" expression is true if and only if the value of at least one of the arguments is true.

5.4 Comparison Operators

comparison : additive | *comparison COP additive*

Comparison operators consist of '<' and '>'. Tests for equality are not provided as values should be treated as more or less continuous. Notions of

equality raise issues of discretization, and SAMPL guides the programmer intentionally away from these issues.

Both arguments to a comparison operator must share the same type. This type may not be boolean. The type of the resulting expression is boolean.

This reference manual does not specify the numerical details of expression values. The behavior of comparison and many other operators is defined in terms of properties they must satisfy.

The operators are opposites in that '$>$' expressions have the same value as '$<$' expressions with swapped argument values. '$>$' is transitive in that, given $a > b$ and $b > c$, $a > c$. Never do both $a > b$ and $a < b$ have true value. Both $a > b$ and $a < b$ have false value if and only if a has the same value as b.

Scalar values compare as `double` values in C.

5.5 Additive Operators

additive : multiplicative | additive AOP multiplicative

Additive operators consist of '$+$' and '$-$'.

Both arguments to an additive operator must share the same type. This type may not be boolean or frequency. The resulting expression has the same type as the arguments.

Excepting rounding errors, addition and subtraction form an abelian group. This is equivalent to the following conditions: addition is associative and commutative. The value written `0` *UNIT*, written here 0, is such that $a + 0$ has the same value as a for all expressions a. For all expressions a, $a - a$ has the same value as the additive identity. $a - b$ is equivalent to $a + (0 - b)$.

5.6 Multiplicative Operators

multiplicative : functional | multiplicative MOP functional

Multiplicative operators consist of '$*$' and '$/$'.

If one argument to '$*$' has type scalar, the other argument may be of any type other than boolean. The resulting expression has this as its type. Otherwise, one argument must have type frequency and the other time. The resulting expression has type angle.

Scalars multiply as `double` do in C. Excepting rounding errors, scalars multiply with other types as a vector spaces, equivalent to the following

conditions. If a and b have scalar type and x has any type other than boolean and scalar, $a * (b * x)$ has the same value as $(a * b) * x$. If c has scalar type and value zero, $c * x$ has the value of the additive identity. Similarly, if c has value one, $c * x$ has the same value as x. If c has value negative one, $c * x$ has the same value as $d - x$, where d is the additive identity of the same type as x. If a is of scalar type and x and y are of the same type, neither boolean nor scalar, $a * (x + y)$ has the same value as $a * x + a * y$. Finally, if $a * x$ has the same value as $x * a$.

If x has frequency type and y has time type, $x * y$ has the same value as $y * x$. Also, if a has scalar type, $a * (x * y)$ has the same value as $(a * x) * y$ and $(a * y) * x$. Furthermore, `1 hz * 1 s` has the same value as `1 rad`.

Expressions which occur as the denominator of a '/' operator may only have types scalar, time, and frequency. If the denominator is a scalar, the numerator may have any type other than boolean, and the resulting expression has the same type as the numerator. If the denominator is either time or frequency, the numerator must have type angle, and the resulting expression has type either frequency or time, whichever is not the type of the denominator.

If the denominator is a scalar, the resulting expression has the same value as though the numerator were multiplied by the reciprocal (as a `double` in C) of the denominator. Otherwise, if the denominator is either of type frequency or time, the value of the resulting expression is such that, when multiplied by the denominator, the result would have the same value of the numerator.

5.7 Function Application

functional : atom | functional atom

Functional expressions consist of a function followed by an argument.

The type of the function must be $a.b$ for some types a and b. The argument must have type a. The resulting expression has type b.

As mentioned in 3.3.1, the value of the function consists of the name of a free variable and an result expression. The value of the function application is the value of the result expression when all occurrences of the identifier named by the free variable have been substituted with the argument.

5.8 Atoms

atom : constant | LPAREN conditional RPAREN

LPAREN consists of a single '(' character, while *RPAREN* consists of a single ')' character. The and type of a parenthesized atom are the same as of the enclosed *conditional* expression.

5.8.1 Constants

constant : true | false | *numerical*
numerical : NUMBER unit?
unit : lfs | sec | rad | hz

Constants may be `true` or `false`, which have boolean type and correspond to the appropriate values, or they may be numerical constants.

Numerical constants are expressions consisting of a number followed by an optional unit specifier. The type of a constant is determined by its unit specifier—`lfs` (*linear full-scale*) corresponds with intensity, `sec` with time, `rad` with angle, and *hz* with frequency. Numbers with no type specifier are scalars.

The value of a scalar constant is simply the value of the `double` value indicated by *NUMBER*. The value of other numerical constants is the same as the value of the scalar indicated by *NUMBER* multiplied by the unit value of appropriate type.