nc Reference Manual

Phillip Ames
COMS W4115
pa2354@columbia.edu

June 27, 2012

Contents

1 Introduction

nc is a programming language designed to simplify the task of performing computations on musical elements. Music and musical notation is inherently quite mathematical, and the syntax and semantics of *nc* reflect this property.

The language *nc* describes is designed to be powerful enough to implement a rich standard library, suitable for use by programmers with basic computer science skills and a desire to solve problems in the musical domain.

2 Lexical Conventions

Programs written in *nc* can contain the following types of tokens:

- constants
- identifiers
- keywords
- operators
- punctuation

A brief description of each follows.

2.1 Constants

nc recognizes the following types of constants:

2.1.1 String constants

Strings in *nc* begin with a double quote and consist of all subsequent characters until another double quote is found. To support the possibility of strings containing a literal double quote, *nc* will ignore a double quote preceded by a backslash character when determining whether the end of the string constant has been reached. Strings cannot be assigned to variables, and can only used with the built-in print function.

2.2 Literals

The recognized literals also represent the different variable types supported by *nc*

2.2.1 Boolean literals

Booleans are represented by the keywords `true` or `false`.

2.2.2 Integer literals

Integers consist of sequences of digits, from 0 to 9. *nc* only supports base-10 representations of integers, i.e. no octal or hexadecimal support as in C. Additionally, *nc* does not support floating point numbers or their representations.

2.2.3 Pitch literals

A pitch consists of a note (selected from the character set {A, B, C, D, E, F, G}), an optional octave specifier (selected from the integer range 0-8), and an optional accidental specifier (from the character set {b, #}). If unspecified, the octave defaults to 4 and no accidental is applied to the pitch.

2.2.4 List literals

The format of a list literal varies depending on whether an empty list is desired. For empty lists, the following syntax must be used:

identifer `:=` `list`<*type*>

Where *type* is one of `bool`, `int` or `note`. Declarations of non-empty lists will have types properly inferred. List elements should be enclosed by brackets and separated by semicolons, e.g. the following statement declares and initializes a list of integers named foo:

foo `:= [1; 2; 3];`

Lists of lists are not supported. Lists are immutable once defined. Operators such as the list concatenation operator will return a new list.

2.3 Identifiers

Identifiers are sequences of letters and digits and the underscore. The first character must be alphabetic, however if it is uppercase, it cannot be in the set {A, B, C, D, E, F, G} to distinguish between identifiers and pitch literals.

2.4 Keywords

The following strings are reserved for use as keywords and may not be used otherwise:

```
bool        down
else       false
for         half
if            in
int          len
list        note
octave     pitch
print     return
reverse       to
true          up
void       while
whole
```

2.5 Operators

The following operators are supported:

```
!  + - / * % = < >
+= -= <= >= << >> [] && || ::
```

The meaning of each is discussed in section 4.

2.6 Punctuation

The following have syntactic meaning within expressions, statements, and declarations:

;	Statement terminator and list element separator
,	Argument separator
()	Permits grouping of expression statements
{ }	Used to provide compound statements and enclose scoping blocks

3 Meaning of Identifiers

3.1 Types

The type of variables in *nc* is inferred when they are declared using the `:=` operator, with the exception of declaring an empty list - the declaration must be annotated to reflect the type of values that will be placed in the list, using the syntax described in section 2.2.

3.2 Declaration

3.2.1 Variable Declarations

Variables are referred to by identifiers. Variable declarations have the following syntax:

identifier `:=` *expression*

Where *identifier* matches the format described in section 2.3. The result of evaluating *expression* is stored in the newly created identifier.

3.2.2 Function Declarations

Functions are declared using the following syntax:

return-type identifier`(`*formal-list*`)`

Function bodies are lists of statements enclosed in braces which immediately follow the declaration. *return-type* should be one of `int`, `note`, `void`, or follow the form of an empty list declaration. *formal-list* is a comma separated sequence of 0 or more types and identifiers. A sample declaration of a function named foo which returns a note and takes a note and list of integers as arguments is:

`note` foo`(note` n, `list<int>` l`)`

The special type `void` can be used to signify a function which does not return a value.

3.3 Scoping Rules

nc has the following notion of variable scope:

Local: An identifier declared in a block (defined as the statements between a pair of { and }) is only visible after its declaration within that block, and within child blocks. If an identifier is declared with the same name as an existing identifier from an outer block or global scope, the outer declarations are shadowed by the inner declaration. It is an error to declare an identifier multiple times with the same name within the same block. Scope environments from calling functions are not consulted when resolving a variable reference.

Global: Identifiers declared outside of blocks have global scope. It is an error to declare an identifier with the same name multiple times in global scope.

4 Expressions and Operators

4.1 Precedence and Associativity Rules

Unless otherwise specified, all binary operators are left associative, and all unary operators are right associative. Operator precedence, from high to low is:

operator	description
`-`	unary minus
`!`	unary logical NOT
`* / %`	multiplication, division, modulo
`+ -`	addition, subtraction
`<< >>`	octave shifting
`< <= > >= == !=`	relational operators
`&& ||`	logical comparators
`= += -=`	assignment, addition/subtraction and assignment

4.2 Unary Operators

4.2.1 `!`*bool-expression*

The logical not operator can be applied to an expression with a boolean type. The resulting value is `true` if *expression* is `false`, and `false` if *expression* evaluates to `true`.

4.2.2 `-`*int-expression*

The unary minus operator can be applied to expressions with an integer type. This is used to convert negative integers to positive integers and vice-versa.

4.3 Binary Arithmetic Operators

4.3.1 *int-expression* `+` *int-expression*

The binary `+` operator indicates addition. When used with two expressions of integer type, arithmetic addition is used to compute the new value.

4.3.2 *note-expression* `+` *interval*

When the binary `+` operator is invoked with a left-hand argument of a note type, the right hand argument must be one of the language keywords **whole** or **half** which describe the interval to add to the note. Adding an interval which would go beyond the note C8 (the highest key on a standard piano) is undefined.

4.3.3 *int-expression* `-` *int-expression*

The binary `-` operator indicates subtraction. When used with two expressions of integer type, arithmetic subtraction is used to compute the new value.

4.3.4 *note-expression* - *interval*

When the binary - operator is invoked with a left-hand argument of a note type, the right hand argument must be one of the language keywords **whole** or **half** which describe the interval to subtract from the note. Subtracting an interval which would go beyond the note A0 (the lowest key on a standard piano) is undefined.

4.3.5 *int-expression* * *int-expression*

The binary * operator indicates multiplication. The type of each expression must be integer.

4.3.6 *int-expression* / *int-expression*

The binary / operator indicates division. The type of each expression must be integer. If the result would be fractional, it is rounded down to the nearest integer value.

4.3.7 *int-expression* % *int-expression*

The binary % operator indicates modulo division. The type of each expression must be integer. The result is the remainder from dividing the first expression by the second.

4.4 Logical Operators

4.4.1 *bool-expression* && *bool-expression*

The binary && operator performs a logical AND operation on the provided bool-expressions, returning `true` if both are `true`, and `false` otherwise.

4.4.2 *bool-expression* || *bool-expression*

The binary || operator performs a logical OR operation on the provided bool-expressions, returning **true** if either is true **true**, and **false** otherwise.

4.5 Shift Operators

4.5.1 *note-expression* << *int-expression*

The binary << operator shifts the value of the note down by the number of octaves specified by the right hand side. Shifting below the note A0 is undefined.

4.5.2 *note-expression* >> *int-expression*

The binary >> operator shifts the value of the note up by the number of octaves specified by the right hand side. Shifting above the note C8 is undefined.

4.6 Comparison Operators

For all comparison operators, the two provided expressions must both be int- or note-expressions. In the case of int-expressions, arithmetic comparison is performed. In the case of note-expressions, the comparison is based on the position of the notes on a standard piano.

4.6.1 *expression* < *expression*

The binary < operator compares the two provided expressions (which must both be int- or note-expressions) and returns `true` if the left hand side is less than the right hand side.

4.6.2 *expression* `>` *expression*

The binary `>` operator compares the two provided expressions (which must both be int- or note-expressions) and returns `true` if the left hand side is greater than the right hand side.

4.6.3 *expression* `<=` *expression*

The binary `<=` operator compares the two provided expressions (which must both be int- or note-expressions) and returns `true` if the left hand side is less than or equal to the right hand side.

4.6.4 *expression* `>=` *expression*

The binary `>=` operator compares the two provided expressions (which must both be int- or note-expressions) and returns `true` if the left hand side is greater than or equal to the right hand side.

4.6.5 *expression* `==` *expression*

The binary `==` operator compares the two provided expressions (which must both be int- or note-expressions) and returns `true` if the left hand side is equal to the right hand side.

4.7 Assignment Operators

These operators are right associative.

4.7.1 *identifier* `:=` *expression*

The binary `:=` operator serves the dual function of both declaring a variable within a defined scope (see section 3.3), and assigning it the value and type of *expression*.

4.7.2 *identifier* `=` *expression*

The binary `=` operator assigns the value of *expression* to *identifier*.

4.7.3 *identifier* `+=` *expression*

The binary `+=` operator is a convenience operator designed to evaluate the addition of the current value of *identifier* with *expression* and store the result in *identifier*. The type of *identifier* and *expression* must be compatible with the rules specified for the binary `+` operator.

4.7.4 *identifier* `-=` *expression*

The binary `-=` operator is a convenience operator designed to evaluate the subtraction of *expression* from the current value of *identifier* and store the result in *identifier*. The type of *identifier* and *expression* must be compatible with the rules specified for the binary `-` operator.

4.8 List Operators

4.8.1 *expression* `::` ***list-expression***

The binary `::` operator prepends the result of *expression* to the list in *list-expression*.

4.8.2 ***list-expression*`[`*int-expression*`]`

The binary `[]` operator returns the member *int-expression* positions away from the beginning of `list-expression`. Accessing an element outside of the range [0, len(*list-expression*)) is undefined.

5 Statements

Unless otherwise specified, statements are executed in the order they are present within a function body.

5.1 Expression Statement

Expression statements can be built using the operators described described in section 4. Additionally, an expression statement can be a function call, which has the syntax:

function-identifier`(`*argument-list*`)`

function-identifier must have been declared as outlined in section 3.2.2 or be a built-in function described in section 6. *argument-list* can consist of 0 or more comma separated expressions.

5.2 Compound Statement

Compound statements allow multiple statements to be used where a single statement is expected. Compound statements are formed by including an opening brace ({), followed by any number of statements, terminating with a closing brace (}).

5.3 Conditional Statement

Conditional statements take the form:

`if(`*bool-expression*`)` *if-statement* `else` *else-statement*

If the *bool-expression* evaluates to `true`, the *if-statement* (which itself can be a compound statement) is executed. If *bool-expression* is `false`, then *else-statement* is executed instead. The "`else else-statement`" is optional. If omitted, and *bool-expression* evaluates to `false`, execution continues with the next statement in the containing body.

5.4 For Statement

For statements take one of the two forms below:

`for` (*loop-identifier* `:=` *initializer-expression* `to` *limit-expression* `:` *step-expression*`)` *statement*

`for` (*loop-identifier* `in` *list-expression*) *statement*

In the first form, *loop-identifier* is assigned the value *initializer-expression*. For each iteration, the value of *loop-identifier* is compared with *limit-expression* and, if they are not equal, *statement* is executed. After this, *step-expression* is added to *loop-identifier*. The comparison and execution are then performed again.

step-expression should be an integer or interval (preceded by **up** or **down**) depending on the type of *loop-identifier*. It is possible to omit the "`:` *step-expression*" in which case a default value of 1 is used if *loop-identifier* is an integer, or `up half` is used if *loop-identifier* is a note. *loop-identifier* only exists within *statement*.

In the second form, *loop-identifier* takes the value of each element of $list - expression$ and executes *statement*. It is essentially an alternate representation of the statement:

`for` (i := 0 to `len`(*list-expression*)) {
loop-identifier := *list-expression*[i];
statement
}

5.5 While Statement

The while statement takes the form:

`while (`*bool-expression*`)` *statement*

If *bool-expression* evaluates to `true`, *statement* is executed. After every execution, *bool-expression* will be re-evaluated, and *statement* will be executed again if *bool-expression* was still `true`. If *bool-expression* is `false`, execution skips *statement* and continues with the next statement in the containing body.

5.6 Return Statement

Return statements take the form: `return` *expression*

Any statements in a function body after a return statement will not execute, as control will be transferred back to the calling function. The return type of the function must match that of *expression*, and all return statements within a function body must return expressions of the same type. It is possible to omit *expression*, in which case the inferred type is the special reserved type `void` which cannot be assigned to any identifier at the function call site.

6 Built-In Functions

6.1 len function

The built-in function len takes a single list as an argument (of any type) and returns the number of elements in the list.

6.2 octave function

The built-in function octave takes a single note as an argument, and returns the octave of the note (an integer ranging from 0 to 8).

6.3 pitch function

The built-in function pitch takes a note as an argument, and returns the same note shifted to the 4th octave with the same accidental properties as the original.

6.4 print function

The built-in print function takes the provided argument and prints it to standard output. For strings, integers, and booleans, the value printed is the value of the expression. For lists, the list is printed in a manner that would allow it to be used on the right hand side of a list initialization statement. For notes, the full note description is printed (i.e. this includes the note name, the octave, and accidental information).

6.5 reverse function

The built-in function reverse takes a single list as an argument (of any type) and returns a list with the same elements in reverse order.