nc Final Report

Phillip Ames
COMS W4115
pa2354@columbia.edu

August 16, 2012

Contents

1 Introduction

nc is a statically typed domain specific language that leverages the mathematical nature of music and its associated primitives and empowers its users to easily represent and perform computations in the musical domain.

Western music theory consists of a well-defined set of rules that dictate how a given pitch (a note) can be manipulated into various scales and chords. This system is built around the measurement of distance between pitches using the increment known as a half-step. nc provides typical programming language constructs that allow programmers or musicians to solve a wide array of problems.

The reference implementation for nc is compiled to an intermediate bytecode representation which is then executed by a virtual machine interpreter.

2 Language Tutorial

A brief example of the language syntax with an explanation of the intended functionality (annotated using C-style comments, i.e. `/* comment */`) is in Listing 1.

Listing 1: example.nc

```
/*
 * Function declarations consist of (in this order):
 *  - a return type (list<note>, int, ...)
 *  - a name (major_scale, main, ...)
 *  - named arguments, preceded by their type, enclosed in parentheses
 * Arguments are optional, and should be separated by commas.
 */
list<note> major_scale(note n)
{
  /* This declares a variable 'l' to be an empty list of notes. */
  l := list<note>;
  /*
   * The :: operator concatenates the note onto the front of a list.
   * the addition operator accepts musical nomenclature such
   * as ``whole'' or ``half''
   */
  l = n :: l;
  /* i is assigned each value in the range [0, 6) */
  for (i := 0 to 6) {
    if (i == 2)
      n += half;
    else
      n += whole;
    l = n :: l;
  }

  /*
   * This returns the result of the function 'reverse' called with
   * 'l' (the list of notes) as its argument to the caller of this
   * function.
   */
  return reverse(l);
}

/*
 * Returns a list which denotes all the white keys in ascending order
 * on a piano (uses the fact that playing a major scale from C1 touches
 * just the white keys).
 */
list<note> glissando()
{
  l := list<note>;
  n := C1;
  /* The lowest two keys on a piano are A0 and B0 */
  l = B0 :: A0 :: l;

  /*
   * The << and >> operators shifts notes the specified number of octaves.
   * The ': 1' in this for loop allows you to specify the 'step' (value
   * added to i in each iteration).
   */
  for (i := 0 to 7 : 1) {
    tmp := major_scale(n);
```

```
    for (j in tmp) {
      l = j :: l;
    }
    n = n >> 1;
  }
  l = reverse(C8 :: l);
  return l;
}

/* This is the entry point for every program */
void main()
{
  /*
   * Assigns to ‘‘n” the value of an A note in the 4th octave,
   * which corresponds to the first A above middle C (a.k.a. A4).
   */
  n := A;

  l := glissando();

  print(l);

  /* Comparison (!=) and list dereferencing ([]) operators. */
  if (len(l) != 52 || l[0] != A0)
    print(”Your piano seems to have an incorrect number of white keys”);

  /*
   * This shifts n down 1 octave and assigns the resulting value (A3)
   * to n, replacing the value.
   */
  print(n);
  n = n << 1;
  print(n);
}
```

To compile and run this program, simply execute:

`nc < example.nc`

The output should be:

```
[A0;B0;C1;D1;E1;F1;G1;A1;B1;C2;D2;E2;F2;G2;A2;B2;C3;D3;E3;F3;G3;A3;B3;C4;D4;E4;F4;G4;
A4;B4;C5;D5;E5;F5;G5;A5;B5;C6;D6;E6;F6;G6;A6;B6;C7;D7;E7;F7;G7;A7;B7;C8]
A4
A3
```

The entry point of every *nc* function is the function named `main`, which should take no arguments and have a return type of `void`.

3 Language Reference Manual

3.1 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.

3.2 Lexical Conventions

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

- constants
- identifiers
- keywords
- operators
- punctuation

A brief description of each follows.

3.2.1 Constants

nc recognizes the following types of constants:

String constants

Strings in *nc* begin with a double quote and consist of all subsequent characters until another double quote is found. Strings cannot be assigned to variables, and can only used with the built-in print function.

3.2.2 Literals

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

Boolean literals

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

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.

Pitch literals

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

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.

3.2.3 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.

3.2.4 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`	

3.2.5 Operators

The following operators are supported:

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

The meaning of each is discussed in section 3.4.

3.2.6 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

3.3 Meaning of Identifiers

3.3.1 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 3.2.2.

3.3.2 Declarations

Variable Declarations

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

identifier `:=` *expression*

Where *identifier* matches the format described in section 3.2.3. The result of evaluating *expression* is stored in the newly created identifier.

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 `bool`, `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.

3.3.3 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 subsequent blocks. If an identifier is declared with the same name as an existing identifier from global scope, the outer declarations are shadowed by the inner declaration. Named function arguments also shadow global declarations, but not local declarations. It is an error to declare an identifier multiple times with the same name within the same block or in a child or subsequent block. Scope environments from calling functions are not consulted when resolving a variable reference.

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

3.4 Expressions and Operators

3.4.1 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
`[]`	array dereference
`-`	unary minus
`!`	unary logical NOT
`* / %`	multiplication, division, modulo
`+ -`	addition, subtraction
`<< >>`	octave shifting
`< <= > >= == !=`	relational operators
`&& ||`	logical comparators
`::`	list concatenation operator
`= += -= ::=`	assignment, addition/subtraction/concatenation and assignment

Unary Operators

3.4.2 `!`*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`.

3.4.3 `-`*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.

Binary Arithmetic Operators

3.4.4 *expression* `+` *expression*

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

3.4.5 *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.

3.4.6 *expression* `-` *expression*

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

3.4.7 *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.

3.4.8 *expression* `*` *expression*

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

3.4.9 *expression* `/` *xpression*

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.

3.4.10 *expression* `%` *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.

Logical Operators

3.4.11 *expression* `&&` *expression*

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

3.4.12 *expression* `||` *expression*

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

Shift Operators

3.4.13 *expression* `<<` *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. The left operand must be an expression with note-type, and the right operand must be of integer type.

3.4.14 *expression* `>>` *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. The same type rules apply as with the shift down operator.

Comparison Operators For all comparison operators, the two provided expressions must have the same type, and this type must be int or note. In the case of int-typed expressions, arithmetic comparison is performed. In the case of note-type expressions, the comparison is based on the position of the notes on standard piano keys.

3.4.15 *expression* `<` *expression*

The binary `<` operator compares the two provided expressions and returns `true` if the left hand side is less than the right hand side.

3.4.16 *expression* `>` *expression*

The binary `>` operator compares the two provided expressions and returns `true` if the left hand side is greater than the right hand side.

3.4.17 *expression* `<=` *expression*

The binary `<=` operator compares the two provided expressions and returns `true` if the left hand side is less than or equal to the right hand side.

3.4.18 *expression* `>=` *expression*

The binary `>=` operator compares the two provided expressions and returns `true` if the left hand side is greater than or equal to the right hand side.

3.4.19 *expression* `==` *expression*

The binary `==` operator compares the two provided expressions and returns `true` if the left hand side is equal to the right hand side.

Assignment Operators These operators are right associative.

3.4.20 *identifier* `:=` *expression*

The binary `:=` operator serves the dual function of both declaring a variable within a defined scope (see section 3.3.3), and assigning it the value and type of *expression*. The type of the variable is inferred from the type of *expression*.

3.4.21 *identifier* `=` *expression*

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

3.4.22 *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.

3.4.23 *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.

List Operators

3.4.24 *expression* `::` ***expression***

The binary `::` operator prepends the result of *expression* to the list in *list-expression*. The right operand must be of a list-type, and the type of the left operand must be the same as the type the list holds. This operator is right associative.

3.4.25 *identifier* `::=` ***expression***

The binary `::=` operator is a convenience representation for: $identifier = expression :: identifier$

The same type rules apply as with the `::` operator.

3.4.26 *expression* `[`*expression*`]`

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

3.5 Statements

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

3.5.1 Expression Statement

Expression statements can be built using the operators described described in section 3.4. 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 3.3.2 or be a built-in function described in section 3.6. *argument-list* can consist of 0 or more comma separated expressions.

3.5.2 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 (}).

3.5.3 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.

3.5.4 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 it is not equal to the limit, *statement* is executed. After this, *step-expression* is added to *loop-identifier*. The comparison and execution are then performed again. It is worth stressing that this is an equality comparison, in order to support step values less than 0 (or in the case of notes, downward intervals). As such it may be possible to enter an infinite loop by selecting a step value which, when added to the current loop identifier, would bypass the terminating condition.

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
}

3.5.5 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.

3.5.6 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.

3.6 Built-In Functions

3.6.1 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.

3.6.2 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).

3.6.3 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.

3.6.4 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).

3.6.5 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.

4 Project Plan

Features for the language were added in an incremental style. The natural breakdown of programming language translation into several distinct phases (e.g. scanning, parsing, intermediate code generation, etc.) aided in this workflow. Major milestones included:

2012-06-06	Language proposal submitted
2012-06-27	Language reference manual submitted
2012-06-28	Scanner complete
2012-07-10	Parser complete, all ambiguities resolved
2012-07-11	Support for "Hello, world" style program
2012-07-15	Variable scoping and basic type checking enforced
2012-07-18	Unary and binary operator support
2012-07-21	Support for loops
2012-07-24	Support for lists and several built-in functions added
2012-08-03	Support alternate style for-loops
2012-08-15	Final rounds of possible edge case testing complete

As functionality was implemented, it was accompanied with test cases that attempted to exercise both the basic functionality as well as any edge cases that were considered during the feature development.

A detailed history of work on the project from the Git repository used during development (the output of `git log --date=iso --pretty=tformat:"%ad %s"`) is in listing 2. Much of the byte code interpretation was built on top of the provided *microc* language translation toolkit. While some small modifications were made (e.g. additional byte code instructions) the bulk of this code was applicable to *nc* and therefore did not represent a substantial amount of development time.

Listing 2: Git Log

```
2012-08-15 00:00:32 -0400 Prevent Void-inferred assignments.
2012-08-14 23:58:32 -0400 Set up appropriate default return values.
2012-08-14 23:28:13 -0400 Fix bug involving declaration + ssignment of locals
    from formal arguments.
2012-08-06 23:26:27 -0400 Capture missing declarations from if and while
    statement blocks.
2012-08-05 14:48:51 -0400 Remove unused pattern match warning.
2012-08-04 19:18:24 -0400 Remove unused case (interpreter) from nc.ml
2012-08-04 17:50:32 -0400 Add support for a ::= operator for convenience.
2012-08-04 09:04:14 -0400 Implementation of builtin pitch function.
2012-08-03 10:06:06 -0400 Support string translation of remaining binary
    operators.
2012-08-03 02:46:00 -0400 Refactor support for assigning / reading formal
    function arguments.
2012-08-03 02:09:20 -0400 Tests for foreach.
2012-08-03 02:07:52 -0400 ForEach support.
2012-07-28 16:24:39 -0400 Emit failure on variable re-declaration.
2012-07-26 01:22:56 -0400 Add test case for 'reverse' function.
2012-07-26 01:21:33 -0400 Add test for built-in octave function, fix bug in
    implementation
2012-07-26 01:00:18 -0400 Add 'foreach' stub
2012-07-26 00:52:34 -0400 Clean up missed pattern match warnings (for catch-
    alls omitted).
2012-07-26 00:09:14 -0400 Type checking refactoring.
2012-07-25 09:21:08 -0400 Remove use of StringMap.Cardinal
2012-07-24 23:58:24 -0400 Remove unnecessary TODO.
2012-07-24 23:56:30 -0400 Test cases for list declarations w/expressions.
2012-07-24 23:38:19 -0400 Process expressions in list literal declarations.
2012-07-24 23:25:01 -0400 Stub support for list literal declarations (no
    expression evaluation/insertion).
2012-07-24 01:09:23 -0400 Update proposal sample file to reflect proper syntax
    .
2012-07-24 00:58:42 -0400 Support nested declarations.
2012-07-24 00:31:56 -0400 Reverse list function implementation.
```

```
2012-07-23 23:41:10 -0400 List reversal.
2012-07-23 23:11:41 -0400 Support for list concatenation/dereference.
2012-07-23 22:34:37 -0400 Type expressions for list dereference/concat
2012-07-22 21:42:30 -0400 len implementation and type checking support
2012-07-22 21:20:50 -0400 Support for printing lists.
2012-07-22 20:35:57 -0400 Support for empty lists.
2012-07-21 21:28:20 -0400 Additional binary operator tests / implementations.
2012-07-21 21:15:25 -0400 Test while loops.
2012-07-21 21:09:42 -0400 Tests for 'for' loops.
2012-07-21 21:01:56 -0400 builtin 'octave' function implementation
2012-07-21 19:58:25 -0400 Add additional match cases to silence compiler
    warnings.
2012-07-21 19:32:39 -0400 Support for loops.
2012-07-18 01:43:54 -0400 Clean up compile warnings.
2012-07-18 01:39:29 -0400 Support binary operators involving notes and
    intervals.
2012-07-18 01:16:52 -0400 Refine type checking, support lots of binary
    operators.
2012-07-18 00:10:04 -0400 Unary operator support.
2012-07-17 23:42:25 -0400 Type definitions for binary expressions.
2012-07-17 00:21:14 -0400 Type checking during assignment.
2012-07-16 23:59:33 -0400 Improve test-if.
2012-07-15 17:41:57 -0400 Support conditional branching and type checking for
    if predicates.
2012-07-15 17:17:05 -0400 Refactor function map into environment for
    convenience.
2012-07-15 10:31:48 -0400 Type checking for function call parameters.
2012-07-15 00:23:51 -0400 Add tests for scoping rules and local variables.
2012-07-14 20:32:13 -0400 Type augment formal arguments
2012-07-14 20:15:29 -0400 Local declaration and type support.
2012-07-14 19:37:00 -0400 Improve global declarations and support for
    declarations within a scope.
2012-07-14 16:40:06 -0400 Support type checking when returning global
    variables
2012-07-14 14:02:19 -0400 Return statement type checking.
2012-07-14 12:13:55 -0400 Test for global variable redeclaration.
2012-07-14 11:51:13 -0400 Support global variable declarations.
2012-07-14 11:06:15 -0400 Boolean support.
2012-07-14 10:09:41 -0400 Refactor builtin_idx and add stubs for other builtin
    functions.
2012-07-14 09:43:23 -0400 Correct global variable type.
2012-07-13 00:37:32 -0400 Support note arguments to print.
2012-07-13 00:26:12 -0400 Note string/integer convenience functions.
2012-07-12 22:10:14 -0400 Refactor tests.
2012-07-12 00:07:56 -0400 Simple function tests.
2012-07-11 23:30:59 -0400 Hello, world!
2012-07-11 00:53:28 -0400 Add StringLiteral bytecode statement (but not
    supported on stack yet).
2012-07-11 00:09:03 -0400 refactor built in function setup
2012-07-10 23:28:35 -0400 Small refactor.
2012-07-10 01:59:43 -0400 Baseline modifications to get error-free compilation
    based on new parse tree.
2012-07-09 21:01:10 -0400 execute.ml and bytecode.ml from microc
2012-07-08 11:53:20 -0400 Update proposed code snippet to reflect LRM grammar.
```

```
2012-07-03 11:12:58 -0400 Support string literals.
2012-07-01 22:25:57 -0400 Support "for" statements.
2012-07-01 18:06:18 -0400 List literal declaration
2012-07-01 17:17:22 -0400 Cons operator.
2012-07-01 17:03:32 -0400 Array dereference support.
2012-07-01 16:32:28 -0400 Additional binary operations, unary operators.
2012-07-01 15:05:31 -0400 Support for note literals, binary expressions,
    function calls.
2012-06-30 21:32:06 -0400 Support reading formal parameters and their types.
2012-06-30 20:44:33 -0400 Support literal declarations, improved type
    declarations.
2012-06-30 18:49:00 -0400 Basic (incomplete) parsing of top level declarations
    .
2012-06-28 01:48:08 -0400 Add new lexemes, additional microc baseline files.
2012-06-28 00:33:32 -0400 Baseline compiler from microc language.
```

Due to the size of the team (one individual) no formal style guide was created, however an effort to review and be consistent with the guidelines at `http://caml.inria.fr/resources/doc/guides/guidelines.en.html` was made. Additionally, research into any OCaml lint-like tools was conducted, however no viable options appear to currently exist.

5 Architecture

The overall architecture of the *nc* compiler can be seen in figure 1.

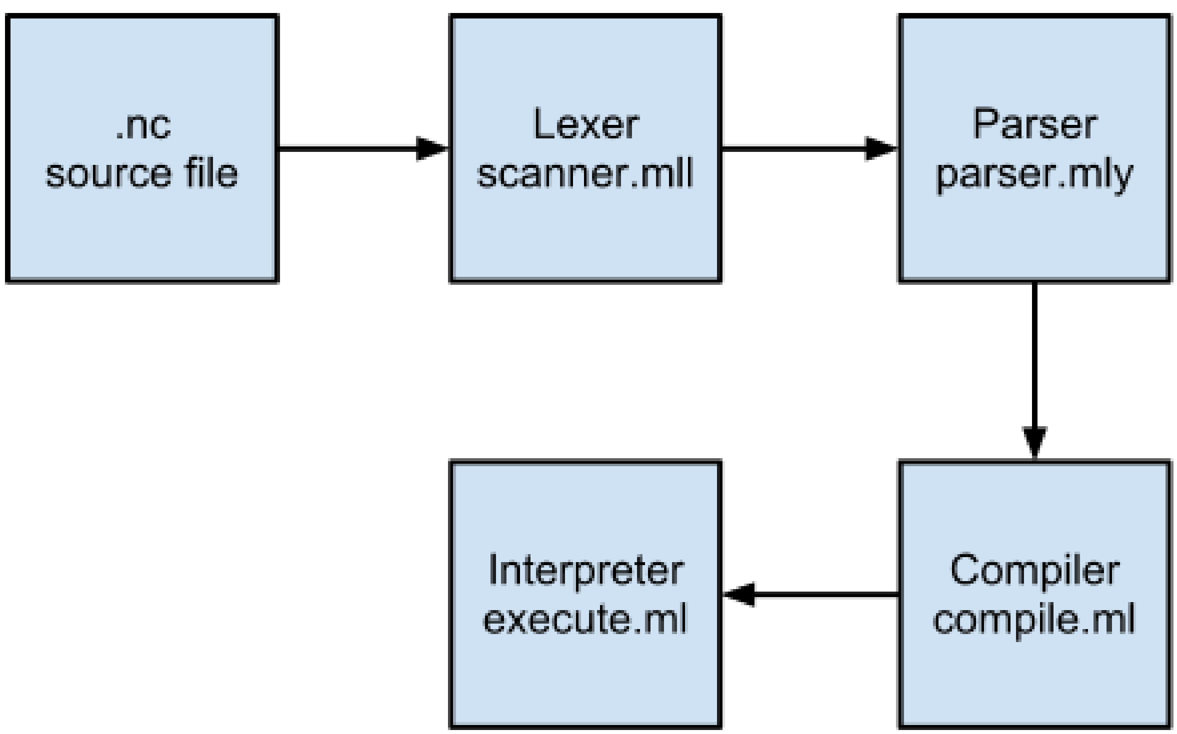

Figure 1: Overall architecture of nc compiler

A small command line utility (listing 11) is responsible for driving this workflow. The lexer(listing 5) consumes the provided source file, emitting a sequence of tokens. These tokens are read by the parser, which uses rules defined in parser.mly(listing 6) to produce an abstract syntax tree.

During the compilation phase(listing 9), the nodes and leaves in the syntax tree are semantically evaluated so tasks such as type inference and type checking can be done. Other important verification performed in this phase includes ensuring that variables have been declared before use, that function calls have the proper number and type of arguments, and so on.

If there were no semantic errors during the compilation phase, the output is a sequence of bytecode instructions which can be processed by the bytecode interpreter(listing 10). While this bytecode is not currently serialized to disk in any form, this feature could be added in order to remove the overhead of the previous phases, as the output should represent a valid *nc* program.

6 Test Plan

A representative source program which illustrates a task that is well specified from an algorithmic perspective, but that can be tricky to perform manually without significant practice, is depicted in listing 3.

Listing 3: 1-4-5.nc

```
/*
 * The I-IV-V chord progression is one of the most common chord progressions
 * in pop music (ref: http://wiki.answers.com/Q/What_is_a_1-4-5_Progression).
 * The Roman numerals represent scale degrees, and are based on the root note
 * of a musical key.  This program can, given any root note as the key,
 * compute the notes which should be played in the chords of a I-IV-V
 * progression.
 */
void main() {
  root := C;
  print(chord(i(root)));
  print(chord(iv(root)));
  print(chord(v(root)));
}

note i(note root) {
  return root;
}

note iv(note root) {
  return root + whole + whole + half;
}

note v(note root) {
  return root + whole + whole + half + whole;
}

/* major chord for a given root note */
list<note> chord(note root) {
  rv := [root];
  rv ::= root + whole + whole;
  rv ::= rv[0] + whole + half;
  return reverse(rv);
}
```

An annotated version of the resulting bytecode for this program can be found in listing 4.

Listing 4: 1-4-5.bc

```
0 global variables
0 Jsr 3             ; Jump to global variable initialization
1 Jsr 64            ; Jump to main
2 Hlt               ; Halt upon return from main
3 Ent 0             ; $start$ entry
4 Rts 0             ; $start$ exit - control flow resumes at #1
5 Ent 1             ; chord(note root) entry - 1 local variable
6 Lfp -2            ; Load formal argument relative to frame pointer (root)
7 Lst note(0 els)   ; Push an empty list of notes
8 Con               ; Insert the formal argument into the just pushed list
9 Sfp 1             ; Store the result (list of 1 element ) in the local slot
10 Lfp -2           ; Load the formal argument again
11 Ivl Whole        ; Push an interval onto the stack
12 Add              ; Add the interval to the note on the stack
13 Ivl Whole        ; ...
14 Add              ; ...
15 Lfp 1            ; Load the list
16 Con              ; Push the resulting note onto the list
17 Sfp 1            ; Store the list locally
18 Drp              ; Remove...
19 Lfp 1            ; ... then re-load the list on the stack
20 Lit 0            ; Push the index we want to dereference
21 Drf              ; Dereference the first element of the list
22 Ivl Whole        ; Perform the next sequence of additions
23 Add              ; ...
24 Ivl Half         ; ...
25 Add              ; ...
26 Lfp 1            ; Load the list again
27 Con              ; Push the final note onto the list
28 Sfp 1            ; Store the list
29 Drp              ; Remove...
30 Lfp 1            ; ... then re-load the list
31 Jsr -5           ; Call built-in 'reverse' function with the list as argument
32 Rts 1            ; Return the reversed list to the caller, consumes 1 formal
33 Lst note(0 els); catch-all default return value for chord()
34 Rts 1            ; return catch-all value
35 Ent 0            ; v(note root) entry - no locals
36 Lfp -2           ; Load formal argument
37 Ivl Whole        ; Perform sequence of additions
38 Add              ; ...
39 Ivl Whole        ; ...
40 Add              ; ...
41 Ivl Half         ; ...
42 Add              ; ...
43 Ivl Whole        ; ...
44 Add              ; ...
45 Rts 1            ; Return result of addition
46 Nte C4           ; catch-all default return value for v()
47 Rts 1            ; return catch-all value
48 Ent 0            ; iv(note root) entry - no locals
49 Lfp -2           ; [as before, in v() function]
50 Ivl Whole        ; ...
51 Add              ; ...
```

```
52 Ivl Whole        ; ...
53 Add              ; ...
54 Ivl Half         ; ...
55 Add              ; ...
56 Rts 1            ; ...
57 Nte C4           ; ...
58 Rts 1            ; ...
59 Ent 0            ; i(note root) entry - no locals
60 Lfp -2           ; Load formal argument
61 Rts 1            ; Return formal argument
62 Nte C4           ; ...
63 Rts 1            ; ...
64 Ent 1            ; main() entry - 1 local
65 Nte C4           ; Push note literal, C4
66 Sfp 1            ; Store note literal in local variable
67 Lfp 1            ; Load local variable
68 Jsr 59           ; Jump to i() function with local variable as argument
69 Jsr 5            ; Jump to chord() with result of i() as argument
70 Jsr -4           ; Jump to print() with result of chord() as argument
71 Drp              ; Drop the resulting list
72 Lfp 1            ; Load local variable
73 Jsr 48           ; Jump to iv() function with local variable as argument
74 Jsr 5            ; [as before, with i() function]
75 Jsr -4           ; ...
76 Drp              ; ...
77 Lfp 1            ; ...
78 Jsr 35           ; Jump to v() function with local variable as argument
79 Jsr 5            ; [as before, with iv() function]
80 Jsr -4           ; ...
81 Drp              ; ...
82 Rts 0            ; Return from main, consume 0 formals
```

A brief description and list of the test suites used to test the translator were:

test-binop-bool.nc	Binary operator test suite for boolean types
test-binop-int.nc	Binary operator test suite for integer types
test-binop-note.nc	Binary operator test suite for note types
test-else.nc	Test 'if (...) else (...)' constructs
test-elseif.nc	Test 'if (...) else if (...) else ...' constructs
test-for.nc	Test basic for loop
test-foreach.nc	Test list-based iterator for loop
test-global.nc	Test global variable declaration and initialization
test-if.nc	Test 'if(...)' constructs
test-list-decl.nc	Test different styles of list declarations
test-list.nc	Test list operations (concatenation, printing, etc.)
test-local.nc	Test for local variable declaration
test-main.nc	Stub program to test basic compilation
test-octave.nc	Test built-in octave function
test-print.nc	Test built-in print function
test-reverse.nc	Test built-in reverse function
test-scope.nc	Test for scoping rules
test-unary.nc	Test unary operators
test-while.nc	Test 'while (...)' construct

These test cases were chosen because they are intended to exercise the code generation and execution paths for all the major language features. Each corresponding test case has an expected output file, and if the output of executing the program does not match the expected output, the test is considered to have failed. During development, changes and refactoring did actually result in failures that were caught by previously written test cases, a testament to their usefulness in ensuring correct operation of the translator.

The test suites were run with the assistance of a shell script. Tests were run before committing to the git repository. In some cases, features were developed and committed using ad-hoc local source files, but feature development would not resume until new tests targeting the feature were written and checked in to the repository.

7 Lessons Learned

Throughout the process of designing and implementing the *nc* compiler, the following key lessons were learned:

- **Value of iterative development** - Any attempt to implement broad pieces of functionality is largely destined for failure. Development should focus on small pieces which can be understood and implemented in isolation. This requires evaluating the feature and perhaps sketching out a design for the larger piece, but it should still be possible to break this down for actual implementation.
- **Value of version control** - Related to the above, having a version control system such as git which allows for frequent, rapid checkpointing, branching for feature development, and other convenient methods, is quite helpful while developing the compiler. As an example, pairing this and the previous lesson, if a single new language construct is added in a distinct set of commits, it is possible for collaborators (or your future self) to go back and identify all areas of the compiler which would need to be modified in order to add additional language constructs. In a team environment, this means all team members would benefit from an individual contributor's work.
- **Value of regression testing** - In combination with version control, regression testing is important because it allows you to identify when and where a particular test suite for the language was broken. In the case of *nc*, a policy was enforced to not commit code if any regression test were to fail. Additionally, requiring that new features be accompanied by tests ensured that the compiler was largely error-free during all stages of development.
- **Value of functional knowledge / OCaml knowledge** - Learning features of OCaml, and how to solve problems using a functional programming paradigm, was helpful in reducing the complexity of the compiler code. OCaml provides constructs which can really aid in reducing the amount of boilerplate necessary to produce a working compiler. Also, learning how to use features of the OCaml compiler/runtime to help debug problems (such as setting the environment variable OCAMLRUNPARAM=b for backtraces of runtime errors, or OCAMLRUNPARAM=p for debugging parsing rule problems) was quite helpful.

8 Appendix

8.1 scanner.mll

Listing 5: scanner.mll

```
{ open Parser }

rule token = parse
[' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"      { comment lexbuf }            (* Comments *)
|  '('      { LPAREN }
|  ')'      { RPAREN }
|  '{'      { LBRACE }
```

```
| '}'       { RBRACE }
| '['       { LBRACKET }
| ']'       { RBRACKET }
| ';'       { SEMI }
| ','       { COMMA }
| '!'       { NOT }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '%'       { MOD }
| '='       { ASSIGN }
| "=="      { EQ }
| "+="      { PLUSEQ }
| "-="      { MINUSEQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "&&"      { AND }
| "||"      { OR }
| ":="      { DECL }
| ":"       { COLON }
| "::"      { CONS }
| "::="     { CONSEQ }
| "<<"      { SHL }
| ">>"      { SHR}
| "bool"    { BOOL }
| "down"    { DOWN }
| "else"    { ELSE }
| "false"   { BOOLLITERAL(false) }
| "for"     { FOR }
| "half"    { HALF }
| "if"      { IF }
| "in"      { IN }
| "int"     { INT }
| "list"    { LIST }
| "note"    { NOTE }
| "to"      { TO }
| "true"    { BOOLLITERAL(true) }
| "up"      { UP }
| "void"    { VOID }
| "while"   { WHILE }
| "whole"   { WHOLE }
| "return"  { RETURN }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['A'-'G']['b' '#']?['0'-'8']? as lxm { NOTELITERAL(lxm) }
| ['a'-'z' 'H'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '\"'[^'"']*'\"' as lxm
            { STRINGLITERAL(String.sub lxm 1 ((String.length lxm) - 2)) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

```

```
and comment = parse
   "*/" { token lexbuf }
| _     { comment lexbuf }
```

8.2 parser.mly

Listing 6: parser.mly

```
%{ open Ast %}

/* assorted punctuation */
%token SEMI COLON LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA QUOTE
/* intervals and other basic types */
%token UP DOWN WHOLE HALF
/* operators */
%token CONS CONSEQ
%token AND OR NOT MOD PLUS MINUS TIMES DIVIDE DECL ASSIGN PLUSEQ MINUSEQ
%token SHL SHR
/* comparators */
%token EQ NEQ LT LEQ GT GEQ
/* keywords */
%token RETURN IF ELSE FOR IN TO WHILE INT BOOL NOTE LIST VOID

%token <int> LITERAL
%token <bool> BOOLLITERAL
%token <string> NOTELITERAL
%token <string> STRINGLITERAL
%token <string> ID
%token NEW
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN PLUSEQ MINUSEQ CONSEQ
%right CONS
%left AND OR
%left EQ NEQ
%left LT GT LEQ GEQ
%left SHL SHR
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right UMINUS NOT
%left LBRACKET

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
   | program vdecl { fst $1 @ [$2], snd $1 }
   | program fdecl { fst $1, ($2 :: snd $1) }
```

```

typedecl:
  formaltype { $1 }
  | VOID { Void }

formaltype:
    BOOL { Bool }
  | INT { Int }
  | NOTE { Note }
  | LIST LT BOOL GT { List(Bool) }
  | LIST LT INT GT { List(Int) }
  | LIST LT NOTE GT { List(Note) }

formals_opt:
  /* nothing */ { [] }
  | formal_list { List.rev $1 }

formal_list:
    formaltype ID { [($1, $2)] }
  | formal_list COMMA formaltype ID { ($3, $4)::$1 }

actuals_opt:
  /* nothing */   { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }

list_members:
    expr { [$1] }
    | list_members SEMI expr { $3 :: $1 }

fdecl:
  typedecl ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE {
        { rettype = $1;
          fname = $2;
          formals = $4;
          body = List.rev $7;
        } }

stmt_list:
  /* nothing */ { [] }
  | stmt_list stmt { $2 :: $1 }

interval:
    WHOLE { Whole }
  | HALF  { Half }

stepexpr:
    expr { $1 }
  | UP interval { StepExpr(Up, $2) }
  | DOWN interval { StepExpr(Down, $2) }

```

```
stmt:
    ID DECL expr SEMI { Declaration($1, $3, false) }
  | expr SEMI { Expr($1) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt { While ($3, $5) }
  | FOR LPAREN ID DECL expr TO expr RPAREN stmt { For($3, $5, $7, Noexpr, $9) }
  | FOR LPAREN ID DECL expr TO expr COLON stepexpr RPAREN stmt
      { For($3, $5, $7, $9, $11) }
  | FOR LPAREN ID IN expr RPAREN stmt { ForEach ($3, $5, $7) }
  | RETURN expr_opt SEMI  { Return($2) }

vdecl:
  ID DECL expr SEMI {
          { vname = $1;
            value = $3;
          } }

expr_opt:
  /* nothing */ { Noexpr }
  | expr        { $1 }

step:
    HALF  { Half }
  | WHOLE { Whole }

expr:
    LITERAL { Literal($1) }
  | ID      { Id($1) }
  | BOOLLITERAL { BoolLiteral($1) }
  | NOTELITERAL { NoteLiteral(noteint_of_str $1) }
  | STRINGLITERAL  { StringLiteral($1) }
  | LIST LT BOOL GT { EmptyList(Bool) }
  | LIST LT INT GT { EmptyList(Int) }
  | LIST LT NOTE GT { EmptyList(Note) }
  | LBRACKET list_members RBRACKET { ListLiteral(List.rev $2) }
  | MINUS expr %prec UMINUS { Binop(Literal(0), Sub, $2) }
  | NOT expr          { Unaryop(Not, $2) }
  | expr LBRACKET expr RBRACKET { Binop($1, Deref, $3) }
  | expr CONS   expr { Binop ($1, Concat, $3) }
  | expr PLUS   step { Binop($1, Add, IntervalLiteral($3)) }
  | expr MINUS  step { Binop($1, Sub, IntervalLiteral($3)) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
```

```
  | expr MOD    expr { Binop($1, Mod, $3) }
  | expr AND    expr { Binop($1, And, $3) }
  | expr OR     expr { Binop($1, Or, $3) }
  | expr SHL    expr { Binop($1, Shl, $3) }
  | expr SHR    expr { Binop($1, Shr, $3) }
  | ID PLUSEQ   step { Assign($1, Binop(Id($1), Add, IntervalLiteral($3))) }
  | ID MINUSEQ  step { Assign($1, Binop(Id($1), Sub, IntervalLiteral($3))) }
  | ID PLUSEQ   expr   { Assign($1, Binop(Id($1), Add, $3)) }
  | ID MINUSEQ  expr   { Assign($1, Binop(Id($1), Sub, $3)) }
  | ID CONSEQ   expr   { Assign($1, Binop($3, Concat, Id($1))) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
```

8.3 ast.ml

Listing 7: ast.ml

```
(* Arguably, "Not" may merit its own type since it is unary *)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq
          | Mod | Not | And | Or | Shl | Shr | Concat | Deref

type direction = Up | Down
type interval = Whole | Half

type primitive =   Bool | Int | Note | List of primitive
                 | Interval | Void | Safe | Illegal

type expr =
    Literal of int
  | BoolLiteral of bool
  | NoteLiteral of int
  | ListLiteral of expr list
  | StringLiteral of string
  | EmptyList of primitive
  | IntervalLiteral of interval
  | Id of string
  | Binop of expr * op * expr
  | Unaryop of op * expr
  | Assign of string * expr
  | Call of string * expr list
  | StepExpr of direction * interval
  | Noexpr

type stmt =
    Block of stmt list
  | Declaration of string * expr * bool (* identifier, value, in foreach? *)
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of string * expr * expr * expr * stmt (* id, initial, limit, step,
     body *)
  | ForEach of string * expr * stmt (* refactor as for? *)
```

```
  | While of expr * stmt

type func_decl = {
    rettype : primitive;
    fname : string;
    formals : (primitive * string) list;
    body : stmt list;
  }

type variable_decl = {
  vname : string;
  value : expr;
}

type program = variable_decl list * func_decl list

let rec string_of_type t = match t with
    Bool -> "bool"
  | Int -> "int"
  | Note -> "note"
  | List(t) -> "list<" ^ string_of_type t ^ ">"
  | Interval -> "interval"
  | Void -> "void"
  | Safe -> "safe"
  | Illegal -> "illegal"

let notestring_of_int i = (match i with
        0 -> "A0"
      | 1 -> "A#0"
      | 2 -> "B0"
      | _ -> (match ((i-3) mod 12) with
          0 -> "C"
        | 1 -> "C#"
        | 2 -> "D"
        | 3 -> "D#"
        | 4 -> "E"
        | 5 -> "F"
        | 6 -> "F#"
        | 7 -> "G"
        | 8 -> "G#"
        | 9 -> "A"
        | 10 -> "A#"
        | 11 -> "B"
        | _ -> raise (Failure("oob note"))) ^ (string_of_int (((i-3)/12)+1))
      )

let index_in_key note = match note with
    'C' -> 0
  | 'D' -> 2
  | 'E' -> 4
  | 'F' -> 5
  | 'G' -> 7
  | 'A' -> 9
  | 'B' -> 11
```

```
  | _ -> raise (Failure("invalid note")) ;;

let accidental_modifier = function
    'b' -> (-1)
  | '#' -> 1
  | _ -> raise (Failure("invalid accidental"))

let noteint_of_str s = match s with
    "A0" -> 0
  | "A#0" -> 1
  | "Bb0" -> 1
  | "B0" -> 2
  | "B#0" -> 3
  | _ -> 3 + (match String.length s with
      1 -> 36 + (index_in_key (String.get s 0))
    | 2 -> (match String.get s 1 with
          'b' -> 36 + (index_in_key (String.get s 0)) - 1
        | '#' -> 36 + (index_in_key (String.get s 0)) + 1
        | _ -> (index_in_key (String.get s 0) +
               (12 * ((int_of_char(String.get s 1) - 1) - int_of_char('0')))))
    | 3 -> (index_in_key (String.get s 0) +
            accidental_modifier (String.get s 1) +
            (12 * ((int_of_char(String.get s 2) - 1) - int_of_char('0'))))
    | _ -> raise (Failure("invalid syntax"))) ;;


let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | BoolLiteral(false) -> "false"
  | BoolLiteral(true) -> "true"
  | NoteLiteral(n) -> notestring_of_int n
  | EmptyList(t) -> "list<" ^ string_of_type t ^ ">"
  | ListLiteral(l) ->
          "[" ^ (String.concat ";" (List.map string_of_expr l)) ^ "]"
  | StringLiteral(s) -> "\"" ^ String.escaped s ^ "\""
  | IntervalLiteral(s) ->
                  (match s with Whole -> "whole" | Half -> "half")
  | Id(s) -> s
  | Binop(e1, Deref, e2) -> "(" ^ string_of_expr e1 ^ "[" ^
      string_of_expr e2 ^ "])"
  | Binop(e1, o, e2) -> "(" ^
      string_of_expr e1 ^ " " ^
      (match o with
        Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      | Shr -> ">>" | Shl -> "<<" | Or -> "||" | And -> "&&"
      | Not -> "!" | Mod -> "%" | Concat -> "::"
      | _ -> raise(Failure("illegal binop"))) ^ " " ^
      string_of_expr e2 ^ ")"
  | Unaryop(o, e) -> "(" ^
      (match o with
        Not -> "!"
      | _ -> raise (Failure("illegal unary operator")))
```

```
      ^ string_of_expr e ^ ")"
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | StepExpr(d, i) ->
                    (match d with Up -> "up" | Down -> "down") ^ " " ^
                    (match i with Whole -> "whole" | Half -> "half")
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Declaration(i, e, _) -> i ^ " := " ^ string_of_expr e ^ ";\n"
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(i, e1, e2, e3, s) -> "for (" ^ i ^ " := " ^ string_of_expr e1 ^
      " to " ^ string_of_expr e2 ^
      (match e3 with Noexpr -> "" | _ -> " : " ^ string_of_expr e3) ^ ") {\n" ^
      string_of_stmt s ^ "\n}\n"
  | ForEach(i, e, s) -> "for (" ^ i ^ " in " ^ string_of_expr e ^ ") {\n" ^
      string_of_stmt s ^ "\n}\n"
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl id = id.vname ^ " := " ^ string_of_expr id.value
                         ^ ";\n"

let string_of_formal f = string_of_type (fst f) ^ " " ^ snd f
let string_of_formals f = String.concat "," (List.map string_of_formal f)

let string_of_fdecl fdecl =
  string_of_type fdecl.rettype ^ " " ^ fdecl.fname ^ "(" ^
  string_of_formals (fdecl.formals) ^ ")\n{\n" ^
(*  String.concat "" (List.map string_of_vdecl fdecl.locals) ^ *)
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.4 bytecode.ml

Listing 8: bytecode.ml

```
open Ast
type bstmt =
    Lit of int    (* Push a literal *)
  | Sng of string (* Push a string *)
```

```
  | Nte of int    (* Push a note *)
  | Bln of bool   (* Push a boolean *)
  | Ivl of Ast.interval (* Push an interval *)
  | Lst of (Ast.primitive * int * (int list))
  | Drp           (* Discard a value *)
  | Dup           (* Duplicate top of stack *)
  | Efr           (* Enter a foreach block *)
  | Swp           (* swap two top stack arguments *)
  | Bin of Ast.op (* Perform arithmetic on top of stack *)
  | Una of Ast.op (* Perform a unary operation *)
  | Dbg           (* dump stack *)
  | Nop           (* no-op *)
  | Lod of int    (* Fetch global variable *)
  | Str of int    (* Store global variable *)
  | Lfp of int    (* Load frame pointer relative *)
  | Sfp of int    (* Store frame pointer relative *)
  | Jsr of int    (* Call function by absolute address *)
  | Ent of int    (* Push FP, FP -> SP, SP += i *)
  | Rts of int    (* Restore FP, SP, consume formals, push result *)
  | Beq of int    (* Branch relative if top-of-stack is true *)
  | Bne of int    (* Branch relative if top-of-stack is false *)
  | Bra of int    (* Branch relative *)
  | Hlt           (* Terminate *)

type prog = {
    num_globals : int;   (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }

let string_of_stmt = function
    Lit(i) -> "Lit " ^ string_of_int i
  | Sng(s) -> "Sng " ^ String.escaped s
  | Nte(n) -> "Nte " ^ notestring_of_int n
  | Bln(b) -> "Bln " ^ string_of_bool b
  | Lst(t, l, c) -> "Lst " ^ string_of_type t ^ "(" ^ string_of_int l ^ " els)"
  | Ivl(Half) -> "Ivl Half"
  | Ivl(Whole) -> "Ivl Whole"
  | Drp -> "Drp"
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Bin(Ast.Deref) -> "Drf"
  | Bin(Ast.Concat) -> "Con"
  | Bin(Ast.Shr) -> "Shr"
  | Bin(Ast.Shl) -> "Shl"
  | Bin(Ast.Or) -> "Or"
```

```
  | Bin(Ast.And) -> "And"
  | Bin(Ast.Mod) -> "Mod"
  | Una(Ast.Not) -> "Not"
  | Dbg -> "Dbg"
  | Nop -> "Nop"
  | Dup -> "Dup"
  | Efr -> "Efr"
  | Swp -> "Swp"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Hlt    -> "Hlt"
  | _ -> raise (Failure("unimplemented instruction"))

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

8.5 compile.ml

Listing 9: compile.ml

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    (* Convenience map from function name -> declaration *)
    fmap : func_decl StringMap.t;
    (* Map from function name -> (index, declaration) *)
    function_index : (int * func_decl) StringMap.t;
    (* "Address" for global variables *)
    global_index   : (int * primitive) StringMap.t;
    (* FP offset for args *)
    formal_index : (int * primitive) StringMap.t;
    (* FP offset for locals *)
    local_index    : (int * primitive) StringMap.t;
    declared_vars : bool array;
  }

(* Determines type based on function declaration / scopes *)
let rec get_type expr env = match expr with
```

```
    Literal(_) -> Int
  | BoolLiteral(_) -> Bool
  | NoteLiteral(_) -> Note
  | IntervalLiteral(_) -> Interval
  (* "Illegal" can still be passed to "Safe" *)
  | StringLiteral(_) -> Illegal
  | EmptyList(t) -> List(t)
  | ListLiteral(e) ->
      let t = (List.fold_left (fun t e -> (match (t, get_type e env) with
        (Illegal, x) -> x
      | (x, y) -> if x = y then x
                  else raise(Failure("conflict types in list literal"))))
      Illegal e) in
              (match t with
                Int -> List(Int) | Bool -> List(Bool) | Note -> List(Note)
              | _ -> raise(Failure("illegal list type")))
  | Unaryop(_, e) -> get_type e env
  | Binop(e1, op, e2) ->
      (let t1 = get_type e1 env and
           t2 = get_type e2 env in match (t1, op, t2) with
      | (x, op, y)
          when (x = y &&
               (op = Equal || op = Neq) &&
               (x = Int || x = Note || x = Bool)) -> Bool
      | (x, op, y)
          when (x = y &&
               (op = Leq || op = Geq || op = Less || op = Greater) &&
               (x = Int || x = Note)) -> Bool
      | (Bool, op, Bool) when (op = And || op = Or) -> Bool
      | (Note, op, Interval) when (op = Add || op = Sub) -> Note
      | (Note, op, Int) when (op = Shl || op = Shr) -> Note
      | (Int, op, Int)
          when (op = Add || op = Sub || op = Mult ||
                op = Div || op = Mod) -> Int
      | (x, Concat, List(y)) when (x = y) -> List(x)
      | (List(t), Deref, Int) -> t
      | _ -> raise(Failure("illegal types to binary operator")))
  | Call(f, formals) -> (match f with
        "reverse" -> get_type (List.nth formals 0) env
      | _ -> (StringMap.find f env.fmap).rettype)
  | Noexpr -> Void
  | Assign(_) -> Void
  | Id(s) ->
      (try snd(StringMap.find s env.local_index)
        with Not_found -> try snd(StringMap.find s env.formal_index)
        with Not_found -> try snd(StringMap.find s env.global_index)
        with Not_found -> raise(Failure("type undeclared variable " ^ s)))
  | _ -> Illegal

(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

```

```
(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(* Count # of local declarations in a function (to allocate stack) *)
let rec sum_decls body =
  let rec sum_decl sum stmt =
      match stmt with
        Declaration(_) -> sum + 1
      | Block(s) -> sum_decls s
      | For(i, e1, e2, e3, s) -> (sum + 1) + (sum_decl 0 s)
      | ForEach(i, e, s) -> sum + 2 + (sum_decl 0 s)
      | If(p, t, e) -> sum + (sum_decl 0 t) + (sum_decl 0 e)
      | While(e, s) -> sum + (sum_decl 0 s)
      | _ -> sum
    in List.fold_left sum_decl 0 body

(* Create list of local declarations (to determine FP offsets) *)
let decl_list body =
  let rec collect_decl l stmt = match stmt with
      Declaration(i, e, _) as d -> d::l
    | For(i, e1, e2, e3, s) -> collect_decl ((Declaration(i, e1, false))::l) s
    (*
     * The "true" 3rd argument indicates this declaration is from a
     * ForEach, so when we do type checking, we know that the type is
     * actually a Foo, not a List<Foo>
     *)
    | ForEach(i, e, s) -> collect_decl ((Declaration(i, e, true))::l) s
    | Block(s) -> List.fold_left collect_decl l s
    | If(p, t, e) -> collect_decl (collect_decl l e) t
    | While(e, s) -> collect_decl l s
    | _ -> l in
    List.rev (List.fold_left collect_decl [] body)

(* Create a list of formals (to determine FP offsets) *)
let list_formals formals = List.fold_left (fun l d -> (snd d)::l) [] formals

let builtins = [
        { rettype = Int;
          fname = "len";
          formals = [(List(Safe), "arg")];
          body = []
        };
        { rettype = Int;
          fname = "octave";
          formals = [(Note, "n")];
          body = []
        };
        { rettype = Note;
          fname = "pitch";
          formals = [(Note, "n")];
          body = []
        };
        { rettype = Void;
```

```
          fname = "print";
          formals = [(Safe, "p")];
          body = [];
        };
        { rettype = Safe;
          fname = "reverse";
          formals = [(List(Safe), "list")];
          body = [];
        }]

(* returns (int * primitive) - address / type for global vdecls *)
let compute_global_index decls env funcs =
  let decls =
    List.map (fun (n, {vname=i; value=e}) ->
            (n, Declaration(i, e, false))) decls in
  let rec build_index decls env =
    match decls with
    | [(n, Declaration(i, e, _))] ->
        StringMap.add i (n, get_type e env) env.global_index
    | (n, Declaration(i, e, _))::tl ->
        let new_idx =
          StringMap.add i (n, get_type e env) env.global_index in
        build_index tl { env with global_index = new_idx }
    | [] -> env.global_index
    | _ -> raise(Failure("illegal argument to compute_index"))
  in build_index decls env

let check_declaration id env =
  let idx =
    try (fst(StringMap.find id env.local_index))
    with Not_found -> try (fst(StringMap.find id env.formal_index))
    with Not_found -> -1 in
  let is_global = try (ignore (StringMap.find id env.global_index); true)
                  with not_found -> false in

  if (idx > 0 && env.declared_vars.(idx - 1) = false) then
    raise(Failure("variable " ^ id ^ " not yet declared"))
  else if (idx = 0 && is_global = false) then
    raise(Failure("undeclared variable " ^ id))
  else ()

let declare id env =
  let idx = (fst(StringMap.find id env.local_index)) in
    if (env.declared_vars.(idx - 1) = false) then
      env.declared_vars.(idx - 1) <- true
    else
      raise(Failure("redeclaration of " ^ id))

(* does the same as compute_global_index but for local variables *)
let compute_local_index decls env funcs =
  let rec build_index decls env =
    match decls with
    | [(n, Declaration(i, e, false))] ->
        StringMap.add i (n, get_type e env) env.local_index
```

```
    | [(n, Declaration(i, e, true))] ->
        let t = (match get_type e env with
            List(Int) -> Int
          | List(Note) -> Note
          | List(Bool) -> Bool
          | _ -> raise(Failure("invalid type in foreach"))) in
        StringMap.add i (n, t) env.local_index
    | (n, Declaration(i, e, false))::tl ->
        let new_idx =
          StringMap.add i (n, get_type e env) env.local_index in
        build_index tl { env with local_index = new_idx }
    | (n, Declaration(i, e, true))::tl ->
        let t = (match get_type e env with
            List(Int) -> Int
          | List(Note) -> Note
          | List(Bool) -> Bool
          | _ -> raise(Failure("invalid type in foreach"))) in
        let new_idx =
          StringMap.add i (n, t) env.local_index in
        build_index tl { env with local_index = new_idx }
    | [] -> env.local_index
    | _ -> raise(Failure("illegal argument to compute_index"))
  in build_index decls env

(* and the same, but for formals *)
let compute_formal_index formals =
    List.fold_left
      (fun m (n, (t, i)) -> StringMap.add i (n, t) m) StringMap.empty formals

let len_arg_check t1 t2 = match (t1, t2) with
    (List(Safe), List(_)) -> true
  | (List(_), List(Safe)) -> true
  | _ -> false

let check_function_args actuals f env =
  let arg_types = List.map (fun formal -> fst formal) f.formals in
  List.map2
  (fun e t2 -> let t1 = get_type e env in
  if t1 = Safe || t2 = Safe || (t1 = t2) || (len_arg_check t1 t2) then true
  else
    raise(Failure("invalid arg type in call to " ^ f.fname)))
  actuals arg_types

let default_return fdecl = match fdecl.rettype with
    Bool -> [Bln false]
  | Int -> [Lit 0]
  | Void -> []
  | Note -> [Nte (noteint_of_str "C4")]
  | List(t) -> [Lst (t, 0, [])]
  | _ -> raise(Failure("illegal return type for " ^ fdecl.fname))

let check_assign id expr env =
  (* verify it was declared *)
  check_declaration id env;
```

```
  let t1 = get_type expr env and
      t2 = (try snd(StringMap.find id env.local_index)
      with Not_found -> try snd(StringMap.find id env.formal_index)
      with Not_found -> try snd(StringMap.find id env.global_index)
      with Not_found -> raise (Failure ("undeclared variable " ^ id))) in
  if (not(t1 = t2) || (t1 = Void || t1 == Illegal)) then
    raise(Failure("illegal type in assignment to " ^ id ^
          "(" ^ (string_of_type t2) ^ " = " ^ (string_of_type t1) ^ ")"))

(** Translate a program in AST form into a bytecode program.  Throw an
    exception if something is wrong, e.g., a reference to an unknown
    variable or function *)
let translate (globals, functions) =
  let fmap = List.fold_left
    (fun m f -> StringMap.add f.fname f m) StringMap.empty functions in
  let fmap = List.fold_left
    (fun m f -> StringMap.add f.fname f m) fmap builtins in
  (* pseudo-function to initialize global variables *)
  let start_fdecl =
          { rettype = Void;
            fname = "$start$";
            formals = [];
            body = List.map
            (function { vname=id; value=expr } -> Expr(Assign(id, expr)))
            globals
          } in
  let functions = start_fdecl::functions in
  (* Assign indexes to function names; built-ins are special *)

  let function_indexes =
    let bf = enum (-1) (-1) builtins and uf = enum 1 1 functions in
      List.fold_left
      (fun m (i, f) -> StringMap.add f.fname (i, f) m)
        StringMap.empty (bf @ uf) in
  (* Allocate "addresses" for each global variable and assign types *)
  let global_indexes =
    (* gi = (0, (Declaration("foo", Expr(Id("i"))) *)
    let gi = enum 1 0 globals in
    let stubenv =
            { fmap = fmap;
              function_index = function_indexes;
              global_index = StringMap.empty;
              formal_index = StringMap.empty;
              local_index = StringMap.empty;
              declared_vars = Array.make 0 false;
          } in
    compute_global_index gi stubenv fmap in

  let sm_length = (StringMap.fold (fun k v s -> s + 1) global_indexes 0) in
  let verify_globals =
    if (List.length globals != sm_length) then
      raise (Failure("global variable redeclaration")) in
      ignore verify_globals;

```

```
(* Translate a function in AST form into a list of bytecode statements *)
let translate env fdecl =
  (* Bookkeeping: FP offsets for locals and arguments *)
  let num_formals = List.length fdecl.formals
  and num_locals = sum_decls fdecl.body
  (* this must come first to allow initialization of locals with formals *)
  in let env = { env with
    formal_index = compute_formal_index (enum (-1) (-2) fdecl.formals);
  }
  in let env = { env with
    local_index =
      compute_local_index (enum 1 1 (decl_list fdecl.body)) env fmap;
    declared_vars = Array.make num_locals false } in
  let rec expr = function
      Literal i -> [Lit i]
    | StringLiteral s -> [Sng s]
    | NoteLiteral n -> [Nte n]
    | BoolLiteral b -> [Bln b]
    | IntervalLiteral i -> [Ivl i]
    | EmptyList(t) -> [Lst(t, 0, [])]
    | ListLiteral(el) as ll -> let
        values =
          (List.fold_left (fun bc e -> bc @ expr e) [] el) and
        ldecl = (match get_type ll env with
            List(t) -> [Lst(t, 0, [])]
          | _ -> raise(Failure("illegal list type"))) and
        cons =
            (List.fold_left (fun bc i -> Bin(Ast.Concat)::bc) [] el)
        in values @ ldecl @ cons
    | Id s ->
        (try [Lfp (fst(StringMap.find s env.local_index))]
        with Not_found -> try [Lfp (fst(StringMap.find s env.formal_index))]
        with Not_found -> try [Lod (fst(StringMap.find s env.global_index))]
        with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    | Unaryop (op, e) -> expr e @ [Una op]
    | Binop (e1, op, e2) as b ->
                    ignore (get_type b env); (* check type of expression *)
                    expr e1 @ expr e2 @ [Bin op]
    | Assign (s, e) -> check_assign s e env; expr e @
        (try [Sfp (fst(StringMap.find s env.local_index))]
      with Not_found -> try [Sfp (fst(StringMap.find s env.formal_index))]
      with Not_found -> [Str (fst(StringMap.find s env.global_index))])
    | Call (fname, actuals) -> (try
      let f = StringMap.find fname env.function_index in
        ignore (check_function_args actuals (snd f) env);
        (List.concat (List.map expr (List.rev actuals))) @
        [Jsr (fst f) ]
      with Not_found -> raise (Failure ("undefined function " ^ fname)))
    | Noexpr -> []
    | _ -> raise(Failure("unimplemented expr translation"))

  in let rec stmt = function
      Block sl     ->  List.concat (List.map stmt sl)
    | Declaration(i, e, _) -> declare i env; expr (Assign(i, e))
```

```
      | Expr e        -> expr e @ [Drp]
      | Return e      ->
        if (fdecl.rettype = (get_type e env)) then
          expr e @ [Rts num_formals]
        else
          raise(Failure("illegal return type in " ^ fdecl.fname))
      | If (p, t, f) -> let t' = stmt t and f' = stmt f in
        if ((get_type p env) = Bool) then
          expr p @ [Bne(2 + List.length t')] @
          t' @ [Bra(1 + List.length f')] @ f'
        else
          raise(Failure("non-bool argument to if statement"))
      | For(i, e1, e2, e3, b) ->
          let d = Declaration(i, e1, false)
          and p = Binop(Id(i), Neq, e2)
          and s = (let t = (get_type (Id(i)) env) in
            match (t, e3) with
              (Int, Noexpr) -> Assign(i, Binop(Id(i), Add, Literal(1)))
            | (Note, Noexpr) -> Assign(i, Binop(Id(i), Add,
                                  IntervalLiteral(Half)))
            | (Note, StepExpr(Up, ivl)) -> Assign(i, Binop(Id(i), Add,
                                  IntervalLiteral(ivl)))
            | (Note, StepExpr(Down, ivl)) -> Assign(i, Binop(Id(i), Sub,
                                  IntervalLiteral(ivl)))
            | (_, _) -> Assign(i, Binop(Id(i), Add, e3))) in
          stmt (Block([d; While(p, Block([b; Expr(s)]))]))
      | ForEach(i, e, b) -> declare i env;
        let t = get_type e env in (match t with
            List(_) -> ()
          | _ -> raise(Failure("cannot call foreach on non-list")));
          let b' = stmt b and e' = expr e and u = [Lit 1; Bin(Add)] in
        let prologue = [Efr; Lit (-1)]
        and epilogue = [Drp; Drp]
        and test = [Dup] @ e' @ [Jsr(-1); Bin(Equal)]
        and asn = [Dup] @ e' @ [Swp; Bin(Deref);
                   Sfp (fst(StringMap.find i env.local_index)); Drp] in
      prologue @
      [Bra(1+List.length b' + List.length asn)] @ asn @ b' @ u @ test @
      [Bne(-(List.length asn + List.length b' +
             List.length u + List.length test))] @
      epilogue
      | While (e, b) ->
          let b' = stmt b and e' = expr e in
          [Bra (1+ List.length b')] @ b' @ e' @
          [Beq (-(List.length b' + List.length e'))]
    in [Ent num_locals] @      (* Entry: allocate space for locals *)
    stmt (Block fdecl.body) @  (* Body *)
    default_return fdecl @
    [Rts num_formals]
  in let env = { fmap = fmap;
     function_index = function_indexes;
                 global_index = global_indexes;
     formal_index = StringMap.empty;
     local_index = StringMap.empty;
```

```
      declared_vars = (Array.make 0 false)} in
  (* Code executed to start the program: Jsr $start$; Jsr main; halt *)
  let entry_function = try
    [Jsr (fst (StringMap.find "$start$" function_indexes));
     Jsr (fst (StringMap.find "main" function_indexes));
     Hlt]
  with Not_found -> raise (Failure ("no \"main\" function"))
  in

  (* Compile the functions *)
  let func_bodies = entry_function :: List.map (translate env) functions in

  (* Calculate function entry points by adding their lengths *)
  let (fun_offset_list, _) = List.fold_left
      (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in
  let func_offset = Array.of_list (List.rev fun_offset_list) in

  { num_globals = List.length globals;
    (* Concatenate the compiled functions and replace the function
       indexes in Jsr statements with PC values *)
    text = Array.of_list (List.map (function
        Jsr i when i > 0 -> Jsr func_offset.(i)
      | _ as s -> s) (List.concat func_bodies))
  }
```

8.6 execute.ml

Listing 10: execute.ml

```
open Ast
open Bytecode

type stack_contents =
    StackInt of int
  | StackNte of int
  | StackSng of string
  | StackBln of bool
  | StackIvl of Ast.interval
  | StackPos of int
  | StackLst of primitive * int * (int list)
  | RetAddr of int

let string_of_stack item = match item with
    StackInt(i) -> "StackInt(" ^ string_of_int(i) ^ ")"
  | StackNte(i) -> "StackNte(" ^ string_of_int(i) ^ ")"
  | StackSng(s) -> "StackSng(" ^ s ^ ")"
  | StackBln(b) -> "StackBln(" ^ string_of_bool(b) ^ ")"
  | StackIvl(Half) -> "StackIvl(Half)"
  | StackIvl(Whole) -> "StackIvl(Whole)"
  | StackPos(i) -> "StackPos(" ^ string_of_int i ^ ")"
  | StackLst(t, l, v) -> "StackLst(" ^ (string_of_type t) ^ ", " ^
                          string_of_int(l) ^ " els)"
  | RetAddr(i) -> "RetAddr(" ^ string_of_int(i) ^ ")"
```

```

let rec print_stack sp pos len =
  if len > 0 then
    (print_endline ("sp(" ^ (string_of_int pos) ^ ") = " ^
    string_of_stack (sp.(pos-1))); print_stack sp (pos-1) (len-1))
  else ()

let ret_addr s = match s with
    RetAddr(i) -> i
  | _ -> raise (Failure("trying to compute ret address from non-address"))

let stack_pos s = match s with
    StackPos(i) -> i
  | _ -> raise (Failure("trying to compute stack address from non-address"))

(* Stack layout just after "Ent":

              <-- SP
   Local n
   ...
   Local 0
   Saved FP   <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)

let execute_prog prog =
  let stack = Array.make 1024 (StackInt(0))
  and globals = Array.make prog.num_globals (StackInt(0)) in

  let rec exec fp sp pc = match prog.text.(pc) with
    Lit i  -> stack.(sp) <- StackInt(i) ; exec fp (sp+1) (pc+1)
  | Sng s  -> stack.(sp) <- StackSng(s); exec fp (sp+1) (pc+1)
  | Nte n  -> stack.(sp) <- StackNte(n); exec fp (sp+1) (pc+1)
  | Bln b  -> stack.(sp) <- StackBln(b); exec fp (sp+1) (pc+1)
  | Ivl i  -> stack.(sp) <- StackIvl(i); exec fp (sp+1) (pc+1)
  | Lst (t, l, c) -> stack.(sp) <- StackLst(t, l, c); exec fp (sp+1) (pc+1)
  | Drp    -> exec fp (sp-1) (pc+1)
  | Una op -> let op1 = stack.(sp-1) in stack.(sp-1) <- (match op with
      Not -> (match op1 with
                StackBln(b) -> StackBln(not b)
              | _ -> raise(Failure("invalid unary operand type")))
      | _ -> raise(Failure("invalid unary operator")));
    exec fp (sp) (pc+1)
  | Bin op -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
      stack.(sp-2) <- (match op with
        Add    -> (match (op1, op2) with
            (StackInt(i1), StackInt(i2)) -> StackInt(i1+i2)
          | (StackNte(n1), StackIvl(i1)) ->
            (let int_of_ivl ivl = (match ivl with Whole -> 2 | Half -> 1) in
                StackNte(n1 + int_of_ivl i1))
          | _ -> raise(Failure("invalid add operands")))
      | Sub -> (match (op1, op2) with
```

```
              (StackInt(i1), StackInt(i2)) -> StackInt(i1-i2)
            | (StackNte(n1), StackIvl(i1)) ->
              (let int_of_ivl ivl = (match ivl with Whole -> 2 | Half -> 1) in
                 StackNte(n1 - int_of_ivl i1))
            | _ -> raise(Failure("invalid sub operands")))
        | Mult -> (match (op1, op2) with
              (StackInt(i1), StackInt(i2)) -> StackInt(i1*i2)
          | _ -> raise(Failure("invalid mul operands")))
        | Div -> (match (op1, op2) with
              (StackInt(i1), StackInt(i2)) -> StackInt(i1/i2)
          | _ -> raise(Failure("invalid div operands")))
        | Mod     -> (match (op1, op2) with
            (StackInt(i1), StackInt(i2)) -> StackInt(i1 mod i2)
          | _ -> raise(Failure("invalid mod operands")))
        | Equal   ->  (match (op1, op2) with
            (StackBln(b1), StackBln(b2)) -> StackBln(b1 == b2)
          | (StackInt(i1), StackInt(i2)) -> StackBln(i1 == i2)
          | (StackNte(n1), StackNte(n2)) -> StackBln(n1 == n2)
          | _ -> raise(Failure("stack corrupt")))
        | Neq     ->  (match (op1, op2) with
            (StackBln(b1), StackBln(b2)) -> StackBln(b1 != b2)
          | (StackInt(i1), StackInt(i2)) -> StackBln(i1 != i2)
          | (StackNte(n1), StackNte(n2)) -> StackBln(n1 != n2)
          | _ -> raise(Failure("stack corrupt")))
        | Less    ->  (match (op1, op2) with
            (StackInt(i1), StackInt(i2)) -> StackBln(i1 < i2)
          | (StackNte(n1), StackNte(n2)) -> StackBln(n1 < n2)
          | _ -> raise(Failure("stack corrupt")))
        | Leq     ->  (match (op1, op2) with
            (StackInt(i1), StackInt(i2)) -> StackBln(i1 <= i2)
          | (StackNte(n1), StackNte(n2)) -> StackBln(n1 <= n2)
          | _ -> raise(Failure("stack corrupt")))
        | Greater ->  (match (op1, op2) with
            (StackInt(i1), StackInt(i2)) -> StackBln(i1 > i2)
          | (StackNte(n1), StackNte(n2)) -> StackBln(n1 > n2)
          | _ -> raise(Failure("stack corrupt")))
        | Geq     ->  (match (op1, op2) with
            (StackInt(i1), StackInt(i2)) -> StackBln(i1 >= i2)
          | (StackNte(n1), StackNte(n2)) -> StackBln(n1 >= n2)
          | _ -> raise(Failure("stack corrupt")))
        | Or      -> (match (op1, op2) with
            (StackBln(b1), StackBln(b2)) -> StackBln(b1 || b2)
          | _ -> raise(Failure("stack corrupt")))
        | And     -> (match (op1, op2) with
            (StackBln(b1), StackBln(b2)) -> StackBln(b1 && b2)
          | _ -> raise(Failure("stack corrupt")))
        | Shl     -> (match (op1, op2) with
            (StackNte(n), StackInt(i)) -> StackNte(n - (12*i))
          | _ -> raise(Failure("stack corrupt")))
        | Shr     -> (match (op1, op2) with
            (StackNte(n), StackInt(i)) -> StackNte(n + (12*i))
          | _ -> raise(Failure("stack corrupt")))
        | Deref -> (match (op1, op2) with
            (StackLst(Int, l, c), StackInt(i)) -> StackInt(List.nth c i)
```

```
        | (StackLst(Bool, l, c), StackInt(i)) ->
                          (match List.nth c i with
                            0 -> StackBln(false) | _ -> StackBln(true))
        | (StackLst(Note, l, c), StackInt(i)) ->
                          (StackNte(List.nth c i))
        | _ -> print_stack stack sp 10; raise(Failure("stack corrupt")))
      | Concat -> (match (op1, op2) with
          (StackInt(i), StackLst(Int, l, c)) -> StackLst(Int, l+1, i::c)
        | (StackBln(b), StackLst(Bool, l, c)) ->
                          (let i = match b with true -> 1 | false -> 0 in
                            StackLst(Bool, l+1, i::c))
        | (StackNte(n), StackLst(Note, l, c)) ->
                          StackLst(Note, l+1, n::c)
        | _ -> raise(Failure("stack corrupt")))
      | _ -> raise (Failure("unimplemented binop")));
      exec fp (sp-1) (pc+1)
  | Dbg     -> print_stack stack sp 5; exec fp (sp) (pc+1)
  | Nop     -> exec fp sp (pc+1)
  | Lod i   -> stack.(sp)   <- globals.(i)   ; exec fp (sp+1) (pc+1)
  | Str i   -> globals.(i)  <- stack.(sp-1)  ; exec fp sp     (pc+1)
  | Lfp i   -> stack.(sp)   <- stack.(fp+i)  ; exec fp (sp+1) (pc+1)
  | Sfp i   -> stack.(fp+i) <- stack.(sp-1)  ; exec fp sp     (pc+1)
  (* len *)
  | Jsr(-1) -> (let l = stack.(sp-1) in match l with
                    StackLst(_, n, _) -> stack.(sp-1) <- StackInt(n)
                  | _ -> raise(Failure("stack corrupt")));
                  exec fp sp (pc+1)
  (* octave *)
  | Jsr(-2) -> (let n = stack.(sp-1) in match n with
                  StackNte(n) when (n <= 2) -> stack.(sp-1) <-
                          StackInt(0)
                | StackNte(n) when (n > 2) -> stack.(sp-1) <-
                          StackInt(((n-3)/12)+1)
                | _ -> raise (Failure("stack corrupt")));
                exec fp sp (pc+1)
  (* pitch *)
  | Jsr(-3) ->  let rec pitchshift i delta = (match i with
                  i when (i < 39 || i > 50) -> pitchshift (i + delta) delta
                | _ -> i) in
                (let n = stack.(sp-1) in match n with
                  StackNte(n) when (n > 50) ->
                          stack.(sp-1) <- StackNte((pitchshift n (-12)))
                | StackNte(n) when (n < 39) ->
                          stack.(sp-1) <- StackNte((pitchshift n 12))
                | StackNte(_) -> stack.(sp-1) <- n
                | _ -> raise(Failure("stack corrupt")));
                exec fp sp (pc+1)
  (* print *)
  | Jsr(-4) -> (let f = stack.(sp-1) in match f with
                  StackInt(i) -> print_endline (string_of_int i)
                | StackSng(s) -> print_endline s
                | StackNte(n) -> print_endline (notestring_of_int n)
                | StackBln(b) -> print_endline (string_of_bool b)
                | StackLst(Bool, n, c) -> print_endline ("[" ^
```

```
                        String.concat ";"
                        (List.map (fun i -> if (i = 0) then "false" else "true") c
                           )
                        ^ "]")
                  | StackLst(Int, n, c) -> print_endline ("[" ^
                        String.concat ";"
                        (List.map string_of_int c) ^ "]")
                  | StackLst(Note, n, c) -> print_endline ("[" ^
                        String.concat ";"
                        (List.map notestring_of_int c) ^ "]")
                  | _ -> raise (Failure("illegal argument to print")));
                  exec fp sp (pc+1)
  (* reverse *)
  | Jsr(-5) -> (let l = stack.(sp-1) in match l with
                         StackLst(t, l, c) ->
                                stack.(sp-1) <- StackLst(t, l, List.rev c)
                      | _ -> raise(Failure("stack corrupt")));
                      exec fp sp (pc+1)

  | Jsr i    -> stack.(sp)    <- RetAddr(pc+1)     ; exec fp (sp+1) i
  | Dup      -> stack.(sp) <- stack.(sp-1); exec fp (sp+1) (pc+1)
  | Efr      -> stack.(sp) <- StackSng("stub"); exec fp (sp+1) (pc+1)
  | Swp      -> let t1 = stack.(sp-1) and t2 = stack.(sp-2) in
                stack.(sp-2) <- t1; stack.(sp-1) <- t2; exec fp sp (pc+1)
  | Ent i    -> stack.(sp)    <- StackPos(fp)      ; exec sp (sp+i+1) (pc+1)
  | Rts i    -> let new_fp = stack_pos stack.(fp) and
                    new_pc = ret_addr stack.(fp-1) in
                stack.(fp-i-1) <- stack.(sp-1) ; exec new_fp (fp-i) new_pc
  | Beq i    ->
      exec fp (sp-1) (pc + if stack.(sp-1) = StackBln(true) then i else 1)
  | Bne i    ->
      exec fp (sp-1) (pc + if stack.(sp-1) = StackBln(false) then i else 1)
  | Bra i    -> exec fp sp (pc+i)
  | Hlt      -> ()

  in exec 0 0 0
```

8.7 nc.ml

Listing 11: nc.ml

```
type action = Ast | Bytecode | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [
            ("-a", Ast);
                                ("-b", Bytecode);
                                ("-c", Compile) ]
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
```

```
      | Ast -> let listing = Ast.string_of_program program
             in print_string listing
    | Bytecode -> let listing =
        Bytecode.string_of_prog (Compile.translate program)
      in print_endline listing
    | Compile -> Execute.execute_prog (Compile.translate program)
```