

# The Northern Lights Programming Language and Design Principles of Domain Specific Languages

William Harris, Zachary Tatlock

May 2, 2006

## Overview

Northern Lights, hereafter referred to simply as NL, is a project dedicated to allowing users regardless of location to manipulate a structure of lights. Although the project is designed in such a way that almost any collection of lights can be manipulated, the structures around which the project is centered are the ones adorning the Wheeler Building in Indianapolis and Purdue's Lawson Computer Science Building. NL can be thought of as a complete system, governing data flow from user input all the way to the final output. The data flow can be summarized as follows: obtain user input in the form of a program written in NL's own domain specific language, queue the data for execution, compile the code for execution into a binary, and finally execute the code, with the output from the binary being directed over a parallel port onto the structure of lights.

## System

### User-Input

The user is presented with an online version of the NL interpreter as a Java applet. The applet consists of a field through which the user enters the code, means of starting and stopping execution of the program, a field for displaying compilation and run-time errors, and a list of rectangles that can alternate between blue and gray coloring, serving as a representation of the light structure. When the user executes their code, the effects are then

simulated on the applet's simulated light structure. After testing is complete, the code can then be submitted through the applet.

## Queueing

Queueing is a relatively simple process. From the online emulator, the user submits their identification and their code. The code is then stored on the server in a "nl" file. The name of the file is a combination of the identification and the time of submission, thus guaranteeing a unique file name. On the server, a task scheduled to execute approximately every five minutes stops the program that is currently running and copies any new code to the machine driving the light structure. The code is then compiled into a C program, and a simple shell script is written and started that executes the program infinitely, to be stopped upon the next execution of the maintenance script.

## Execution

The C program is two *for* loops, one nested within the other, that output the array of Patterns one light at a time over the parallel port. The inner *for* loop gets the current light value in the pattern, sends the data over the port, sets the clock high to confirm that the data has been sent, and finally sets the clock low to prepare for sending the next byte of data. After an entire Pattern has been output, the latch is set high and the program then sleeps for the duration specified.

## The NL Programming Language

It was decided that a domain-specific language would be ideal in providing the programmer with the specific tools they needed to perform their job, while not overwhelming them with the flexibility inherent to a general-purpose language. Thus was born NL, a language designed specifically to allow the user to program a light structure.

## Data Types

NL is strongly-typed, containing only two built-in data types, integer and Pattern, and not allowing user-defined types. The integer data type hardly needs an introduction, although it should be noted that it is the same size as an integer in Java. The Pattern data type, however, requires a bit of explanation. A Pattern is analogous to Java's String class; essentially it is

an array of lights that can be either on or off, so it is essentially an array of boolean objects. One of the difficulties in developing an object that represents an entire structure of lights is designing a simple, effective way for users to specify pattern literals. NL provides three options for this:

1. Literal - the user specifies a pattern in double-quotation marks. The resulting pattern is of the length that the user specified, but if it is ever output, all unspecified lights will be assumed to be off.
2. Repetition - the user specifies a pattern in single-quotation marks that is repeated until a pattern the length of the device is created.
3. Scaling - the user specifies a pattern between pairs of three single quotation marks. The resulting pattern is scaled to form a pattern the size of the light structure. Thus, for a pattern with  $n$  objects specified for a device of length  $m$ , each light in the original pattern is copied to form  $m/n$  lights in the new pattern.

## Control Flow Structures

Control flow structures include ones that a programmer would expect to find in any general purpose language. These include *if – else* statements, *while* and C-style *for* loops, and functions that can return either *void* or either of the two data types. Syntax was kept similar to C and Java whenever possible.

## The Compilation Process

### Lexical Analysis

Users can specify either integer or Pattern literals as given by the rules outlined in the section *Data Types*. Variable names follow the standard of a letter followed by any sequence of alphanumeric characters and underscores. Lexical analysis generates a representation of the program as a series of tokens; each token stores what type of data it represents (a variable name, an integer literal, etc.), along with storing the relevant information about that token.

### Syntactical Analysis

An NL program can be seen as a collection of functions, with execution starting at a function *main()* written by the user. Each function holds the actual

sequences of instructions that constitute the program. Syntactical analysis processes the list of the tokens, generating a tree representation of the program. This tree is known as an Abstract Syntax Tree, often abbreviated as an AST. The root of the tree is a simply a node that denotes a program, containing a list of function nodes. Subsequently, each function node contains more specific information about its contents.

## Semantic Analysis

Semantic analysis takes as input the Abstract Syntax Tree generated by syntactical analysis and verifies the correctness of a program. This is essentially done through type checking. Type checking makes sure that if an expression is assigned to a variable, the type of the value returned by that expression matches the type of the variable. It also verifies that for any binary operation  $*$  between two objects  $a$  and  $b$ ,  $*$  is defined for an object of type  $a$  and an object of type  $b$ .

## Interpretation

Guaranteed that the program is now valid, it is now time to execute it. NL's interpretation process is in a sense similar to its semantic analysis. Semantic analysis traverses an AST, determining that the type to which a given node evaluates. Similarly, the interpreter traverses an AST, but instead of a type, it returns a computed value. Given NL's limited data types, this object will be either an integer or a Pattern.

## Criticisms

While the NL language is fairly simple for the power of expression that it gives to the programmer, it is doubtful that a person new to programming would be willing to invest the time necessary to understand the language and write interesting programs. If it is to achieve widespread success, NL will require a graphical frontend alongside the programming one currently available. This graphical frontend will allow users to generate pattern sequences by utilizing graphical tools, including an editable representation of the light structure, to generate programs.

## References

- [1] Appel, Andrew W., *Modern Compiler Implementation in Java*. Cambridge University Press, United Kingdom, 2nd Edition, 2002.