NOPEL Language Reference Manual
Daniel T Faltyn Jr. dtf2110
W4115 Spring06

This manual describes the NOPEL (Network Oriented Programming Language) lexical conventions and syntax notation. NOPEL syntax is very similar to C but there are some subtle differences especially surrounding functions.

```
// A word on notation:  Sections like this in grey w/ a dashed border are code snippets.
```

1 Lexical Clambakes (Conventions)

A NOPEL program consists of a single file that may have network functions (functions that should be available over a network) and zero or more local functions that are visible to other functions (including network functions) in the same file. These functions and various global variables are all parsed into a stream of tokens as described in the following section.

1.1 Tokens

The categories of tokens in NOPEL are comments, identifiers, keywords, numbers, strings, and dates. The NOPEL lexer ignores white space, new lines and tabs except where they separate tokens. The NOPEL lexer similarly recognizes and ignores comments and various separators such as commas, curly, and square braces.

1.1.1 Comments

Comments in NOPEL come in the usual 2 flavors: single line and multiple lines. Single line comments begin with a double slash and terminate with a new line. Multiple line comments begin with a slash star and terminate with a star slash.

```
// this is a single line comment in NOPEL. It terminates with a new line.
/* this is a multi-line comment in NOPEL. It can be on one line */
/* or it can span
   multiple lines */
```

1.1.2 Identifiers

An identifier is a sequence of letters, digits and the underscore character beginning with an underscore or a character. Case is significant so the identifier *myVariable* is distinct from *MyVariable*.

```
int g_globalVar = 10; // the identifier is g_globalVar and holds the integer val 10
```

1.1.3 Keywords

The NOPEL language is translated into Java so all Java keywords[1] are also reserved. The following keywords are explicitly defined in NOPEL and programmers may not use them other than for their intended usage.

function	network	date	true	false
if	else	while	break	continue
return	returns	struct	void	LOG_FATAL
LOG_INFO	LOG_WARN	LOG_ERROR	LOG_DEBUG	

1.1.4 Numbers

The NOPEL lexer recognizes Integers as a sequence of one or more digits and Real numbers as one or more digits followed by a "." followed by one or more digits. The following are various examples of NOPEL numbers. All numbers in NOPEL can have an optional prefix "-" that indicates the number has a value less than zero.

```
1 // is an integer
2222 // is an integer
2.3 // is a real
222.2222 // is a real
-20 // negative integer
```

1.1.5 Strings

The NOPEL lexer recognizes strings as a sequence of characters surrounded by double quotes. Commonly escaped characters t, b, n, \, and the double quote are allowed as long as they are prefixed with a \.

```
"This is a string with an escaped backslash as the last character \\"
```

[1] See the appendix for a complete list of java key words.

1.1.6 Dates

The NOPEL lexer recognizes dates as **ddMMMyyyy** where d and y are digits and M is an alphabetic character. The first 2 digits represent the day of the month, the 3 characters are the first 3 characters of any of the 12 months, and the final 4 digits are the year.

The Lexer does not differentiate the case for the month characters so 12FEB2006 and 12Feb2006 are equivalent. The Lexer does not accept dates that include invalid days of the month (e.g. 45Mar2006), bad months (e.g. 05ZOR2006), or abbreviated years (e.g. 23Mar06.)

```
02Mar2006 // this is a valid date.
```

1.1.7 Convenience Tokens

NOPEL recognizes these other common tokens for the sake of punctuation and clarity.

Type	Comment
{ }	Demarcates the beginning and end of a function or structure
[ ]	Demarcates the beginning and end of an array
;	Separates expressions
,	Separates parameters in functions

1.2 Syntax Notation

NOPEL is a statically typed language in which each object has a declared immutable type. Programmers can use these types using common programming expressions, statements, and user defined functions. (A thorough discussion of functions is addressed in its own section.)

```
// A word on notation:  Sections like this in light blue with a solid border are syntax
```

1.2.1 Types

There are three classifications of types in NOPEL: primitives, structures, and arrays.

1.2.1.1 Primitive Types

NOPEL is ultimately compiled into Java so the following table lists the corresponding java type, the default value, and an example declaration/assigment.

Type	Java conversion	Default value	Usage
boolean	boolean	false	boolean isANiceDay = true;
int	int	0	int age = 29;
real	double	0.0	real weight = 180.22;
string	java.lang.String	“”	string name = “Daniel”;
date	java.util.Date	todays date[2]	date today = 18Feb2006;

[2] The date returned is the date as of the running of the program. I.e. the default date is determined at run time and changes every day. (This may be a bad idea.)

1.2.1.2 Structures

A structure is a grouping of named primitive types into a single object that can be referenced as a unit. The declaration of a structure is achieved though the keyword struct, a left curly, a declaration of one or more typed fields, and a right curly brace.

A NOPEL program must declare a structure before usage. Programs can reference fields in the structure with the usual "dot" notation of C.

```
struct
{
        ((boolean | int | real | string | date) identifier ;)*
}
```

This is an example declaration of a structure that contains information about a trader followed by an example usage.[3]

```
// declaration of the structure type trader
struct trader
{
   string firstName;
   string lastName;
   int age;
   date startDate;
   string desk;
};
trader bob; // instance of type trader
bob.firstName = "Robert";     // assigning the value of the field firstName
int age = bob.age;            // retrieving the value of the field age.
```

1.2.1.3 Arrays

NOPEL provides the means to group primitive types and structures into typed arrays of a fixed size. Arrays have the property **length** that indicates the size of the array as an integer. Programs can retrieve values from an array by accessing the values indexed from 0 to length -1 as shown.

```
(boolean | int | real | string | date) identifier "[" int "]"";"
```

```
int accountNumbers[10]; // create an array of size 10 (default values initialized to 0)
int second = accountNumbers[1]; // get the second element of the array
int arrayLen = accountNumbers.length; // returns the integer 10.
```

[3] My underlying implementation is Java so I'm not sure at this point if structs will be converted to a specific Java object whereby I can access each field on the class, or if I will have one generic class which has a simple HashMap field and one getter and setter. If the final implementation is the latter, accessing/mutating fields may be done via trader.get("firstName"); and trader.set("firstName", "Robert")

1.2.2 Expressions

Expressions in NOPEL are categorized as identifiers, constants, structure usage, array usage, and function calls separated by a semi-colon. Parenthesized expressions take on the value of the enclosed expression.

1.2.2.1 Identifiers

Identifiers are variables that derive their value via the assignment operator, thus making them left-value expressions. In cases where assignment is not performed before evaluation of the variable, some default value is assigned to the identifier. Since every object in NOPEL must have a type, the type of the right-hand value must correspond to the declared type.

```
type identifier = <right-hand value>;
```

```
int g_globalVar = 10;  // declaration of a variable and an initial value.
trader bob;            // declaration of a variable bob of type trader (defined previously)
int years;             // declaration of a variable with the default value.
```

1.2.2.2 Constants

Numbers (ints and reals), strings, and dates are all right-value expressions that evaluate to their innate value.

1.2.2.3 Function calls

A function call is a right-value expression that takes zero or more parameters and returns a declared type or possible void (no value.) See a later section for more information on functions.

```
type identifier = func-identifier(<param-list>);
```

In the following example the beta function is calling the alpha function and assigning the resultant value to the variable someName.

```
// declaration of a function
function alpha(string name) returns string { return “Hello: “+ name; };
// calling function alpha from function beta.
function beta() { string someName = alpha(“Bob”); };
```

1.2.2.4 Arithmetic

NOPEL supports the mathematical expressions +, -, *, and / for integer and real data types using infix notation. If an integer and real data type are mixed in the expression, the resultant data type is real. The usual order of precedence is in effect, namely * and / are evaluated before + or -.

The + operator is available as an infix operator between a string and any other NOPEL primitive type. When one of the operands is not a string, the resultant data type is a string regardless of the placement of the non-string operand.

The + operator is also available as an infix operator between a date and an int. The resultant data type is date such that the value of the given date is incremented by the value of the int.

Example usages of arithmetic operators are as shown:

```
real j = 0.2;
int i = 10;
// adding a real and an int (int id converted to a real)
real k = i + j; // value is 10.2

// adding (concatenating) a string and an int
string myString = “This is:”;
string myStringAsPrefix = myNumber+ i; // results in the string “This is 10”
string myStringAsSuffix = i + myString; // results in the string “10This is “

// adding an int and a date
date tomorrow = 23Jan06 +1; // tomorrow has the value 24Jan06
```

1.2.2.5 Relational

The usual binary operators, “<”, “>”, “<=”, “>=”, “==”, “!=” are available on int and real data types as an infix operator. When comparing objects of different types, int’s are converted to real’s. The resultant value is a boolean.

Equal (==) and not equal (!=) can be applied as infix operators on any data type. When 2 objects of different types are compared, the resultant value is always false for “==” and true for “!=” (except as noted above for integers and reals.) When 2 objects are of the same type, if the value is the same then the result is true for “==” and false for “!=”. With structure objects, if all the fields have the same values then the result is also true for “==” and false for “!=”

1.2.2.6 Logical

The logical operators “!” (not,) “&” (and,) and “|” (or) are available as infix operators when comparing expressions that result in a boolean value. The order of precedence from high to low is “!”, “&”, “|”.

1.2.3 Basic Statements

The types of statements in NOPEL are assignment, if-then-else, while loops and functions. The first three categories are very similar in syntax to C or Java and are addressed in this section.

1.2.3.1 Assignment

Identifiers are left-value expressions that take on a value through the assignment operator “=”. Casting is not available in the NOPEL language so only like values on the left and right side of the “=” are allowed.

```
type identifier = expression;
```

1.2.3.2 If-Then-Else Conditionals

If-Then-Else conditionals determine the flow of control based on a logical expression. The first execution path is entered if the expression evaluates to true otherwise if the second case "else" is specified, that execution path is entered. Similar to C, else's are assigned to the closest else-less if[4].

```
if ( logical )
{
        (expression; | statement )*
} else {
        (expression; | statement )*
}
```

In the following example "out" has the value of 9 at the end of the execution of the conditional. If the value of "in" had any other value than 10, out would have the value 11.

```
int in = 10;
int out = 0;
if ( 10 == in )
{
  out = 9;
} else {
  out =11;
}
```

1.2.3.3 While Loops

For simplicity, there is a single iterative statement "while" that repeats a given set of expressions and statements until some condition is no longer true. The syntax for a while loop is:

```
while ( conditional )
{
        (expression | statement | break | continue)*
}
```

The reserved words break and continue have the following effect when placed between { and } in a while statement:

- **break --** causes the while loop to exit and execution to continue after the }
- **continue –** causes the execution of the loop to stop at this point and resume after the { in the while declaration.

[4] As discussed in lecture an else-less if is an elusive creature but for easier identification I have obtained a photo and attached it as an appendix.

1.2.4 Functions

The two classifications for functions, local and networked, share the same syntax apart from the keyword *network*. Networked functions are available in a remote context, i.e. available for usage over a network when ultimately compiled and run as a Java program.

Given the special nature of functions marked *network*, no other function, local or remote may reference these functions. To promote reuse in the language however, networked functions may reference any local function.

Each function declaration begins with the keyword function, the name of the function, a parenthesized list of zero or more arguments, the keyword *returns* and the return type, possibly *void*. The implementation of the function begins with "{"and is terminated with "}"

1.2.4.1 Local

The syntax for a local function is generalized as:

```
function identifier(type identifier (,type identifier)* | )
        returns (type | void)
{
        (expression; | statement;)*
        return (identifier | constant | );
}
```

An example implementation of a function is shown here.

```
// a local function
function getAgeLocal( date _birthDate ) returns int {
                // implementation
}
```

1.2.4.2 Networked

A network function is identical in syntax to a local function with the addition of the keyword *network*.

```
network function identifier(type identifier (,type identifier)* | )
        returns (type | void)
{
        (expression; | statement;)*
        return (identifier | constant | );
}
```

An example implementation of a network function is as shown here.

```
// networked function that calls a local function
network function getAge( date _birthDate ) returns int {
                retrun getAgeLocal(_birthDate);
}
```

1.2.5 Servers

The top level construct in NOPEL is a Server. The declaration of a server consists of the keyword *server* an identifier used as the name for the server and the implementation enclosed in { and }. The body of a server consists of global variable (possibly assigned a value, and zero or more network and local functions. All functions and variables must be defined before use.

```
Server identifier
{
        (identifiers | assignment statment)*
        (local functions; | network functions)*
}
```

The following is a complete NOPEL program.

```
server HelloWorld
{
int countSaidHiTo = 0; // global variable to keep track of usage

// local function for reuse
function localHello( string _name ) returns string {
                countSaidHiTo = countSaidHiTo + 1;
                return “Hello “ +_name”;
}
network function sayHello( string _name ) returns string {
                LOG_INFO (“starting sayHello with param:” + _name);
                string out = localHello( _name );
                LOG_INFO (“I said hi to “+ countSaidHiTo +”people so far.”);
                return out;
}
```

1.2.6 Additional functions

Most languages allow the use of some access to system out. NOPEL uses categorized logging based on the priority of the message. The basic implementation will print out to the console when the application is run. In the future, configurations should allow the redirection of the output to files, sockets, etc. and filtering based on the criteria. The syntax for logging is as shown. Further, the priority of the logging message is listed from least important to highest importance.

```
(LOG_DEBUG | LOG_INFO | LOG_WARN | LOG_ERROR | LOG_FATAL) (string);
```

In this example the first logging message prints out the in parameter before the execution of the local function and the second logging message prints the result of the local function.

```
network function getAge( date _birthDate ) returns int {
                LOG_INFO (“starting getAge with param: “ + _birthDate);
                int out = getAgeLocal(_birthDate);
                LOG_INFO (“ending getAge with result: “ + out);
                retrun out;
}
```

1.2.7 Appendix A Java Keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	If	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	Int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	Super	while

1.2.8 Else-less if