Noodl™

a negligible-overhead object description language for MicroCosm™
by
Chip Morningstar

Lucasfilm Ltd. Games Division
September 26, 1985

INTRODUCTION

This document describes **Noodl™**, the **N**egligible **O**verhead **O**bject **D**escription **L**anguage, which provides a convenient format for defining object class descriptions to be run under **Looi™**, the **L**ow **O**verhead **O**bject **I**nterpreter in the **MicroCosm™** host database system. The purpose of this document is to try to get it all straight before we get bogged down in implementation.

WHAT IT IS

Noodl is a simple **C** preprocessor that adds an extra layer of ''syntactic sugar'' for the definition of the arrays, structs and functions required to specify **Looi** objects.

OBJECT CLASS DEFINITIONS

Input to **Noodl** consists of two sorts of things: blocks of **C** statements

```
{
        any-C-statements-you-choose
}
```

and object class definitions

```
class class-name ( property-names )
        capability-definition
        capability-definition
        ...
        capability-definition
!
```

where *class-name* is an identifier to use as the name of the object class being defined. *Property-names* is a comma-separated list of identifiers to be used as the names of the object's properties (i.e., the elements in the object's property vector). These properties are in addition to the four essential properties possessed by all objects: *class*, *owner*, *location*, and *image*. The list of property names may be empty, in which case no extra properties are defined. *Capability-definition* is (surprise!) a capability definition (described below).

CAPABILITY DEFINITIONS

A capability definition specifies one capability of an object. Its form is

```
capability-name ( arguments ) {
        any-C-statements-you-choose
}
```

where *capability-name* is an identifier chosen to be the name of the capability. This need be unique only

within the context of the particular object definition being given (i.e., different object classes can have capabilities with the same name). *Ar guments* is a comma-separated list of identifiers to be used as the symbolic names of the arguments to the capability procedure that will be generated. The body of the capability definition can be any arbitrary group of **C** statements, as long as it does not include any function definitions (since it will itself become the body of a **C** function).

WHAT IT DOES

What **Noodl** does with all of this is generate some `struct` declarations and some compilable functions.

CLASS DESCRIPTOR TYPE

The ultimate output of **Noodl** is **C** code to initialize instances of a class descriptor `struct`. For reference, here is this `struct`'s definition:

```
typedef struct {
    capabilityVectorType *capabilityVector;
    int                   capabilityCount;
    int                   propertyCount;
} classDescriptorType;
```

CAPABILITY VECTOR TYPE

The class descriptors are initialized with capability vectors, whose type is defined as:

```
typedef requestResultType (*capabilityVectorType[])();
```

This peculiar definition translates into english as ‘‘array of pointers to functions returning `requestResultTypes`’’, i.e., an array of capability procedures. `requestResultType` is the data type of all results passed back by capability procedures.

PROPERTY VECTOR TYPES

Noodl defines a separate property vector data type for each class of object. However, they all have essentially the same form:

```
typedef struct {
    int     class;
    int     location;
    int     owner;
    int     image;
    other properties as defined
} objectNamePropertyVectorType;
```

Yes, we define it internally as a `struct` even thought we've been calling it a ‘‘vector’’ all along.

PUTTING IT ALL TOGETHER

The following example illustrates what happens when we put all these definitions and so on together and run it through **Noodl**:

```
{
    C-stuff-1
}

class foo (zip, zap, zim, zam)
```

```
        snoz (arg1) {
            C-stuff-2
        }
        sneez () {
            C-stuff-3
        }
        snuuuz (arg1, arg2, arg3) {
            C-stuff-4
        }
!

class bar (oop) {
        wop () {
            C-stuff-5
        }
        bop (foo) {
            C-stuff-6
        }
!
```

which translates into

```
C-stuff-1

typedef struct {
        int     class;
        int     location;
        int     owner;
        int     image;
        int     zip;
        int     zap;
        int     zim;
        int     zam;
} fooPropertyVectorType;

requestResultType foo_snozCapability(object, capabilityNumber, arg1)
        int     object;
        int     capabilityNumber;
        int     arg1;
{
        C-stuff-2
}

requestResultType foo_sneezCapability(object, capabilityNumber)
        int     object;
        int     capabilityNumber;
{
        C-stuff-3
}

requestResultType foo_snuuuzCapability(object, capabilityNumber, arg1,
                        arg2, arg3)
        int     object;
        int     capabilityNumber;
```

```
        int     arg1;
        int     arg2;
        int     arg3;
    {
        C-stuff-4
    }

    capabilityVectorType fooCapabilityVector = {
        foo_snozCapability,
        foo_sneezCapability,
        foo_snuuuzCapability,
    };

    classDescriptorType fooClass = {
        fooCapabilityVector,
        3,
        4
    };

    typedef struct {
        int     class;
        int     location;
        int     owner;
        int     image;
        int     oop;
    } barPropertyVectorType;

    requestResultType bar_wopCapability(object, capabilityNumber)
        int     object;
        int     capabilityNumber;
    {
        C-stuff-5
    }

    requestResultType bar_bopCapability(object, capabilityNumber, foo)
        int     object;
        int     capabilityNumber;
        int     foo;
    {
        C-stuff-6
    }

    capabilityVectorType barCapabilityVector = {
        bar_wopCapability,
        bar_bopCapability,
    };

    classDescriptorType barClass = {
        barCapabilityVector,
        2,
        1
    };
```

Clearly, the form before being processed by **Noodl** is quite a bit more concise and expressive than the

output.

CODING OF CAPABILITY PROCEDURES

In addition to using the preprocessor, coding **C** functions under **Noodl** requires that you obey certain conventions. Macros and functions to facilitate the coding of capability procedure bodies are defined in an ''include'' file. The **Noodl** preprocessor automatically inserts a `#include` line for this file at the beginning of its output. Incidently, this file also contains the definitions for the various data types required by the resulting **C** code.

The primary thing that a capability procedure will want to do is access (both for reading and for writing) the properties of an object. Since the property vector is defined as a `struct`, simple references to elements in the property vector may be handled by ordinary `struct` field selection (i.e., the ''`.`'' and ''`->`'' operators in **C**). This requires that the capability procedure have a pointer to the property vector itself. What the procedure gets passed, however, is the object's noid. Thus we need a means of translating between noids and property vector addresses. The call

`propertyVector(`*noid*`)`

takes a noid and returns a pointer to the associated property vector, or `NULL` if there is no such object. One might ask why we perform this mapping inside the capability procedure instead of doing at the top level (where it only needs to be called from one place) and passing the property vector pointer down instead of the noid. The reason is because the translation from one to the other, though we will design it to be fast and simple, will still take a certain amount of time. Some capabilities, however, may not require this mapping at all. By placing the decision to map or not to map in the capability procedure we can skip the overhead of mapping whenever we don't need it.

An important overhead reducing principle we apply is to minimize the number of internal consistency checks and protection barriers maintained by the system. This means that capability procedures are responsible for the security of the objects they are dealing with — nobody is going to stop a capability procedure from violating the rules, so it is up to us, the coders of the capability procedures, to make sure they work right. This adds some risk to the system from bugs and so on, but also enables us to avoid being bogged down in bureaucratic layerings of one protocol on top of another. In order to make this work, however, we must provide an easy means to make various sorts of checks when they *are* required.

There are different ways in which checks can be done, and we will provide macros or functions for several of them. The most elementary of these is a simple predicate: a true-false test of some condition. You merely call the predicate and do something conditionally based on the result. Built on top of predicates are some higher level checks that we call *gates*. A gate consists of a call to a ''pass/fail'' predicate and some canned code to abort the capability procedure (with an appropriate error return) if the result is not a ''pass''. Once the flow of control passes through a gate you can safely assume that a given condition is satisfied thereafter. Gates are so useful that we have defined one for each predicate that can have a pass/fail interpretation. Another sort of check is implemented through a query function which returns the status of some quantity. The value returned can then be tested in any number of ways.

One of the things you might want to check is the class of an object. Internally we use the address of a class' class descriptor as the class identifier. We can get away with this because it's completely internal (class descriptor address never reaches the outside world). The function

`objectClass(`*noid*`)`

returns a pointer to the class descriptor associated with the noid. The predicate

`isClassMember(`*noid*`, `*class*`)`

returns a boolean value indicating whether or not the particular noid represents an object of class *class*, where *class* is a pointer to a class descriptor. The gate

`mustBeClassMember(`*noid*`, `*class*`)`

ensures that the noid belongs to the particular class.

Another thing that needs to be checked frequently is an object's owner. Owners are denoted by player account numbers. The function

`objectOwner(`*noid*`)`

returns the owner of *noid*. The predicate

`isOwnedBy(`*noid*`, `*account*`)`

returns a boolean value indicating whether or not *account* is the owner of the object denoted by *noid*. The gate associated with this is

`mustBeOwnedBy(`*noid*`, `*account*`)`

which ensures that the given player owns the object.

Appendix — Yacc Grammar for Noodl

```
/*
   Noodl: the Negligable Overhead Object Description Language for the
    encoding of host-based object behavior code in the MicroCosm system.

   Kindly refer to MicroCosm design document #3, "Noodl", and MicroCosm
    design document #2, "Looi", for more detailed descriptive prose as to what
    this is all about.  If you don't know already, you needn't bother to read
    further!

   This is the first crack at a Noodl to C translator using Yacc.

   Chip Morningstar, Lucasfilm Ltd., 2-January-1986
 */

%{

#include "noodlTypes.h"

%}

/* Tokens are recognized by the lexer */
%token CLASS     /* the keyword 'class' */
%token OBJECT        /* the keyword 'object' */
%token ENDFILE       /* the end of the input */
%token CCode     /* anything between '{' and '}' */
%token Identifier    /* the usual */
%token Number        /* a decimal integer */

%%

noodl:
        noodlDescription ENDFILE /* no action */
|
        ENDFILE              /* no action */
 ;

noodlDescription:
        noodlThing           /* no action */
 |      noodlDescription noodlThing  /* no action */
 ;

noodlThing:
        CCode
{
    outputCCode($1);
}
 |      objectDefinition
{
    outputObjectListDefinition($1);
}
 |      classDefinition          /* no action */
 ;
```

```
objectDefinition:
        OBJECT identifierList '!'
{
    $$ = $2;
}
 ;

classDefinition:
        CLASS className propertyList
{
    outputPropertyVectorDefinition($2, $3);
}                        capabilities '!'
{
    outputCapabilityVectorDefinition($2, $5);
    outputClassDescriptorDefinition($2, $3, $5);
}
 |      CLASS className propertyList
{
    outputPropertyVectorDefinition($2, $3);
}                       '!'
{
    outputCapabilityVectorDefinition($2, NULL);
    outputClassDescriptorDefinition($2, $3, NULL);
}
 ;

className:
        Identifier   /* default action */
 ;

propertyList:
        nameList/* default action */
 ;

nameList:
        '(' identifierList ')'
{
    $$ = $2;
}
 |      '('        ')'
{
    $$ = NULL;
}
 ;

identifierList:
        Identifier
{
    $$ = buildIdentifierList(NULL, $1, 0);
}
 |      identifierList ',' Identifier
{
    $$ = buildIdentifierList($1, $3, 0);
}
```

```
 ;

capabilities:
        explicitCapabilityDescription
{
    $$ = buildIdentifierList(NULL, $1, EXPLICIT_CAPABILITY);
}
 |      capabilities explicitCapabilityDescription
{
    $$ = buildIdentifierList($1, $2, EXPLICIT_CAPABILITY);
}
 |      implicitCapabilityDescription
{
    $$ = buildIdentifierList(NULL, $1, IMPLICIT_CAPABILITY);
}
 |      capabilities implicitCapabilityDescription
{
    $$ = buildIdentifierList($1, $2, IMPLICIT_CAPABILITY);
}
 ;

explicitCapabilityDescription:
        capabilityIdentifier argumentList body
{
    $$ = outputCapabilityDescriptionExplicit($1, $2, $3);
}
 ;

implicitCapabilityDescription:
        capabilityIdentifier argumentList functionName
{
    $$ = outputCapabilityDescriptionImplicit($1, $2, $3);
}
 ;

functionName:
        Identifier   /* default action */
 ;

capabilityIdentifier:
        capabilityName
{
    $$ = buildCapabilityIdentifier($1, -1);
}
 |      capabilityName ':' capabilityNumber
{
    $$ = buildCapabilityIdentifier($1, $3);
}
 ;

capabilityName:
        Identifier   /* default action */
 ;
```

```
capabilityNumber:
        Number       /* default action */
 ;

argumentList:
        nameList/* default action */
 ;

body:
        CCode        /* default action */
 ;
```