Introduction

The NDF language is used to describe game data.

It is a declarative language that defines objects with properties and relations between them, in order to instantiate the correct data during game execution.

Syntax

Charset

All NDF files shall be encoded in either **UTF-8** or **ANSI restricted to the 7bit table**.

The NDF parser is case-insensitive : 'TRUE' and 'true' are equivalent.

Keywords

`export, is, template, unnamed, nil, private, int, string, true, false, div, map`

Symbols

`//, /, ?, :, =, |, &, <, >, >=, <=, !=, -, +, *, %, , (comma), . (dot)`

Block delimiters

`{}, [], (), <>, (* *), /* */, ' ', " "`

Comments

- Everything after the symbol **//** is ignored until the end of the line
- Everything in blocks delimited by **{ }** or **(* *)** or **/* */** is ignored.

Built-in types

Boolean

Can only be **true** or **false**.

Character strings

Character strings are delimited by single- or double-quotes and can contain accentuated characters.

- `"This is a string"`

- `'This ïs ànothér on€'`

Integer

Integers are written in base 10 or hexadecimal notation.

- `3`
- `150486`
- `-36584`
- `0xFF00A8`
- `0x1`

Floating point

Floating point numbers are written in base 10 natural notation, there is no exponent or hexadecimal notation. The decimal separator is the **.** (dot) character.

- `3.1415954`
- `-9.81`
- `654987.1248`

The integer part can be omitted (implicit 0). The fractional part can be omitted (implicit 0).

- `.127`
- `53.`

Vector

A vector is a list of zero or more elements enclosed in a **[ ]** block and separated by **,** (comma).

- `[] // an empty vector`
- `[1, 2, 3] // a vector of integers`
- `["Hello", 'World',] // a trailing comma separator is accepted`

Pair

A pair is an object containing two inner objects, enclosed in a **( )** block and separated by **,** (comma).

- `(22, 7) // a pair of integers`
- `("Hello", "World") // a pair of strings`

Map

A map is a list of zero or more pairs containing a key and its associated value. It is created by adding the keyword **MAP** before a **[ ]** block.

Bloc de code

```
// A map of integers to strings
MAP[
    (1, 'one'),
    (2, 'two'),
    (3, 'three')
]
```

Objects

Naming a built-in value

One can give names to values of built-in types, akin to creating variables in programming languages, by using the keyword **is**.

Bloc de code

```
Gravity is -9.81
Currencies is MAP[
    ('EUR', '€'),
    ('USD', '$'),
    ('GBP', '£')
]
Places is ['Antarctica', 'Everest', 'Mars']
```

Arithmetic operations

Usual operations on numbers are available : addition, subtraction, multiplication, division, modulo.

Bloc de code

```
Pi is 3 + 0.1415954
PiCube is Pi * Pi * Pi
X is 35 div 8 // Division is used through the keyword 'div'
Y is 35 % 8
```

You can also concatenate strings, vectors and map with the **+** operator.

Bloc de code

```
A is [1, 2] + [3, 4]
B is MAP[ (1, 'one')] + MAP[ (2, 'two')]
C is "Hello" + " World!"
```

Object definition

Besides built-in values, NDF allows to create and use complex objects.

An object is defined by its name and type, and can contains member values. Types (almost ?) always start with a capital 'T'. They represent game's internal data-structures and their definition is not available.

When loading data, the game will create objects of the desired type and populate its fields with the member values filled in the NDF description.

Bloc de code

```
// This will create an instance of TExampleType containing two members
ExampleObject is TExampleType (
    MemberInteger = 12
    MemberString = "something"
)
```

If a type definition contains a member value that is not filled in the NDF definition of the object, it will default to a base value. This will often be 0 for numbers, false for booleans, and empty for containers (strings, vectors, maps).

Types can have other types as member, resulting in "nested objects".

Bloc de code

```
ExampleObject is TExampleType
(
    innerObject = TOtherType
    (
        ValueString = "I am a member of TExampleType"
    )
)
```

Objects can be defined without a name with the keyword **unnamed**, they are called unnamed objects. Unnamed objects can only be defined as "top objects", meaning that they are not a member of another object.

Bloc de code

```
unnamed TExampleType
(
    ValueString = "I am an unnamed object"
)
```

Namespaces

Every definition of an object, be it named or not, creates a namespace from its name. We'll use the **$/Namespace1/Namespace2/Object** notation for absolute names of objects.

First example

Bloc de code

```
ExampleObject is TExampleType
(
    InnerExample = ExampleObject2 is TExampleType
    (
        InnerExample = ExampleObject3 is TExampleType()
    )
)
```

In that example, the absolute name of **ExampleObject3** is **$/ExampleObject/ExampleObject2/ExampleObject3**.

Info

Note that objects assigned to members can also be named, as it's the case for ExampleObject2 and ExampleObject3

Second example

Bloc de code

```
ExampleObject is TExampleType
(
    InnerExample = TExampleType
```

```
    (
        InnerExample = ExampleObject3 is TExampleType()
    )
)
```

Here, the absolute name of ExampleObject3 is **$/ExampleObject//ExampleObject3** (note the double slash).

Third example

Bloc de code

```
ExampleObject is TExampleType
(
    InnerExample = _ is TExampleType
    (
        InnerExample = ExampleObject3 is TExampleType()
    )
)
```

Finally, in this case, the absolute name of **ExampleObject3** is **$/ExampleObject/ExampleObject3**. The token _ (underscore) acts as a special name that introduces no namespace.

Referencing objects

A reference is a label corresponding to the name of an object. A reference can be null, and is in that case equal to **nil**.

Some types have references as members, because they need to be able to access other objects to work.

Bloc de code

```
DataHolder is TDataHolder
(
    SomeInt = 456
    SomeString = "A string"
    SomeMap = MAP[ {...} ]
)

DataUser is TDataUser
(
    Condition = false
    DataHolderReference = DataHolder // Here we are taking a reference to the
DataHolder object
)
```

In the above case, the first object named **DataHolder** found will be taken as a reference. There are other ways of referencing objects, depending on their scope. Without going into details, a lot of other NDF files invisible to the modder are loaded during start-up, and moddable files may need to reference them.

This is where other types of reference come into play :

Bloc de code

```
$/Path/To/OtherObject // absolute reference
~/Path/To/OtherObject // reference from the loading namespace (can't be known by the
modder)
./Path/To/OtherObject // reference from the current namespace
```

Prototypes

A prototype is a regular object that will be used to create another object from it. Any named object can serve as a prototype.

Creating an object from a prototype will make a copy of the prototype into the new object, and allow to override some members at will.

Bloc de code

```
Prototype is TThing
(
    ValueString = 'I am a prototype object'
    ValueInt = 666
)

ObjectFromPrototype is Prototype
(
    ValueString = 'I am just me'
)
```

ObjectFromPrototype has a member **ValueInt = 666** that is copied from **Prototype**, and overrides its **ValueString** member.

Templates

Templates allow to generate objects from a parameter list in a generic manner. They are close to macros in the C-family languages.

They are defined by the keyword **template**, a name, the parameter list enclosed in a **[ ]** block, and then the template body.

Bloc de code

```
// Basic template that does nothing
template MyTemplate // template name
[]                  // template parameters
is TType            // final type of the object created by the template
()                  // object members
```

The parameter block defines a list of parameters, separated by **,** (comma).

The syntax for parameters is **%name% [: %type%] [= %default%]**. The name is mandatory, whereas :

- an optional type can be specified, otherwise it will be deduced from the context
- an optional default value can be specified, otherwise it **must** be provided when using the template

In the template body, template parameters are used by their name enclosed in **< >**.

Bloc de code

```
// More elaborated example
// Suppose there is a TWeapon and TCharacter types existing.

Axe is TWeapon
(
    // here would be members describing this weapon
)

Knife is TWeapon ()

template Character
```

```
[
    Name : string, // Name has to be a string
    Level : int = 1, // Level has a default value of 1
    Weapon : TWeapon = TWeapon() // Weapon has to be a TWeapon
]
is TCharacter
(
    Name = <Name> // <Name> refers to the template parameter named 'Name'
    Level = <Level>
    HP = <Level> * 100
    Weapon = <Weapon>
    Damages = <Weapon>/Damages * (1 + Level div 10)
)

Hero is Character
(
    Name = "Hero"
    Level = 12
    Weapon = Axe
)

Creep is Character
(
    Name = "Creep"
    Weapon = Knife
    // Level is not specified, default value is used
)
```

Advanced templates

Scoped objects

Objects can be declared inside of template bodies.

Bloc de code

```
template Character
[
    Name : string,
    Level : int = 1,
    Weapon : TWeapon = TWeapon()
]
is TCharacter
(
    Bag is TInventory
    (
        MaxItemCount = <Level> * 3
    )

    Name = <Name>
    Level = <Level>
    HP = <Level> * 100
    Weapon = <Weapon>
    Damages = <Weapon>/Damages * (1 + Level div 10)
    Inventory = Bag
)
```

Template template

A template can derive from another template.

Bloc de code

```
template CloneHero
[
    Name : string
]
is Character
(
    Name = <Name>
    Level = 12
    Weapon = Axe
)
```

A derived template can also override scoped objects.

Bloc de code

```
template CloneHero
[
    Name : string
]
is Character
(
    Name = <Name>
    Level = 12
    Weapon = Axe

    Bag is TInventory ( MaxItemCount = 0 )
)
```