Naggum Documentation

Release 0.0.1

Naggum authors

Aug 25, 2019

Contents

CHAPTER 1

About

1.1 Disclaimer

Naggum doesn't aim to be yet another Common Lisp or Scheme or whatever implementation. Instead, we are trying to deliver a modern Lisp dialect that makes use of most of CLI benefits.

1.2 Features

Naggum provides both direct access to low-level features of CLI, and allows its user to define types and functions just like any other high-level language. At the same time, it combines power of Lisp-inspired metaprogramming (macros) with strictness of strong-typed language.

1.3 Contribute

- Source Code: https://github.com/codingteam/naggum
- Issue Tracker: https://github.com/codingteam/naggum/issues

1.4 License

CHAPTER 2

Build guide

To use Naggum, first of all you need to build it from source.

2.1 Windows

To build Naggum on Windows, just use Visual Studio or MSBuild like that:

```
$ cd naggum
$ nuget restore
$ msbuild /p:Configuration=Release Naggum.sln
```

2.2 Linux

See general build instructions for Linux in the file `.travis.yml` inside the Naggum source directory.

You'll need Mono, NuGet and F# Compiler installed. Some of them may or may not be part of your Mono installation; just make sure you've got them all.

Please note that currently the project is compatible with Mono 4.4.2+.

Below is an example of setting up these tools on NixOS Linux; feel free to add instructions for any other distributions.

2.2.1 NixOS Linux

The instructions have been verified on NixOS 16.03. If something doesn't work, please file an issue.

Enter the development environment:

```
$ cd naggum
$ nix-shell
```

After that you can download the dependencies and build the project using `xbuild`:

```
$ nuget restore
$ xbuild /p:Configuration=Release /p:TargetFrameworkVersion="v4.5"
```

After that, you can run `Naggum.Compiler`, for example:

```
$ cd Naggum.Compiler/bin/Release/
$ mono Naggum.Compiler.exe ../../../tests/test.naggum
$ mono test.exe
```

2.3 Documentation

You can build a local copy of Naggum documentation. To do that, install Python 2.7 and Sphinx. Ensure that you have `sphinx-build` binary in your `PATH` or define `SPHINXBUILD` environment variable to choose an alternative Sphinx builder. After that go to *docs* directory and execute `make html` (on Linux) or `.\make.bat html` (on Windows).

CHAPTER 3

Naggum usage

Currently there are two dialects of Naggum: high-level *Compiler* and low-level *Assembler*.

3.1 Naggum Compiler

Command line syntax for Naggum Compiler is:

```
$ Naggum.Compiler source.naggum... [/r:assembly]...
```

Each input source file will be compiled to a separate executable assembly (i.e. an `.exe` file) in the current directory. You can also pass a list of files to be referenced by these assemblies.

`.naggum` extension is recommended for high-level Naggum files.

3.2 Naggum Assembler

Naggum Assembler uses low-level Naggum dialect. Command line syntax is:

```
$ Naggum.Assembler source.nga...
```

Each input file may contain zero or more assembly constructs. Every assembly will be saved to its own executable file in the current directory.

`.nga` extension is recommended for low-level Naggum files.

3.3 S-expression syntax

Each Naggum program (either high-level or low-level) is written as a sequence of S-expression forms. In s-expression, everything is either an atom or a list. Atoms are written as-is, lists should be taken into parens.

Possible atom values are:

```
"A string"
1.4e-5 ; a number
System.Console ; a symbol
```

A symbol is a sequence of letters, digits, and any of the following characters: `+-*/=<>!?.`.

Lists are simply sequences of s-expressions in parens:

```
(this is a list)

(this (is ("Also") a.list))
```

Naggum source code may also include comments. Everything after `;` character will be ignored till the end of the line:

```
(valid atom) ; this is a comment
```

3.4 Low-level syntax

Naggum low-level syntax is closer to CIL. It may be used to define CLI constructs such as assemblies, modules, types and methods. Every `.nga` file may contain zero or more assembly definitions.

3.4.1 Assembly definition

Assembly defitinion should have the following form:

```
(.assembly Name
    Item1
    Item2
    ...)
```

Assembly items can be methods and types. Top level methods defined in an `.assembly` form will be compiled to global CIL functions.

Type definitions are not supported yet.

Each assembly may contain one entry point method (either a static type method or an assembly global function marked by `.entrypoint` property).

3.4.2 Method definition

Method definition should have the following form:

```
(.method Name (argument types) return-type (metadata items)
    body-statements
    ...)
```

Method argument and return types should be fully-qualified (e.g. must include a namespace: for example, `System.Void`).

The only supported metadata item is `.entrypoint`. It marks a method as an assembly entry point.

Method example:

```
(.method Main () System.Void (.entrypoint)
    (ldstr "Hello, world!")
    (call (mscorlib System.Console WriteLine (System.String) System.Void))
    (ret))
```

Method body should be a CIL instruction sequence.

3.4.3 CIL instructions

Currently only a small subset of all available CIL instructions is supported by Naggum. This set will be extended in future.

1. Call instruction:

   ```
   (call (assembly type-name method-name (argument types) return-type))
   ```

   Currently assembly name is ignored; only `mscorlib` methods can be called. Static assembly function calls are not supported yet.

   Method argument and return types should be fully-qualified.

2. Load string instruction:

   ```
   (ldstr "Hello, world")
   ```

   Loads a string onto a CLI stack.

3. Return instruction:

   ```
   (ret)
   ```

   Return from current method.

3.4.4 Example assembly definition

```
(.assembly Hello
    (.method Main () System.Void (.entrypoint)
        (ldstr "Hello, world!")
        (call (mscorlib System.Console WriteLine (System.String) System.Void))
        (ret)))
```

3.5 High-level syntax

Every high-level Naggum program is a sequence of function definitions and a top-level executable statements. Functions defined in an assembly are also available as public static methods to be called by external assemblies.

Functions are defined using `defun` special form:

```
(defun function-name (arg1 arg2)
    statement1
    statement2)
```

For example:

```
(defun println (arg)
    (System.Console.WriteLine arg))
```

Naggum is a Lisp-2, henceforth a function and a variable can share their names.

Currently executable statements may be one of the following.

1. Let bindings:

```
(let ((variable-name expression)
      (variable-name-2 expression-2))
     body
     statements)
```

Creates a lexical scope, evaluates initial values, binds them to corresponding names and evaluates the body, returning the value of last expression.

Naggum's `let` is a loner: every one is inherently iterative (like `let*`) and recursive (like *let rec*).

1. Arithmetic statements:

```
(+ 2 2)
```

2. Function calls:

```
(defun func () (+ 2 2))

(func)
```

3. Static CLI method calls:

```
(System.Console.WriteLine "Math:")
```

4. Conditional statements:

```
(if condition
    true-statement
    false-statement)
```

If the `condition` is true (as in "not null, not zero, not false") it evaluates the `true-statement` form and returns its result. If the `condition` evaluates to false, null or zero, then the `false-statement` form is evaluated and its result is returned from `if`.

1. Reduced if statements:

```
(if condition
    true-statement)
```

2. Constructor calls:

```
(new Naggum.Runtime.Cons "OK" "FAILURE")
```

Calls an applicable constructor of a type named *Naggum.Runtime.Cons* with the given arguments and returns an object created.

CHAPTER 4

Naggum Specification

4.1 Features

- based on CLR;
- Lisp-2;
- compiles to CIL assemblies;
- is not a Common Lisp implementation;
- seamlessly interoperates with other CLR code.

4.2 Language

4.2.1 Special forms

1. `(let (bindings*) body*)` where `bindings` follow a pattern of `(name initial-value)` creates a lexical scope, evaluates initial values, binds them to corresponding names and evaluates the body, returning the value of last expression. Naggum's `let` is a loner: every one is inherently iterative (like `let*`) and recursive (like `let rec`).
2. `(defun name (parms*) body*)` defines a function (internally it will be a public static method). Naggum is a Lisp-2, henceforth a function and a variable can share their names.
3. `(if condition if-true [if-false])` evaluates given `condition`. If it is true (as in "not null, not zero, not false") it evaluates `if-true` form and returns it's result. If `condition` evaluates to false, null or zero then `if-false` form (if given) is evaluated and it's result (or null, if no `if-false` form is given) is returned from `if`.
4. `(fun-name args*)` applies function named `fun-name` to given arguments.
5. `(new type-name args*)` calls applicable constructor of type named `type-name` with given arguments and returns created object. `(new (type-name generic-args*) args*) new` form calls applicable

constructor of generic type named `type-name`, assuming generic parameters in `generic-args` and with given arguments and returns created object.

6. `(call method-name object-var args*)` Performs virtual call of method named `method-name` on object referenced by `object-var` with given arguments.
7. `(lambda (parms*) body*)` Constructs anonymous function with `parms` as parameters and `body` as body and returns it as a result.
8. `(eval form [environment])` evaluates form using supplied lexical environment. If no environment is given, uses current one.
9. `(error error-type args*)` throws an exception of `error-type`, constructed with `args`.
10. `(try form (catch-forms*))` where `catch-forms` follow a pattern of `(error-type handle-form)` tries to evaluate `form`. If any error is encountered, evaluates `handle-form` with the most appropriate `error-type`.
11. `(defmacro name (args*))` defines a macro that will be expanded at compile time.
12. `(require namespaces*)` states that `namespaces` should be used to search for symbols.
13. `(cond (cond-clauses*))` where `cond-clauses` follow a pattern of `(condition form)` sequentially evaluates conditions, until one of them is evaluated to `true`, non-null or non-zero value, then the corresponding `form` is evaluated and it's result returned.
14. `(set var value)` sets the value of `var` to `value`. `var` can be a local variable, function parameter or a field of some object.

4.2.2 Quoting

1. `(quote form)` indicates simple quoting. `form` is returned as-is.
2. `(quasi-quote form)` returns `form` with `unquote` and `splice-unquote` expressions inside evaluated and substituted with their results accordingly
3. `(unquote form)` if encountered in `quasi-quote` form, will be substituted by a result of `form` evaluation
4. `(splice-unquote form)` same as `unquote`, but if `form` evaluation result is a list, then it's elements will be spliced as an elements of the containing `list`.

4.2.3 Type declaration forms

- `(deftype type-name ([parent-types*]) members*)` Defines CLR type, inheriting from `parent-types` with defined members.
- `(deftype (type-name generic-parms*) ([parent-types*]) members*)` Defines generic CLR type, polymorphic by `generic-parms`, inheriting from `parent-types` with defined members.
- `(definterface type-name ([parent-types*]) members*)` Defines CLR interface type, inheriting from `parent-types` with defined members.
- `(definterface (type-name generic-parms*) ([parent-types*]) members*)` Defines generic CLR interface type, polymorphic by `generic-parms`, inheriting from `parent-types` with defined members.

If no `parent-types` is supplied, `System.Object` is assumed.

4.2.4 Member declaration forms

- `(field [access-type] field-name)` declares a field with name given by `field-name` and access permissions defined by `access-type`.
- `(method [access-type] method-name (parms*) body*)` declares an instance method. Otherwise identical to `defun`.

Available values for `access-type` are `public`(available to everybody), `internal`(available to types that inherit from this type) and `private`(available only to methods in this type). If no `access-type` is given, `private` is assumed.

4.3 Standard library

Naggum is designed to use CLR standard libraries, but some types and routines are provided to facilitate lisp-style programming.

4.3.1 Cons

Cons-cell is the most basic building block of complex data structures. It contains exactly two objects of any types, referenced as *CAR* (left part, head) and *CDR* (right part, tail)

4.3.2 Symbol

Symbol is a type that represents language primitives like variable, function and type names.

4.3.3 Naggum Reader

Reader reads Lisp objects from any input stream, returning them as lists and atoms.

4.3.4 Naggum Writer

Writer writes Lisp objects to any output stream, performing output formatting if needed.

Naggum (named in honor of Erik Naggum) is a modern statically typed Lisp variant that is targeting Common Language Infrastructure (CLI) runtime system.