The KerLang Programming Language Manual

an introduction to comment-oriented programming using the *Glorious KerLang Compiler*

Arthur Correnson, Anima Libera, Igor Martayan

Introduction

Commenting the code is good practice, but no one ever said that coding the comments is good practice. In our programming language *KerLang*, the compiler/interpreter does it for the programmer, thus reducing the boiler plate code that merely parahases the comments.

What is comment-oriented programming ?

COP (Comment-Oriented Programming) is a programming paradigm that relies on the fact that programmers shouln't be trusted at the task of writing code matching the comments. Instead of relying of first-class code, the language relies on first-class comments that describes what to do to some extent.

How to use the Glorious KerLang Compiler

COP is all about a tight relationship between the programmer and its compiler. A COP programmer typically start a new project by writing tons of comment as a blueprint of a piece of software. Then, the compiler can be asked to generate code based on the provided comments.

If the comments are not precise enough, the compiler may get stuck and report the impossibility to synthesize a piece of code.

Syntax of the language

A KerLang program consist of a list of function declarations all preceded by a documentation comment :

```
/*
  ... Comment1 ...
*/
```

```
function f1;

/*
  ... Comment2 ...
*/
function f2;

...
```

Such a program shall be written in a `.kl` file and then feed into `gklc`.

Commenting your code

Comments follows a flexible syntax allowing you to precisely describe the expected behavior of your code. The documentation of each function consists of a list of sentences each separated by a dot:

```
/*
  sentence 1.
  sentence 2.
  sentence 3.
*/
function my_fun;
```

Sentences can be of different kinds: 1. **Argument specification** : + The signature of a function need to be provided. This can be done by writing the comment `This function takes 3 arguments.` for example + If the number of argument is'nt specified, it may lead to a compile error 2. **Result description** : + Functions computes things so we need a way to describe what is computed by each function. + The return value of a function can be described using the `Returns` keyword. For example `It returns 1` + Complex expressions may be returned. For example `It returns the sum of argument 1 and argument 2` 3. **Let bindings** : + it may be useful to break complex expressions into smaller pieces + this is achieved via the `Let` keyword + For example the comment `Let x be the sum of 1 and 2.` binds the name `x` to `(1 + 2)` + A local binding can be referenced in comments (as in `It returns the sum of x and 1`) 4. **Hint on the result** : + Hints can be provided to precise what operations should be used to compute the result of a function. This can be done using the `Uses` keyword + For example, `Uses x.` specify that `x` should be used in the result + If the compiler can't generate a function because of under-specification, it tries to uses hints to generate a pseudo-random function 5. **Printing** : + Printing integers can be done by asking the compiler to `show` expressions, such as in `This function shows the sum of 2 and 3.`

Note : Many equivalent formulations are understood by gklc (provided they are written in english !). See the `examples/` folder for more details.

Note : Values can be a `let` binding name, a constant, an argument, or even just

`something` (provided the compiler has things to `use` in its place). Operations can be basic arithmetic such as sum, multiplication, etc., a function call, a condition.