libGOCR API

GNU
Optical
Character
Recognition

Bruno Barberi Gnecco *brunobg@sourceforge.net*

July 22, 2001

Contents

Chapter 1

Introduction

GOCR is an attempt to fulfill a large gap in the Linux world: the lack of an OCR program. At the time the project started, there were some available, but their quality was very deceptive. Licensed using the LGPL license, it can be used by anyone.

As of the 0.3.x versions, it was decided that gocr, until then a program, should become a library. I (Bruno) decided then to be responsible for it, and this is the result. I hope I made a good work, or at least something that's quite usable.

This documentation covers three different views on the API, which are the layers it's subdivided. First, there's the GOCR frontend API itself, which allows you to write a program that uses the library to do some OCRing. It's a small set of functions that allow you to decide what operations should be done, and in what order, and to tune some of the attributes of the library. Second, there's the module interface. GOCR library let you write new pieces of code or to complement the existing ones without recompiling; we call these pieces *modules*, but many other programs call them plugins. It's just nomenclature. This API is fully independent of the first one, and has a completely different functionality. Last, but not least, is the internal GOCR API. You don't need to know what it is, or even that it exists, but it's what joins the two first API's, all the modules you're using, the program you wrote, and makes it all work together, or not. It's GOCR itself, and you only want to know about it if you want to develop GOCR.

With the API, there is the possibility of writing wrappers, or bindings, to other languages. C++ and Python are on the list, and soon will be available.

This document was written not only as a reference, but as a tutorial too; the language is light, a handful of jokes are spread around, etc. The code is well documented, and automatic documentation, man pages, etc, can be generated using Doxygen.

1.1 Authors and contact information

GOCR project was created by Jörg Schulenburg
<`Joerg.Schulenburg@physik.uni-magdeburg.de`>.

It's currently hosted at Sourceforge: `http://jocr.sourceforge.net` (yes, with an 'j').

Other developers joined the effort, and several people send patches, bug reports and ideas.

The API was designed and this manual was written by Bruno Barberi Gnecco <`brunobg@geocities.com`>

1.2 Version information/development plan

This manual contains the 0.7.1 API standard. 0.7.x versions are development versions, which will be used until a stable, usable and complete version is reached. By that time, version number will be upgraded to 0.9. The 0.9.x versions will be for debugging and testing, because minor corrections are to be expected. Once it's good enough to be widely, publicly used, it will be 1.0.

So, in other words, while it's not 1.0, you can't blame it at all that it sucks and doesn't work. After that, it's OK. :-)

1.2.1 Current status

The frontend API is pratically stable, but new additions will come. A new image loading system was designed and implemented (0.7.1). In the near future a wrapper to a GUI system will be added, so modules can interact with the frontend-user.

The internal API is being done solidly, to avoid future problems. I'm taking special care to make sure that it's a good system, and will support the rest well. There's a real bunch of fprintf's to the inevitable debugging. ;-)

The module API is not stable. It's being developed. The general idea, however, is here.

Chapter 2

GOCR API

GOCR API is a simple set of functions that let you easily write a frontend. You are responsible for what modules you are calling. A module is simply a piece of code that performs a certain kind of function; it will be explained more detailedly below.

2.1 Initializing and finalizing

The header that contains the prototypes, etc is `gocr.h`.

It's mandatory that you call two functions when using GOCR. They are:

```
int gocr_init ( int argc, char **argv );
void gocr_finalize ( void );
```

The first function parses the arguments your program got, setups all the internal structures of GOCR, initializes all that it's needed to run. It must be called before any other GOCR function. It returns 0 if GOCR could be correctly initialized, -1 otherwise. This is a constant in the API: if a function returns -1, it failed. You should always test the return values. GOCR also outputs to stderr what was the problem.

At the end of your program, or when you don't intend to use GOCR anymore, you must call the second function.

Currently, GOCR accept the following arguments: none yet.

2.2 Attributes

After calling `gocr_init()`, the next thing to do is to set the attributes of the library. These are parameters that let you tune several aspects of the API. They can be set and read using these two functions:

```
int gocr_setAttribute ( gocr_AttributeType t, void *value );
void *gocr_getAttribute ( gocr_AttributeType t );
```

The first function sets the attribute `t` with a value `value`. The second returns the current value of attribute `t`. The list of attributes currently supported and the values you can pass to them is:

Attribute type	Value	Function	Default
LIBVERSION	string	Returns a string containing the library version. This is a read-only attribute.	none
VERBOSE	an integer from 0 to 3	Sets the level of output: 0 nothing; 1 error messages; 2 warnings and errors; 3: everything. Used mostly for debugging.	1
BLOCK_OVERLAP	boolean	If true, allows two blocks to overlap	FALSE
NO_BLOCK	boolean	If true, and no block was found, creates a block covering whole image.	TRUE
CHAR_OVERLAP	boolean	If true, allows characters to overlap	TRUE
CHAR_RECTANGLES	boolean	If true, all characters are selected as rectangles	TRUE
FIND_ALL	boolean	If true, first find all characters, saving in memory, and then process.	FALSE
ERROR_FILE	(FILE *) variable	Sets the error messages output file.	stderr
PRINT	an integer from 0 to 6	0: only data bit (. = white, * = black) 1: marked bits (mark1 + 2*mark2 + 4*mark3) 2: data and marked bits: if white, a...h; if black, marked bits->A...H 3: only isblock bit (. = is not block, * = is block) 4: only ischar bit (. = is not char, * = is char) 5: complete byte, in hexadecimal 6: complete byte, in ASCII	0
PRINT_IMAGE	boolean	If true, gocr_print* functions will print the image associated with the structure.	1

Boolean values are either GOCR_TRUE or GOCR_FALSE. *Do not use TRUE or FALSE, since they are defined with different values by Unicode.*

Some module packages may require certain attributes; take a look at their documentation. They may automatically set these attributes, so don't be stubborn and override. Certain functions of libgocr may lock some attributes, to avoid chaos.

2.3 Images

If the purpose of your program isn't opening an image and processing it to turn into some kind of text, you are reading the wrong document ;-). GOCR currently works this way: you open an image, let the modules process it, and close it. This can be done any number of times you want. Image loading and closing is done using:

```
int  gocr_imageLoad( const char *filename, void *data );
void gocr_imageClose ( void );
```

well, they are pretty clear. `gocr_imageLoad()` returns 0 in case of success, -1 otherwise. If you try to open an image while there's one already open, `gocr_imageLoad()` will return -1.

Image loading is part of a module, and `gocr_imageLoad()` may be overriden. Libgocr provides a default one, which is capable of opening the most common image types. It accepts, as the second argument, one of these:

GOCR_BW Convert to black and white.

GOCR_GRAY Convert to grayscale.

GOCR_COLOR Convert to RGB (24 bit) color.

GOCR_NONE Do not convert.

2.4 Modules

2.4.1 Introduction to modules

There are three things that could be called a *module* in GOCR, so here's a thorough specification:

- the module type. There are many different types of modules, as explained below. For example, there's a imageFilter type, that may be used to do clean the image dust, for example, and a charRecognizer, that is intended to get a small image of a single character and find out which one it is. When I refer to *module*, I usually mean an instance of a module type.

- the function . Each module type may have several different functions. For example, imageFilter module may have a function to increase contrast, another to clean dust, and a third to remove coffee mug stains. These are called *module functions*, or simply *functions*.

- the file, which ends with .so, and is a shared object. In our terminology, this is a *shared object file*, or (same thing different name) a *module package*. This file contains the module functions, which may be of different module types.

There are several module types:

Module type	Function	Examples
imageLoader	Loads an image.	Load images. There can be only one image loader.
imageFilter	Filter the image.	Dust removal, etc.
blockFinder	Find blocks, i.e., groups of similar data and add information of its content.	Find pictures, find columns of text, find mathematical expressions.
charFinder	Frame characters, and add information of its content.	Frame characters, font recognition.
charRecognizer	Recognize the framed characters.	Italic, bold, greek specialiazed OCR.
contextCorrection	Try to recognize the still unrecognized characters.	Spell checker, ligature checker.
outputFormatter	Output data to some format and file.	HTML output, LaTeX output.

All of the modules (except imageLoader) may be composed of several different functions, which may be in different module packages. The following sections explain how to load modules, set their order, and run them.

2.4.2 Loading shared object files

The first thing to do, when you want to add some function to a module, is to open its file. All the work is done internally by the library, and you just need to call:

```
int gocr_moduleLoad ( char *filename );
```

If `filename` is just the filename, libgocr will search for the file in the following directories:

- A colon-separated list of directories in the user's LD_LIBRARY path environment variable.
- The list of libraries specified in /etc/ld.so.cache.
- /usr/lib, followed by /lib.
- The directory libgocr was installed in.

This function returns a module id (that can be used to set attributes, see below) if the operation was successful, -1 otherwise.

2.4.3 Setting module attributes

Some module packages allow you to set their attributes. You can do this using this function:

```
int gocr_moduleSetAttribute ( int id, void *a, void *b );
```

id is the module package id

a, b fields are passed directly to the module package, refer to its documentation to know how to use them.

The function returns -1 in case of some internal error, or the value returned by the module package.

2.4.4 Loading module functions

Since a shared object file may have several different module functions, and you may be interested only in one of them, GOCR enables you to decide exactly which module function should be run, and the order they do that. The functions that load module functions are:

```
int gocr_functionAppend ( gocr_moduleType t,
    char *functionname, void *data );
int gocr_functionInsertBefore ( gocr_moduleType t,
    char *functionname, void *data, int id );
int gocr_functionDeleteById ( int id );
```

Module functions are internally saved in a linked list, but you don't have to know that (so, I shouldn't haven written... well, knowledge is never too much). Let's first see `gocr_functionAppend`. The arguments are:

t the module type, as in the first column of the table above.

functionname this is the name of the module function you want to load. Refer to the documentation that should come with the shared file object.

data this is a parameter that will be passed to the function when it's called. It's a pointer, that you are responsible for allocation. Do not free it until you call `gocr_functionDelete` or `gocr_finalize`. Being a void pointer, you can pass anything to it. If you need more than one argument, use a structure. Read the module function docs to know what you can do with this.

`gocr_functionAppend` returns -1 in case of error, or a non-negative number if successful. This number is the function's ID. It can be used if you want to do access this function.

`gocr_functionDeleteById` is straight forward. Its sole argument is the id of the module function you want to delete. As usual, returns -1 if error, 0 on success.

Last, but not least, there's `gocr_functionInsertBefore`. It works like its counterpart `gocr_functionAppend`, but there's a difference: it allows you to insert a function in the middle of the list. Good for the absent minded ones. The first three arguments are the same of `gocr_functionAppend`, and the fourth argument is the id of the function that is be just after the position you want to insert the new function. So, if you want to insert a function in the first position, you should pass the id of the current first position function. Hm. Read it again, and it should become clearer. ;-)

The order of the inclusion is very important, since it will determine the order of running. So, if you add a module function to recognize cyrilic text before latin text and try to decode a latin text, it'll be much slower than if you did vice-versa. Always sort the functions by the probability of their usefullness.

Note that you don't need to specify in which shared object file the function is; GOCR does it automatically for you.

2.4.5 Running modules

Now that you did everything, there remains only to run the modules. GOCR allows you to run them all at once, module by module, or module function by module function:

```
int gocr_runModuleFunction ( int id );
int gocr_runModuleType ( gocr_moduleType t );
int gocr_runAllModules ( void );
```

The functions are simple to use. `gocr_runAllModules` runs all the modules, taking care of how it's done. For example, charFinder module functions must be called one for each block. It's not a trivial for(), and this is the recommended way to do it. It follows the order that you provided when you appended and inserted the module functions, as described in the last section.

The two other functions are currently not working, due to design issues.

`gocr_runModuleType` runs a specific module. There's no care taken of the internal data, which must be manually updated. It may be useful if you want just to apply some filters to the image, for example, or if you want to do a different implementation of the existing `gocr_runAllModules`.

Last there's `gocr_runModuleFunction`. It runs just one module function, and also doesn't take care of internal data. If you want to use it, you probably know what you are doing.

All functions return 0 on success, -1 on error.

2.4.6 Closing modules

It's possible to close a module. `gocr_finalize` automatically takes care of closing all modules, but if you have some special reason to close a module, you can do it. Libgocr automatically deletes all the module functions of this module. Just call:

```
void gocr_moduleClose ( int id );
```

And that's it.

2.5 A simple example

Ok, time to do something concrete.

Usually examples are neat little programs, heavily commented, that do something completely useless. Since this is a tradition, I was unable to refrain using it. Unfortunadly GOCR can't do "Hello World", and so I had to imagine something equally uninteresting, and I used the filter example I just told you.

```
/* filter.c
 * A simple program, that applies a filter to a
 * image, and outputs the image.
 */

#include <gocr.h>
int main(int argc, char **argv) {
     /* Initialize the library */
     if ( gocr_init(argc, argv) == -1 )
         exit(1);
     /* Set output to zero */
     if ( gocr_setAttribute(VERBOSE, 0) == -1 )
         exit(1);
     /* Load a shared object file */
     if ( gocr_moduleLoad("modulename.so") == -1 )
         exit(1);
     /* Load a module function that cleans dust */
     if ( gocr_functionAppend(imageFilter, "cleanDust", NULL) != -1 )
         exit(1);
     /* Load a module function that outputs an image */
     if ( gocr_functionAppend(outputFormatter, "imageOutput",
                    "output.jpg") != -1 )
         exit(1);
     /* Load the image */
     if ( gocr_imageLoad("image.jpg", (void *)GOCR_NONE) )
         exit(1);
     /* Run all modules. */
     gocr_runAllModules();
     /* Ok, say good bye */
     gocr_finalize();
}
```

The usual comments, now. Notice that two module functions were loaded. The first cleans 'dust' of the image, i.e., those nasty pixels that are black in what should be a perfectly white background. The second module outputs the image after the cleaning. Notice how this hypothetical module function takes as argument the name of the output file.

When you call `gocr_finalize()`, it takes care of unloading shared objects, deleting module functions, closing the image, etc. Don't worry with hundreds or close()s, free()s, etc.

2.6 Serious tweaking

Although libgocr has several module types, you don't have to use them all, and is free to abuse of the architecture. In fact, only doing so you'll be able to take full advantage of libgocr's power.

Let's say, for example, that you are writing an algorithm that skips the segmentation process, finding characters directly. At first, it seems that such algorithm would be completely incompatible with libgocr's structure; but it's not. Here are some possible solutions:

- use the algorithm as a blockFinder module, and do not use any charFinder or charRecognizer modules. This way you work with the entire image.
- use the algorithm as a charFinder module. It allows the separation of the image in blocks, and you can treat it block as a whole image. It's also 100% compatible with other charFinder modules.

You may think that this is an ugly hack, but it's not. I'll explain why: since the architecture of libgocr is modular, and the modules can be used independently (with certain exceptions), it's not only OK to do it, it's designed to be used this way. The module types had to be given names, but it's as wrong to think that a charFinder module should only frame characters as to think that charRecognizer can only recognize usual characters, and not musical notes.

Something else: do not get stuck with `gocr_runAllModules()`. Since you may change interpretation of module types, it may be interesting to run them in a different way, skip some, run some twice, allow feedback, etc.

The question that arises now is: why not make the modules objetcs, similarly to what is done with block types (see 3.4.1)?

- To do so, the module type objects (MTO) would need to have their own `run()` methods. Since some modules use information of their predecessors (charFinder uses blockFinder, charRecognizer uses charFinder), MTOs would have to be attached to each other, making a mess.
- There could be an unnecessary multiplication of MTOs. It would be very easy to decide that "I don't like that MTO, because the method names are too big", and write a new MTO with the same functionality.
- Compatibility. Current module types are 100% compatible with each other, sharing common structures and variables. Since they are part of libgocr, you are assured that your module will be compatible with any other module, something that would not happen with MTOs.
- Current architecture was carefully designed to work well and in a broad range of situations, and abusing of it is legal.

If you need to create a new module type, it's likely to be a very specific situation, where you do not care about compatibility.

2.7 GUI wrapper - message system

todo

Chapter 3

Modules API

This chapter is intended to those that want to write a module. Please take a look at section 2.4 first.

It's necessary to include the file `gocr_module.h`, which defines all the necessary stuff. Unless you need some function declared there, there's no need to include `gocr.h`.

3.1 Modules in brief

There are somethings to say about modules that apply to all types.

Upon loading a shared object file, GOCR tries to call a function with the following prototype:

```
int initModule ( void );
```

so, if you need to initialize some data, just declare this function. If the function returns something different than 0, it's assumed that some error occured, and the module package is imediately closed.

Similarly, when a module package is closed, GOCR tries to call

```
void closeModule ( void );
```

which you can use to free memory, etc.[1]

Besides these two functions, there's a third function, also optional, that may be used to set attributes in real time:

```
int setAttribute ( void *, void * );
```

This function can (and will only) be directly accessed by the user. Note that you can cast directly a `void *` to a `int` or to a `char *`, so the user can access parameters #including some header with #defines, etc.

[1] You may be wondering about the _init and _fini symbols, used by libdl. GOCR doesn't use libdl directly, since libdl is not portable. To avoid conflicts and undefined behavior, do not define _init or _fini.

3.1.1 Module Development Kit

To load shared object files, GOCR uses libltdl, which is included in libtool. It's a bit less straight forward than working with libdl directly, but in return it's much more portable.

If you never worked with libraries, libdl, or just don't have a clue of what I'm talking about, and "just want to write this module to recognize handwriting, man, that's all", don't worry. The developers of GOCR have spent countless hours to make your life easier[2]. You don't even have to know anything of the confusing world of libraries, shared, static, cryptic gcc arguments, weird makefiles and confusing configures.

All you have to do is write your code, and get the module development kit (MDK) from http://jocr.sourceforge.net/download.html. This package is a whole bunch of files that take care of the libtool, automake, autoconf, and every other little pesty thing that would add more hours of work to you, trying to figure out what the hell did you forget in Makefile.am. Or configure.in. See, that's what I'm talking about.

The MDK comes with it's own documentation, which you should read before you start coding. All you have to do, however, is to edit the `module-setup` script, fill some of its fields properly, and run it. It will create all necessary files, and all you have to do is run `./configure` to create the Makefiles.

That's it. If you think it's too much work, do all the rest yourself ;). As I said, I'm gonna write a nice script to automatize it soon. Note that you need the automake/autoconf packages installed in your computer. They are available at your closest GNU repository. Anyway, as I said, MDK is properly documented, so read it.

3.1.2 Packaging and releasing

Here are some guidelines to help you release your module:

- **Write documentation**. This is a complete must, because if you don't write it, people won't know what module functions are available in the package, and won't be able to use your module, and then I think you'd missing the point. Be sure to explain what each module function does, and what arguments it may receive.

- It's a good idea to add a prefix to the module functions of a module package. For example: foo_clean(), foo_recognize().

- Do not duplicate code. If someone already did what you want, don't replicate it in your code. By the other hand, don't ask to the user to have several libraries of module packages; if you need only a function, have it in your own code (respect the software license).

- There's already a easy way to package: type `make dist`. It will generate the appropriate tar file.

- Read the Software Release Practice HOWTO.

[2] That is, I spent some time I had nothing to do developing methods to let you spend some time you have nothing to do developing.

- Take a look at existing modules. If you are having some problem, chances are that by peeking at other's work you can find a solution. This is one of the most important laws of software coding. Be nice and add a thanks note to your documentation.

3.2 imageLoader

This module is a special one; every good rule must have a exception. The differences between imageLoader and the other modules are:

- There may be only one function in the image loader module at a time, which makes sense, since there may be only one open image at a time.
- The imageLoader function may be accessed directly by calling `gocr_imageLoad()`.
- This module is not called by the `gocr_run*Modules()` functions.

libGOCR has a default image loader module, which currently opens the following images types[3]:

- .pnm
- .pbm
- .pgm
- .ppm
- .jpg/.jpeg
- .gif
- .bmp
- .tiff
- .png

3.2.1 Image and pixels

When implementing libGOCR, the question arised: should we use grayscale? Is black and white enough? What about colors? We decided to use black and white only, since it seemed more than enough, and saved memory. Later, it was realized that color would be essential to some recognition systems — specially if you want to use libGOCR to recognize something other than plain text. The design was changed, and now libGOCR support these image types[4]:

Type	Symbol	Pixel size
Black & white	GOCR_BW	1
Grayscale	GOCR_GRAY	2
Color	GOCR_COLOR	4
User-defined	GOCR_OTHER	-

[3] Subject to availability of certain libraries. See the README file.

[4] Pixel size is in bytes, and is valid only for the x86 architecture (although if you have a decent compiler and sizeof(char)==1 then the results are likely to be the same o others).

You may only access the image indirectly.

The whole point of using an image is that you can access pixels individually, so, after several conferences and hundreds of emails, we decided that yes, we would have pixels in our images. Ok, the joke was not funny.

To support the different image types, a slight hack was done in the gocrImageData structure, which contains the individual pixel data (section 3.2.3 has info about it, but you definitely don't need to know). In fact, you only won: you can access any image type just as if it's the type you want; that is, suppose the image loaded is in color, but you want to work in black and white: you can. The functions are:

```
void gocr_imagePixelSetBW ( gocrImage *image
    int x, int y, unsigned char data );
unsigned char gocr_imagePixelGetBW ( gocrImage *image,
    int x, int y );
void gocr_imagePixelSetGray ( gocrImage *image,
    int x, int y, unsigned char data );
unsigned char gocr_imagePixelGetGray ( gocrImage *image,
    int x, int y );
void gocr_imagePixelSetColor ( gocrImage *image,
    int x, int y, unsigned char data[3] );
unsigned char *gocr_imagePixelGetColor ( gocrImage *image,
    int x, int y );
```

Examples:

```
if ( gocr_imagePixelGetBW(img,0,0) == GOCR_WHITE )
    gocr_pixelPixelSetBW(img,0,0,GOCR_BLACK);

for ( i = 0; i < img->width; i++ )
    for ( j = 0; j < img->height; j++ )

        if ( gocr_imagePixelGetGray(img,i,j) > threshold )


           gocr_imagePixelSetBW(img,i,j, GOCR_WHITE);
        else

           gocr_imagePixelSetBW(img,i,j, GOCR_BLACK);
```

The only thing to note is that, if you provide (x,y) coordinates out of bounds, the functions will return 0, which is also a valid value for a pixel.

Each pixel has three fields that may be used as flags. They are boolean variables, and to access them use:

```
int gocr_pixelGetMark1 ( gocrImage *image, int x, int y );
int gocr_pixelSetMark1 ( gocrImage *image, int x, int y,
    char value );
int gocr_pixelGetMark2 ( gocrImage *image, int x, int y );
int gocr_pixelSetMark2 ( gocrImage *image, int x, int y,
    char value );
int gocr_pixelGetMark3 ( gocrImage *image, int x, int y );
int gocr_pixelSetMark3 ( gocrImage *image, int x, int y,
    char value );
```

They are pretty clear, and return -1 in case of error.

3.2.2 The module

The `imageLoader` module has the following prototype:

```
int gocr_imageLoaderFunction ( const char *filename,
    void *data );
```

which, of course, may be named whatever you want. It's directly accessible by the user (by calling `gocr_imageLoad`), and you can use the `data` field to pass arguments.

GOCRlib provides a default image loader, which handles the most common formats, and can convert images to any of the GOCRlib supported types (GOCR_BW, GOCR_GRAY, GOCR_COLOR) by using one of these symbols as argument. You should use GOCR_BW whenever you don't need extra information, since it's likely to take much less memory than the others.

It can be accessed with gocr_moduleAppend/etc by using "default" as argument. etc

3.2.3 Creating your own image type

This is not currently supported. It may be taken out, since C is unlikely to let us do it easily.

If you need to create a special type, here's how to do it. It's not recommended that you do it, for the following reasons:

- it's likely to be incompatible with current modules.
- blabla

What you need to do is quite simple. Declare your pixel like this:

```
struct mypixel {
    unsigned char pad : 1; /* pad pixel */
    unsigned char mark1 : 1; /* user defined marker 1 */
    unsigned char mark2 : 1; /* user defined marker 1 */
    unsigned char mark3 : 1; /* user defined marker 1 */
```

```
        unsigned char isblock : 1; /* is part of a block? */
        unsigned char ischar : 1; /* is part of a character? */
        unsigned char private1: 1; /* internal field. */
        unsigned char private2: 1; /* internal field. */
        /* your data goes here */
    };
    typedef struct mypixel MyPixel;
```

You should name your data field `value`.

More: struct size, etc.

3.3 imageFilter

It my be interesting to apply some filters to the image, to remove dust, etc. The functions of this module will get the image and apply the filter to it.

Prototype is

```
int gocr_imageFilterFunction ( gocrImage *image, void *v );
```

You can work freely with the image, and apply any filters you desire; remember that modules that were not written by you may be used too, so do not apply a filter that changes the image data (gradient, laplacian, Fourier transform, etc). As a special note, do not create (complete) copies of the data, since it's likely to be big (expect a few megabytes for the image size).

todo: document application of filters to blocks of data, which may be transformations, etc.

3.4 blockFinder

The objective of this module type is to divide the image in a number of blocks. A *block* is a set of pixels that are part of the original image, whose contents are all of the same type. Examples: a picture, a text column, a mathematical expression, a title.

You must take care to avoid recognizing what should be only one block into more than one. Sometimes that's perfectly fine: for example, if a picture is recognized as two blocks, as long as they don't intersect each other, the only price to pay is to have two image files saved instead of only one; or if a text column is divided in half, along the horizontal, the output is likely to not take notice. But if the column is divided along the vertical, you may have a bad output. It's easier to say than to do, but a warning never hurts.

The prototype of a blockFinder function is:

```
void gocr_blockFinder ( gocrImage *img, void *v );
```

3.4.1 Block types

Besides finding each block, you should try to recognize what kind of information that block carries. This will make the work of subsequent modules much easier, and will improve the speed of the processing.

GOCR automatically defines three types of blocks:

Block type
TEXT
PICTURE
MATH_EXPRESSION

but you can define new types, as explained below. The default is TEXT.

The block types are objects, which all derive from a common parent, `gocrBlock`. This allows any module to access the block, regardless of its type. This is what allows you to create new block types on the fly. To do that, you must first define the `struct` of your new block type, which must be in the following format:

```
struct newblocktype {
    gocrBlock b;
    /* other fields */
};
```

It's absolutely necessary that the first field of your structure be `gocrBlock b`. This is what allows to cast your structure to a simple `gocrBlock` (If you are wondering why the hell I didn't use C++ instead of C, these are the reasons: it's easier to use C from C++ than the opposite; I have much more experience with C than C++; there are several people that program in C but not in C++; the use of C as an OO language, although slightly obfuscated, has proven to be possible and used in successful projects, such as GTK; C++ name mangling makes it more difficult to write modules, and is not supported yet by libtool).

You must register your block type, to make GOCR aware of its existance. To do that, use the following function:

```
blockType gocr_blockTypeRegister ( char *name );
```

This function takes the `name` of your new block type, registers it, and returns a non negative number, which is the block type id, or -1 if some error occurred. This id should be saved, to provide a quick way to check what is the block type. Alternatively, you can use:

```
blockType gocr_blockTypeGetByName ( char *name );
```

which returns the id of a already registered block type, or -1 if none was found. Since this function is kind of slow, as it must compare the string given to every other block type name registered, it's a good idea to save the id in a variable. Last, a convenience:

```
const char *gocr_blockTypeGetNameByType ( gocrblockType t );
```

given the block type, returns its name. Do not free this string.

3.4.2 Finding blocks

Once you find a block, you have to notify GOCR:

```
int gocr_blockAdd ( gocrBlock *b );
```

You are responsible for filling the `x0`, `x1`, `y0`, `y1` and `t` fields of the block structure, and *only* those (well, if you fill anything else nothing will happen, you'll just be wasting processor time). You can pass the address of a derived block type to it. The function returns 0 if OK, -1 if error (if the block type isn't registered, it's considered an error). If two blocks overlap, and the BLOCK_OVERLAP flag is set to 0, the function returns -2.[5]

3.4.3 Blocks are more than frames

The blockFinder module is really half of the core of GOCR. It's responsible to setup everything to make the recognition itself a simple (ahn, simpler) task. It should, therefore, do all that it can in order to make the next two modules perform a simple, linear operation.

Here's a description of what the module function should do for the three basic block types:

Text block

This structure will probably be severely changed.

The text block structure is:

```
struct gocrtextblock {

    gocrBlock b;   /* parent; must be first field */
    List  linelist;

};
typedef struct gocrtextblock gocrTextBlock;
```

The `gocrBlock b`, as described above, is used to perform OO, and must be the first field. The only other field is a linked list (see section4.2) of text lines:

```
struct line {
    int  x0, x1; /* x-boundaries */
    int  m0, m1, m2, m3; /* y-boundaries */
    List  boxlist;
}; typedef struct line gocrLine;
```

the `x0` and `x1` fields are the vertical boundaries, and the `m?` fields are y boundaries:

[5] In the future, it will be possible to have blocks of any format, using a system similar to the used in characters currently. The problems are two: outputFormatter, and how to save the data without memory waste.

Field	Description
m0	Top boundary
m1	Middle
m2	Baseline
m3	Bottom

PICTURE describing them

These fields are of utmost importance to the charRecognizer and charFinder modules, and their correct determination is crucial. Last is `boxlist`, which is a list of `Boxes`, a structure described in the next section.

Picture block

This is a very simple structure:

```
struct gocrpictureblock {
    gocrBlock b; /* parent; must be first field */
    char *name;
};
typedef struct gocrpictureblock gocrPictureBlock;
```

The structure contains only one field, `name`, which is the name of the file to which the picture will be saved.

Math block

Will use trees. To do.

3.4.4 Final considerations

If no block was found, NO_BLOCK is set to 1 and `gocr_runAllModules()` was called, GOCR creates a block covering the entire image, and continues to process the image, calling the charFinder module. If NO_BLOCK is set to 0, then `gocr_runAllModules` returns -1.

3.5 charFinder

This module should parse each block and frame every character found. It should also provide information about the character, such as if it's bold or italic, the font, etc. This information is used by the charRecognizer module functions to quickly check if they will be able to recognize the character or will just waste processing time. Prototype:

```
int gocr_charFinder ( gocrBlock *b, void *v );
```

In more detail, what should happen in this module is in this pseudo code:

```
sweep the block
for each character {
```

```
        find pertinent pixels
        find pertinent attributes
    }
    return 0
```

The function should return 0 if it took care of the block, -1 otherwise (for example, you don't recognize the block type).

The way you sweep the block is completely on yourself, and but it must be done in a way that the outputFormatter module will understand. It makes sense. at least when parsing text, to sweep as one would read it (which means that you are not stuck to left to right, top to bottom languages). GOCR saves the characters in the order you add them. Talk about how charRecognizer will receive the data and add to a linked list, etc. Add some way to override this default behaviour of adding characters to the list

3.5.1 Getting block information

The charFinder module functions are specialized in certain block types, and thus get extra information from the blockFinder module. They must be so, otherwise they won't be able to read properly the block structure, which must be cast to the appropriate type. Your module function is likely to be something like this:

```
gocrBlockType your_block_type;
int charFinderFunction ( gocrBlock *b, void *v ) {
    switch ( b->type ) {
        case TEXT:

            gocrTextBlock *tb = (gocrTextBlock *)b;
            /* your code */
            return 0;
        case YOUR_BLOCK_TYPE:

            your_block_struct *mb = (your_block_struct *)b;
            /* your code */
            return 0;
        case PICTURE:
        default:
            return -1;
    }
}
```

This hypothetical function can deal with text blocks and a special block type that was previously registered, but not pictures or anything else; if you can't process a block, return -1; if you could, return 0. Currently, once a function process a block, GOCR supposes that it could do all the job there was to be done, and no other function is called (this is to avoid processing the same block twice and ending with duplicated information). Future versions may allow partial processing.

3.5.2 Delimiting characters

To delimit a character, GOCR API provides a set of functions that let you select only the pixels that are part of the character.

First thing to do is to declare that you are starting a new character:

```
int gocr_charBegin ( void );
```

This function returns -1 in case something is wrong; starting a character without ending the last one is considered an error. To end a character:

```
int gocr_charEnd ( void );
```

This function creates an image that is initially filled with the background color, with all bits unset. This image is big enough to contain all the pixels selected; these pixels are copied to the new image (only the data, the info bits are still unset), and will be passed to the charRecognizer module. gocr_charEnd automatically calls the charRecognizer module? Explain FIND_ALL

Between these two functions, you can set the pixels of the character, using the functions explained below. The `action` field is common to all of them; if GOCR_SET, then the function will select; if GOCR_UNSET, the function will unselect.

```
int gocr_charSetPixel ( int action, int x, int y );
```

Selects the pixel at (x, y).

```
int gocr_charSetAllNearPixels ( int action, int x, int y,
    int connect );
```

If `connect` is 4, selects all the pixels of the same color that are 4-connected with the pixel at (x, y); if connect is 8, selects all the pixels of the same color that are 8-connected with the pixel at (x, y). If `connect` is neither 4 nor 8, the function assumes 4-connection.

```
int gocr_charSetRect ( int action, int x0, int y0, int x1,
    int y1 );
```

Selects all pixels contained at the rectangle defined by (x0, y0) and (x1, y1). These points don't need to be top left and right bottom; they can be any diagonally opposite vertices. Internally, however, GOCR always convert (x0, y0) to be top left and (x1, y1) to be bottom right. This is valid for any function that takes two points defining a rectangle as arguments.

If you change your mind after a call to `gocr_charBegin`, you can still save the nation:

```
void gocr_charAbort ( void );
```

This function aborts a character begun using gocr_charBegin. All changes done by the gocr_charSet* functions since the last call to gocr_charBegin are undone.

When you can `gocr_charEnd`, the character can be saved as a simple rectangle that covers all the pixels you selected, or saving each individual pixel. While the later gives a lot more freedom, letting you select awkward regions, it consumes about 12.5% more memory, and is slower. This is controlled by the CHAR_RECTANGLES flag. Done as argument to gocr_charEnd?

3.5.3 Setting attributes

Setting attributes of the text can get quite complicated if you want to be fancy. It was decided to design a very simple, yet powerful system, that should be able to handle most of the stuff you ever need. First, a reminding note: these attributes should only be those that are applied directly to the text, such as bold, italic, font type, etc.

As usual in GOCR, the first thing to do is to create the attribute:

```
int gocr_charAttributeRegister ( char *name,
    gocrCharAttributeType t, char *format );
```

name attribute name; must be unique. We recommend to use capital letters, but it's up to you.

type there are two possible values:

- **SETTABLE** the attribute works like a flag: either it's set, or not set. Example: boldness.
- **UNTIL_OVERRIDEN** the attribute is valid for ever; you can only change it's values. Example: font. There must always be a font type and size, but they may change during the text.

format this field is used to store any attributes of the attribute (wow). It will be explained below, with a example.

As usual, the function returns 0 if OK, -1 if error (inserting an existant attribute is considered an error). Now that you created your attributes, you are processing the text and find that you need to set an attribute. Do it with the following function:This function name may be changed.

```
int gocr_charAttributeInsert ( char *name, ... );
```

name attribute name.

I bet you are probably wondering how the hell this stuff works. Me too. Uh, I mean, it's easier to understand using an example. The first one is simple:

```
gocr_charAttributeRegister("BOLD", SETTABLE, NULL);
...
gocr_charAttributeInsert("BOLD");
/* insert some text */
gocr_charAttributeInsert("BOLD");
```

Quite easy: first you register the bold style. It's a settable attribute, and since you don't need any extra information, the `format` field is NULL. Then, when processing the text, you find a word in bold. What you do is simple: insert a bold, insert the text, insert another bold. Since it's a settable attribute, the second one cancels the effect.

Let's do something fancier now:

```
gocr_charAttributeRegister("FONT", UNTIL_OVERRIDEN, "%s %d");
...
gocr_charAttributeInsert("FONT", "Arial", 18);
/* insert some text */
gocr_charAttributeInsert("FONT", "TimesNewRoman", 12);
/* insert some more text */
```

Now the explanation of the `format` field: it's just a printf-like format field! So, you can save whatever you want in a format that will be easily read by anybody, even if they do not know what it means — this is specially good when you are writing a outputFormatter module. When you insert the attribute, you pass the arguments to the format string. So, what happens in the example: we create an attribute "FONT", which is valid for ever. Note that, although it's valid for ever, it only starts to have effect when you first call `gocr_charAttributeInsert`, because you need to set its internal attributes (even if it doesn't have any). In the example, you are parsing a page, and finds that the title is typeset in Arial, size 18. The text in in Times New Roman, size 12.

Always remember that this system is subject to all the limitations of printf and scanf. For example: in scanf, %s reads a string up to the first white space, so you can't use spaces in a %s string, even though printf accepts it. And, since GOCR does not check the format string, if you screw it up you are screwing everything.

3.6 charRecognizer

This is the core of the OCRing. This module, using some ingenious algorithm, must be able to find that the bitmap it processed is a certain character. Prototype:

```
void gocr_charRecognizer ( gocrImage *pix, gocrBox *b, void *v );
```

`pix` is an image of the framed character, whose structure `gocrImage` is described in section **??**. There are two reasons to prefer to access `pix` than the `get/setData`: first, the former is much smaller, and will be entirely in the processor's cache, therefore being accessed much more quickly; second, the former starts on 0, while you'll have to add b->x0 and b->y0 to the latter. Of course, you may still use the `set/getData` functions.

3.6.1 Using UNICODE©

Quoting from a document by Markus Kuhn <Markus.Kuhn@cl.cam.ac.uk> that can be found at: http://www.cl.cam.ac.uk/˜mgk25/unicode.html. It's a very good document, and you should read it.

> What is UNICODE?
>
> Historically, there have been two independent attempts to create a single unified character set. One was the ISO 10646 project of the International Organization for Standardization (ISO), the other was the Unicode Project organized by a consortium of (initially mostly US) manufacturers of multi-lingual software. Fortunately, the participants of both projects realized around 1991 that two different unified character sets is not what the world needs. They joined their efforts and worked together on creating a single code table. Both projects still exist and publish their respective standards independently, however the Unicode Consortium and ISO/IEC JTC1/SC2 have agreed to keep the code tables of the Unicode and ISO 10646 standards compatible and they closely coordinate any further extensions. Unicode 1.1 corresponds to ISO 10646-1:1993 and Unicode 3.0 corresponds to ISO 10646-1:2000.

In GOCR, we adopted the Unicode Standard version 3.0. To the programmer using GOCR, this is a very simple way to deal with characters that are not in the ASCII or the ISO-8859-1 table, and let one to support any language.

Support in GOCR is very simple, as it should be. There's a list #defining some of the characters in `unicode.h`. Note that only a small portion of the Unicode set is present there, which reflect what we hope to be able to recognize in the near future, and what we already do. If you need to support other characters not found there, please feel free to. Be sure to use their correct codes; you can get a full list of them in:

```
http://www.unicode.org
```

and if you notify us, we add them to the header. As GOCR treats the codes as simple numbers, it doesn't matter if it's in the header or not. The only problem you may find is with the outputFormatter plugin, which may not support some characters.

GOCR provides a simple function that helps to compose characters and accents:

```
wchar_t gocr_compose ( wchar_t main, wchar_t modifier );
```

First, `wchar_t` is the type to use with wide characters. People are still discussing how many bytes it should be, so some systems may define it to be 2-byte and others to be 4-byte. Eventually it will be 4-byte everywhere.

Now the arguments: `main` is the character, and `modifier` is the accent; the function returns the code of the accented character. Example:

```
character = gocr_compose( a, ACUTE_ACCENT );
```

returns the code of the character á. Currently this function supports the following:

Modifier	Characters
ACUTE_ACCENT	aeiouy AEIOUY
CEDILLA	c C
TILDE	ano ANO
GRAVE_ACCENT	aeiou AEIOU
DIAERESIS	aeiouy AEIOUY
CIRCUMFLEX_ACCENT	aeiou AEIOU
RING_ABOVE	a A
e or E (æ, œ)	ao AO

Besides that, it also supports a latin→greek character translation, if you pass 'g' as `modifier`. See the table for reference.

Latin	Greek
a	α
b	β
g	γ
d	δ
e	ϵ
z	ζ
h	η
q	θ
i	ι
k	κ
l	λ
m	μ
n	ν
x	ξ
o	o
p	π
r	ρ
&	ς
s	σ
t	τ
y	υ
f	ϕ
c	χ
v	ψ
w	ω

Table 3.1: Latin→greek reference for gocr_compose.

If `main` is a capital letter, the returning characters will also be capital letters. Support of greek accents (tonos, dialytika, etc) is under way.

3.6.2 Setting characters

When you are ready to add a character, use:

```
int gocr_boxCharSet ( gocrBox *b, wchar_t w, float prob );
```

The arguments are:

b the box you are processing.

w the character code.

prob the probability that the recognition is correct: 0.0 is none (which will take the character out of the list) and 1.0 is 100% sure.

The most probable character will be returned later, etc

3.6.3 Attributes again

The charFinder may not have found all the attributes of a character. Don't worry: this module may access the `gocr_charAttributeSet` too.

talk about using charAttribute funcitons here too, and how is gocrBox importnat here

3.7 contextCorrection

After everything, there will remain some characters that weren't recognized, and it's the task of this module to recognize them. These characters can be divided in three groups[6]:

- merged characters. Due to imperfections of the original text, two or more characters ended touching it other, and should be separated. Ligatures may fall in this group too.
- unsupported characters. There's not much to do with these; they just are not supported by any of the modules.
- unrecognizable characters. Bad printing, bad scanning or some accident with the original document could have rendered some of the characters unrecognizable. They can be recognized by using some filter and reprocessing, or to use the context.

So, these are the issues you must consider.

3.7.1 Accessing text

TODO.

[6] It's widely known that there are two types of people, those who separate people in two groups and those who don't. You might argue that there are three groups: those who separate people in three groups, those who don't separate people in groups, and those who can't decide. But then there are four groups: you must include those who separate people in two groups. And, since we are separating people in four groups, there is a fifth group. The problem is those idiots that can't make up their minds.

3.7.2 Splitting characters

GOCR provides a set of functions similar, or better, (almost) identical to those used to create characters to split them.

Let's take a look of the situation: you have added a character that you later find out is in fact composed of two (or more, but let's assume two for simplicity; you can take care of more applying this procedure several times) characters. How to split them? Although you could delete the box, taking care of saving its attributes, create two new characters, etc, there's an easier way to do it:

```
int gocr_charSplitBegin ( gocrBox *box );
```

box the box to be splited.

Now you can work just as if you were adding a character. All the `gocr_charSet` functions can be used as usual. When you are done, call

```
int gocr_charSplitEnd ( void );
```

It's time for the fine print. First, what happens: all the pixels you select will be part of the a new box. This box is inserted in the list *before* the original one, which is updated to hold the rest of the pixels only. All attributes that were part of the original box are now transferred to the new one (so, the original one doesn't have any attributes anymore; but since they are applied to the box before it, they are applied to it too). You can call `gocr_Abort` just as if you were adding a character.

Future: there may be a flag to set which of the boxes goes before which.

3.7.3 Joining characters

Still not planned.

3.8 outputFormatter

Once it's all done, the user usually wants the output sent to a file in some way that he/she can read it, instead of the beautiful, complex structures that are spread all over the computer memory. This module should satisfy this caprice. The prototype is:

```
void (*outputFormatter) (List *bl, void *v);
```

where the list contains all the blocks, in the order you added them. This module may be changed in the near future.

Each block has a field called `text` which contains all the characters of the block and the attributes. If you just want to dump them, lousily converted to ascii, here's an example of what you may do:

```
for_each_data(bl) {

    wchar_t *w = ((gocrBlock *)list_get_current(bl))->text;
    while (*w)
```

```
        putc(*w++);
} end_for_each(bl);
```

You can read more about lists in section 4.2.

3.8.1 Dealing with unknown characters

Since the user may be using any modules available, it's possible that they recognize some characters that are not supported by the outputFormatter function. Some may be not even in the UNICODE standard.

We suggest three ways to deal with this situation. The first is to print the code in a readable format: U39A0, for example. The user probably can find what character is this, and using and editor easily replace the code by whatever he wants.

The second suggestion is to let the user provide some mappings of his own, either by a configuration file or by using the gocr_setModuleAttribute (see 2.4.3). This is our preferred solution, since it allows user customization with minimum effort.

The third suggestion is to ask the user on the fly.

3.8.2 Dealing with unknown attributes

TODO

Chapter 4

Modules in deep

While last chapter focused in an overview of what you have to do, this chapter presents utilities that are part of the GOCR module API, written to make your life a bit easier.

4.1 Printing image, blocks and boxes

GOCR provides a number of functions that print images, blocks or boxes, which are very helpful for debugging. How the image is printed depends of the PRINT attribute and the output file is controlled by the ERROR_FILE attribute (see section 2.2).

```
int gocr_printBlock ( gocrBlock *b );
```

Prints all information in `gocrBlock *b`, if PRINT_IMAGE is GOCR_TRUE, prints framed image too. Here's an example of what is printed (PRINT = 0):

```
Block: x0:1, y0:1, x1:117, y1:16; type TEXT
..**........******......*******..........**.....********..
****.......*....***....*.....**..........**.....*******...
..**......**.....***...**....***........***.....*.........
..**......***....***...**....***.......****.....*.........
..**......***.....**.........**........*.**.....*.........
..**.......**....***........***.......**.**.....*..***....
..**.............***.......***.......**..**.....****.**...
..**............***......*****.......*...**.....**....**..
..**............***.........***.....**...**...........***.
..**...........***...........***....*....**............**.
..**..........***............***...*.....**............**.
..**.........**.......***.....**...***********.***.....**.
..**.........*.....*..***.....**.........**....***....***.
..**........*......*..***....***.........**....**.....***.
..**.......*********..**.....***.........**.....*.....**..
*******...**********...***..***........*******..***.***...
```

Same for boxes:

```
int gocr_printBox ( gocrBox *b );
```

prints all information in `gocrBox *b`; if PRINT_IMAGE is GOCR_TRUE, prints framed image too.

```
int gocr_printBox2 ( gocrBox *b1, gocrBox *b2 );
```

Prints two boxes, side by side. Neat for that quick check of what the heck is going wrong.

```
int gocr_printArea ( gocrImage *image, int x0,
int y0, int x1, int y1 );
```

Prints the part of the `image` framed by the (x0, y0) and (x1, y1) coordinates.

4.2 Linked lists

Internally, GOCR abuses of linked lists to store information. They are very useful for this kind of program, and you may need them. Include `list.h`, and take advantage of our linked list functions, which were thoroughly tested! FREE!

```
void list_init ( List *l );
```

Must be called before you do any operations with the list, otherwise strange behaviors may occur. It doesn't not allocate memory, and so must received a non-NULL pointer.

```
int list_app ( List *l, void *data );
```

Appends an element `data` to the end of the list. Returns 0 if OK, 1 otherwise.

```
int list_del ( List *l, void *data );
```

Deletes the node containing data. Use carefully. See `for_each_data`, below.

```
int list_empty ( List *l );
```

Returns 1 if the list is empty, 0 otherwise.

```
void list_free ( List *l );
```

Frees the list structure and nodes. Does not free the data stored in it.

```
void *list_get_current(l) ( List *l );
```

Returns the data in the current node. See `for_each_data`, below.

```
void *list_get_cur_prev(l) ( List *l );
```

Returns the data stored before the current node. See `for_each_data`, below.

```
void *list_get_cur_next(l) ( List *l );
```

Returns the data stored after the current node. See `for_each_data`, below.

```
void *list_get_header ( List *l );
```

Returns the data in the first node.

```
void *list_get_tail(l) ( List *l );
```

Returns the data in the last node.

```
int list_ins ( List *l, void *data_after, void *data );
```

Inserts `data` before `data_after`.

```
void * list_next ( List *l, void *data );
```

Returns the data stored after `data`.

```
void * list_prev ( List *l, void *data );
```

Returns the data stored before `data`.

```
void list_sort(List *l, int (*compare)(const void *, const void *));
```

Similar to qsort: sorts the list. `compare` function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined. Uses a bubble sort to do the task.

```
int list_total ( List *l );
```

Returns the total number of nodes in the linked list.

```
for_each_data ( List *l ) {
    code
} end_for_each ( List *l );
```

This piece of code implements a for that sweeps the entire list, node by node. You can get the current node data using `list_get_current`, the data before it using `list_get_cur_prev`, and the data after it using `list_get_cur_next`. Use these functions if possible instead of `list_next` and `list_prev`, since they are much faster.

You can nest `for_each_data`, but take care when you call `list_del`, since you may be deleting one of the nodes that is the current one in a lower level. The internal code takes care of access to previous/next elements of the now defunct node. Here's an example:

```
for_each_data(l) {
    for_each_data(l) {
       list_del(l, header_data);
       free(header_data);
```

```
        } end_for_each(l);
        tempnode = list_cur_next(l);
    } end_for_each(l);
```

Although you have deleted the current node of the outer loop, the line in italic will work as if nothing happened. But if it's replaced with:

```
    tempnode = list_next(l, list_get_current(l));
```

the code will break, since `list_get_current` will return either NULL or some garbage. The best way to avoid this problem is not using `list_del` in a big stack of loops, or test the return value of `list_get_current()`. You can use `break` and `continue`, just as if you were in a normal for loop, but *never* use a goto to somewhere outside the loop (theoretically you can do it, using the `list_lower` function explained below, but if you do **take care**).

Note: if you have two elements with the same data, the functions will assume that the first one is the wanted one. Not a bug, a feature.

Another note: avoid calling `list_prev` and `list_next`. They are intensive and slow functions. Keep the result in a variable or, if you need something more, use `list_get_element_from_data`, described below.

4.2.1 Internal list functions

There are some functions that are used internally, but may be used by you to do some clever optimizations. Note that, if not used correctly, you may break the code.

```
    Element *list_element_from_data ( List *l, void *data );
```

Given a data, returns the Element it's stored in. Element is a structure:

```
    struct element {
        struct element *next, *previous;
        void *data;
    };
    typedef struct element Element;
```

This may be interesting if you need to access the next and previous nodes several times and you are not using a `for_each_data`, i.e., you need to use `list_next` and `list_prev` heavily.

```
    int list_higher_level ( List *l );
    void list_lower_level ( List *l );
```

These functions are used internally by `for_each_data` and should not be directly called by the user.

4.3 Hash tables

Hash tables are used internally to access string arrays (which are used to save attributes that are created in real time, for example), and may be useful to you. The functions provided are not as flexible as the linked list ones, but should suffice for most uses. Remember to include `hash.h`.

```
int hash_init ( HashTable *t, int size, int (*hash_func)(char *));
```

Initialize a hash table, with `size` entries, using `hash_func` as the hash generator func. If `t` is NULL, the function automatically mallocs memory for it. If `hash_func` is NULL, the default internal hash generator is used. Returns -1 on error, 0 if OK.

```
int hash_insert ( HashTable *t, char *key, void *data );
```

Inserts a new entry in table `t`, with key `key`, which will contain `data`. Returns -1 on error, -2 if the data already exists, or the hash if everything was OK (although theoretically the hash should be hidden from the user, etc, it's used internally by GOCR to store character attributes. You can safely ignore the hash, and use if (hash_insert()) < 0 { error}).

```
void *hash_del ( HashTable *t, char *key );
```

Deletes the entry associated with the `key`. Returns a pointer to the data structure, which is not freed.

```
void *hash_data ( HashTable *t, char *key );
```

Returns the a pointer to the data associated associated with `key.`

```
int hash_free ( HashTable *t, void (*free_func)(void *));
```

Frees the hash table contents. If `free_func` is not NULL, it's called for every data stored in the table. Does not free the hash table structure itself.

```
char *hash_key ( HashTable *t, void *data );
```

Searches the hash table for the first ocurrence of data, and returns the corresponding key.

Chapter 5

Troubleshooting

No matter how hard we developers work, writing perfect code, computers stubbornly do not adapt to our code and insist in showing bugs and problems.

Q: I'm having NetPBM problems.
Q: The compiler issues several warning about enum pm_check.
Q: Image input or output is not working correctly.
A: These are very likely to be result of a bad NetPBM install.

For some reason, many Linux distributions still come with old NetPBM libraries. They lack functionality that GOCR could use, and probably have bugs that were already fixed. That would not be so bad if it were not for another problem: if you download the latest NetPBM package (http://netpbm.sourceforge.net), and do a `make install`, (at least in my computer) the install is not complete. Besides the usual problem of things going to /usr, /usr/local/, /usr/local/share, etc, possibly resulting in keeping the old libraries and executables, the Makefile doesn't install the headers. This will lead to the `enum pm_check` warnings, which seem kind of harmless, but end up messing everything. Solution: manually install the new headers (which are: pnm.h, pam.h, pbm.h, pgm.h, ppm.h, pbmplus.h and shhopt.h), and make sure that the old libraries are deleted (or at least that symbolic links point to the new ones).

Q: Why TRUE is defined as 0x22A8 (8872 in decimal)?
A: Because UNICODE defines the symbol $\models$ as TRUE, as code 0x22A8. If you need to use boolean values, use GOCR_TRUE and GOCR_FALSE, which are what you want.

Chapter 6

Notes

These chapter contains internal notes to remind myself. Disregard them.

6.1 image

finish the IO functions (support non-pam lib)

6.2 Blocks

New architecture: instead of gocr_addBlock, use the gocr_beginBlock(geometry type) paradigm. Probably only in next version.

6.3 charFinder

gocr_endCharacter() may or may not automatically call charRecognizer. Set a flag to do it.

How to save boxes? In a linked list in the gocrBlock structure? Otherwise, it's reponsability of the user?

6.4 Characters recognizer

images Passed as copies, to improve speed with use of processor's cache. They are called/stored by gocr_endCharacter() (on flag, see above) ???

Finish charSplit[Begin/End]

how to store the characters? A wchar_t *data is very inconvenient. Perhaps a linked list, paging the text. Probably wrapper functions.

Take care of the char attributes in unicode.c

6.5 contextcorrection

Let it access the characters without seeing internal codes (E0XX-EXXX). Should it never see the attributes? I think that knowledge such as "this is in italic" may be helpful. But using ispell will require conversion to text, which is not straight forward and should be done by outputFormatter.

6.6 outputformatter

It should get the text preferably in one big chunk.

Index