

# Ruby Hacking Guide

## Preface

---

This book explores several themes with the following goals in mind:

- To have knowledge of the structure of `ruby`
- To gain knowledge about language processing systems in general
- To acquire skills in reading source code

Ruby is an object-oriented language developed by Yukihiro Matsumoto. The official implementation of the Ruby language is called `ruby`. It is actively developed and maintained by the open source community. Our first goal is to understand the inner-workings of the `ruby` implementation. This book is going to investigate `ruby` as a whole.

Secondly, by knowing about the implementation of Ruby, we will be able to know about other language processing systems. I tried to cover all topics necessary for implementing a language, such as hash table, scanner and parser, evaluation procedure, and many others. Because this book is not intended as a text book, going through entire areas and ideas without any lack was not

reasonable. However the parts relating to the essential structures of a language implementation are adequately explained. And a brief summary of Ruby language itself is also included so that readers who don't know about Ruby can read this book.

The main themes of this book are the first and the second point above. Though, what I want to emphasize the most is the third one: To acquire skill in reading source code. I dare to say it's a "hidden" theme. I will explain why I thought it is necessary.

It is often said "To be a skilled programmer, you should read source code written by others." This is certainly true. But I haven't found a book that explains how you can actually do it. There are many books that explain OS kernels and the interior of language processing systems by showing the concrete structure or "the answer," but they don't explain the way to reach that answer. It's clearly one-sided.

Can you, perhaps, naturally read code just because you know how to write a program? Is it true that reading codes is so easy that all people in this world can read code written by others with no sweat? I don't think so. Reading programs is certainly as difficult as writing programs.

Therefore, this book does not simply explain `ruby` as something already known, rather demonstrate the analyzing process as graphic as possible. Though I think I'm a reasonably seasoned Ruby programmer, I did not fully understand the inner structure of `ruby` at the time when I started to write this book. In other words,

regarding the content of ruby, I started from the position as close as possible to readers. This book is the summary of both the analyzing process started from that point and its result.

I asked Yukihiro Matsumoto, the author of ruby, for supervision. But I thought the spirit of this book would be lost if each analysis was monitored by the author of the language himself. Therefore I limited his review to the final stage of writing. In this way, without loosing the sense of actually reading the source codes, I think I could also assure the correctness of the contents.

To be honest, this book is not easy. In the very least, it is limited in its simplicity by the inherent complexity of its aim. However, this complexity may be what makes the book interesting to you. Do you find it interesting to be chattering around a piece of cake? Do you take to your desk to solve a puzzle that you know the answer to in a heartbeat? How about a suspense novel whose criminal you can guess halfway through? If you really want to come to new knowledge, you need to solve a problem engaging all your capacities. This is the book that lets you practice such idealism exhaustively. “It’s interesting because it’s difficult.” I’m glad if the number of people who think so will increase because of this book.

## **Target audience**

---

Firstly, knowledge about the Ruby language isn’t required.

However, since the knowledge of the Ruby language is absolutely necessary to understand certain explanations of its structure, supplementary explanations of the language are inserted here and there.

Knowledge about the C language is required, to some extent. I assume you can allocate some structs with `malloc()` at runtime to create a list or a stack and you have experience of using function pointers at least a few times.

Also, since the basics of object-oriented programming will not be explained so seriously, without having any experience of using at least one of object-oriented languages, you will probably have a difficult time. In this book, I tried to use many examples in Java and C++.

## **Structure of this book**

---

This book has four main parts:

Part 1: Objects

Part 2: Syntactic analysis

Part 3: Evaluation

Part 4: Peripheral around the evaluator

Supplementary chapters are included at the beginning of each part when necessary. These provide a basic introduction for those who

are not familiar with Ruby and the general mechanism of a language processing system.

Now, we are going through the overview of the four main parts. The symbol in parentheses after the explanation indicates the difficulty gauge. They are (C), (B), (A) in order of easy to hard, (S) being the highest.

## Part 1: Object

Chapter1 Focuses the basics of Ruby to get ready to accomplish Part 1. (C)

Chapter2 Gives concrete inner structure of Ruby objects. (C)

Chapter3 States about hash table. (C)

Chapter4 Writes about Ruby class system. You may read through this chapter quickly at first, because it tells plenty of abstract stories. (A)

Chapter5 Shows the garbage collector which is responsible for generating and releasing objects. The first story in low-level series. (B)

Chapter6 Describes the implementation of global variables, class variables, and constants. (C)

Chapter7 Outline of the security features of Ruby. (C)

## Part 2: Syntactic analysis

Chapter8 Talks about almost complete specification of the Ruby language, in order to prepare for Part 2 and Part 3. (C)

Chapter9 Introduction to yacc required to read the syntax file at least. (B)

Chapter10 Look through the rules and physical structure of the parser. (A)

Explore around the peripherals of `lex_state`, which is Chapter11 the most difficult part of the parser. The most difficult part of this book. (S)

Chapter12 Finalization of Part 2 and connection to Part 3. (C)

## Part 3: Evaluator

Chapter13 Describe the basic mechanism of the evaluator. (C)

Chapter14 Reads the evaluation stack that creates the main context of Ruby. (A)

Chapter15 Talks about search and initialization of methods. (B)

Chapter16 Defies the implementation of the iterator, the most characteristic feature of Ruby. (A)

Chapter17 Describe the implementation of the eval methods. (B)

## Part 4: Peripheral around the evaluator

Chapter18 Run-time loading of libraries in C and Ruby. (B)

Chapter19 Describes the implementation of thread at the end of the core part. (A)

## Environment

---

This book describes on ruby 1.7.3 2002-09-12 version. It's attached on the CD-ROM. Choose any one of `ruby-rhg.tar.gz`, `ruby-rhg.lzh`, or `ruby-rhg.zip` according to your convenience. Content is the same for all. Alternatively you can obtain from the support site (footnote{<http://i.loveruby.net/ja/rhg/>}) of this book.

For the publication of this book, the following build environment was prepared for confirmation of compiling and testing the basic operation. The details of this build test are given in `doc/buildtest.html` in the attached CD-ROM. However, it doesn't necessarily assume the probability of the execution even under the same environment listed in the table. The author doesn't guarantee in any form the execution of `ruby`.

- BeOS 5 Personal Edition/i386
- Debian GNU/Linux potato/i386
- Debian GNU/Linux woody/i386
- Debian GNU/Linux sid/i386
- FreeBSD 4.4-RELEASE/Alpha (Requires the local patch for this book)
- FreeBSD 4.5-RELEASE/i386
- FreeBSD 4.5-RELEASE/PC98
- FreeBSD 5-CURRENT/i386
- HP-UX 10.20
- HP-UX 11.00 (32bit mode)
- HP-UX 11.11 (32bit mode)
- Mac OS X 10.2
- NetBSD 1.6F/i386
- OpenBSD 3.1
- Plamo Linux 2.0/i386
- Linux for PlayStation2 Release 1.0
- Redhat Linux 7.3/i386
- Solaris 2.6/Sparc
- Solaris 8/Sparc

- UX/4800
- Vine Linux 2.1.5
- Vine Linux 2.5
- VineSeed
- Windows 98SE (Cygwin, MinGW+Cygwin, MinGW+MSYS)
- Windows Me (Borland C++ Compiler 5.5, Cygwin, MinGW+Cygwin, MinGW+MSYS, Visual C++ 6)
- Windows NT 4.0 (Cygwin, MinGW+Cygwin)
- Windows 2000 (Borland C++ Compiler 5.5, Visual C++ 6, Visual C++.NET)
- Windows XP (Visual C++.NET, MinGW+Cygwin)

These numerous tests aren't of a lone effort by the author. Those test build couldn't be achieved without magnificent cooperations by the people listed below.

I'd like to extend warmest thanks from my heart.

Tietew

kjana

nyasu

sakazuki

Masahiro Sato

Kenichi Tamura

Morikyu

Yuya Kato

Yasuhiro Kubo

Kentaro Goto

Tomoyuki Shimomura

Masaki Sukeda

Koji Arai

Kazuhiro Nishiyama

Shinya Kawaji

Tetsuya Watanabe

Naokuni Fujimoto

However, the author owes the responsibility for this test. Please refrain from attempting to contact these people directly. If there's any flaw in execution, please be advised to contact the author by e-mail: [aamine@loveruby.net](mailto:aamine@loveruby.net).

## Web site

---

The web site for this book is <http://i.loveruby.net/ja/rhg/>. I will add information about related programs and additional documentation, as well as errata. In addition, I'm going to publisize the first few chapters of this book at the same time of the release. I will look for a certain circumstance to publicize more chapters, and the whole contents of the book will be at this website at the end.

## Acknowledgment

---

First of all, I would like to thank Mr. Yukihiro Matsumoto. He is

the author of Ruby, and he made it in public as an open source software. Not only he willingly approved me to publish a book about analyzing ruby, but also he agreed to supervise the content of it. In addition, he helped my stay in Florida with simultaneous translation. There are plenty of things beyond enumeration I have to say thanks to him. Instead of writing all the things, I give this book to him.

Next, I would like to thank arton, who proposed me to publish this book. The words of arton always moves me. One of the things I'm currently struggled due to his words is that I have no reason I don't get a .NET machine.

Koji Arai, the 'captain' of documentation in the Ruby society, conducted a scrutiny review as if he became the official editor of this book while I was not told so. I thank all his review.

Also I'd like to mention those who gave me comments, pointed out mistakes and submitted proposals about the construction of the book throughout all my work.

Tietew, Yuya, Kawaji, Gotoken, Tamura, Funaba, Morikyu, Ishizuka, Shimomura, Kubo, Sukeda, Nishiyama, Fujimoto, Yanagawa, (I'm sorry if there's any people missing), I thank all those people contributed.

As a final note, I thank Otsuka , Haruta, and Kanemitsu who you for arranging everything despite my broke deadline as much as four times, and that the manuscript exceeded 200 pages than

originally planned.

I cannot expand the full list here to mention the name of all people contributed to this book, but I say that I couldn't successfully publish this book without such assistance. Let me take this place to express my appreciation. Thank you very much.

Minero Aoki

If you want to send remarks, suggestions and reports of typographical errors, please address to [Minero Aoki <aamine@loveruby.net>](mailto:aamine@loveruby.net) .

“Rubyソースコード完全解説” can be reserved/ordered at ImpressDirect. ([Jump to the introduction page](#))

Copyright © 2002-2004 Minero Aoki, All rights reserved.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](#)

# Ruby Hacking Guide

## Introduction

### **Characteristics of Ruby**

---

Some of the readers may have already been familiar with Ruby, but (I hope) there are also many readers who have not. First let's go though a rough summary of the characteristics of Ruby for such people.

Hereafter capital “Ruby” refers to Ruby as a language specification, and lowercase “ruby” refers to `ruby` command as an implementation.

### **Development style**

Ruby is a language that is being developped by the hand of Yukihiro Matsumoto as an individual. Unlike C or Java or Scheme, it does not have any standard. The specification is merely shown as an implementation as `ruby`, and its varying continuously. For good

or bad, it's free.

Furthermore `ruby` itself is a free software. It's probably necessary to mention at least the two points here: The source code is open in public and distributed free of charge. Thanks to such condition, an attempt like this book can be approved.

If you'd like to know the exact licence, you can read `README` and `LEGAL`. For the time being, I'd like you to remember that you can do at least the following things:

- You can redistribute source code of `ruby`
- You can modify source code of `ruby`
- You can redistribute a copy of source code with your modification

There is no need for special permission and payment in all these cases.

By the way, the purpose of this book is to read the original `ruby`, thus the target source is the one not modified unless it is particularly specified. However, white spaces, new lines and comments were added or removed without asking.

## It's conservative

Ruby is a very conservative language. It is equipped with only carefully chosen features that have been tested and washed out in a variety of languages. Therefore it doesn't have plenty of fresh and

experimental features very much. So it has a tendency to appeal to programmers who put importance on practical functionalities. The dyed-in-the-wool hackers like Scheme and Haskell lovers don't seem to find appeal in ruby, at least in a short glance.

The library is conservative in the same way. Clear and unabbreviated names are given for new functions, while names that appears in C and Perl libraries have been taken from them. For example, `printf`, `getpwent`, `sub`, and `tr`.

It is also conservative in implementation. Assembler is not its option for seeking speed. Portability is always considered a higher priority when it conflicts with speed.

## **It is an object-oriented language**

Ruby is an object-oriented language. It is absolutely impossible to exclude it from the features of Ruby.

I will not give a page to this book about what an object-oriented language is. To tell about an object-oriented feature about Ruby, the expression of the code that just going to be explained is the exact sample.

## **It is a script language**

Ruby is a script language. It seems also absolutely impossible to exclude this from the features of Ruby. To gain agreement of everyone, an introduction of Ruby must include “object-oriented”

and “script language”.

However, what is a “script language” for example? I couldn’t figure out the definition successfully. For example, John K. Ousterhout, the author of Tcl/Tk, gives a definition as “executable language using #! on UNIX”. There are other definitions depending on the view points, such as one that can express a useful program with only one line, or that can execute the code by passing a program file from the command line, etc.

However, I dare to use another definition, because I don’t find much interest in “what” a script language. I have the only one measure to decide to call it a script language, that is, whether no one would complain about calling it a script language. To fulfill this definition, I would define the meaning of “script language” as follows.

A language that its author calls it a “script language”.

I’m sure this definition will have no failure. And Ruby fulfills this point. Therefore I call Ruby a “script language”.

## **It’s an interpreter**

ruby is an interpreter. That’s the fact. But why it’s an interpreter? For example, couldn’t it be made as a compiler? It must be because in some points being an interpreter is better than being a compiler ... at least for ruby, it must be better. Well, what is good about being an interpreter?

As a preparation step to investigating into it, let's start by thinking about the difference between an interpreter and a compiler. If the matter is to attempt a theoretical comparison in the process how a program is executed, there's no difference between an interpreter language and a compile language. Because it works by letting CPU interpret the code compiled to the machine language, it may be possible to say it works as an interpretor. Then where is the place that actually makes a difference? It is a more practical place, in the process of development.

I know somebody, as soon as hearing “in the process of development”, would claim using a stereotypical phrase, that an interpreter reduces effort of compilation that makes the development procedure easier. But I don't think it's accurate. A language could possibly be planned so that it won't show the process of compilation. Actually, Delphi can compile a project by hitting just F5. A claim about a long time for compilation is derived from the size of the project or optimization of the codes. Compilation itself doesn't owe a negative side.

Well, why people perceive an interpreter and compiler so much different like this? I think that it is because the language developers so far have chosen either implementation based on the trait of each language. In other words, if it is a language for a comparatively small purpose such as a daily routine, it would be an interpretor. If it is for a large project where a number of people are involved in the development and accuracy is required, it would be a compiler. That may be because of the speed, as well as the ease of

creating a language.

Therefore, I think “it’s handy because it’s an interpreter” is an outsized myth. Being an interpreter doesn’t necessarily contribute the readiness in usage; seeking readiness in usage naturally makes your path toward building an interpreter language.

Anyway, `ruby` is an interpreter; it has an important fact about where this book is facing, so I emphasize it here again. Though I don’t know about “it’s handy because it is an interpreter”, anyway `ruby` is implemented as an interpreter.

## High portability

Even with a problem that fundamentally the interfaces are Unix-centered, I would insist `ruby` possesses a high portability. It doesn’t require any extremely unfamiliar library. It has only a few parts written in assembler. Therefore porting to a new platform is comparatively easy. Namely, it works on the following platforms currently.

- Linux
- Win32 (Windows 95, 98, Me, NT, 2000, XP)
- Cygwin
- djgpp
- FreeBSD
- NetBSD
- OpenBSD

- BSD/OS
- Mac OS X
- Solaris
- Tru64 UNIX
- HP-UX
- AIX
- VMS
- UX/4800
- BeOS
- OS/2 (emx)
- Psion

I heard that the main machine of the author Matsumoto is Linux. Thus when using Linux, you will not fail to compile any time.

Furthermore, you can expect a stable functionality on a (typical) Unix environment. Considering the release cycle of packages, the primary option for the environment to hit around `ruby` should fall on a branch of PC UNIX, currently.

On the other hand, the Win32 environment tends to cause problems definitely. The large gaps in the targeting OS model tend to cause problems around the machine stack and the linker. Yet, recently Windows hackers have contributed to make better support. I use a native ruby on Windows 2000 and Me. Once it gets successfully run, it doesn't seem to show special concerns like frequent crashing. The main problems on Windows may be the gaps in the specifications.

Another type of OS that many people may be interested in should probably be Mac OS (prior to v9) and handheld OS like Palm.

Around ruby 1.2 and before, it supported legacy Mac OS, but the development seems to be in suspension. Even a compiling can't get through. The biggest cause is that the compiler environment of legacy Mac OS and the decrease of developers. Talking about Mac OS X, there's no worries because the body is UNIX.

There seem to be discussions the portability to Palm several branches, but I have never heard of a successful project. I guess the difficulty lies in the necessity of settling down the specification-level standards such as `stdio` on the Palm platform, rather than the processes of actual implementation. Well I saw a porting to Psion has been done. ([ruby-list:36028]).

How about hot stories about VM seen in Java and .NET? Because I'd like to talk about them combining together with the implementation, this topic will be in the final chapter.

## **Automatic memory control**

Functionally it's called GC, or Garbage Collection. Saying it in C-language, this feature allows you to skip `free()` after `malloc()`. Unused memory is detected by the system automatically, and will be released. It's so convenient that once you get used to GC you won't be willing to do such manual memory control again.

The topics about GC have been common because of its popularity

in recent languages with GC as a standard set, and it is fun that its algorithms can still be improved further.

## Typeless variables

The variables in Ruby don't have types. The reason is probably typeless variables conforms more with polymorphism, which is one of the strongest advantages of an object-oriented language. Of course a language with variable type has a way to deal with polymorphism. What I mean here is a typeless variables have better conformance.

The level of "better conformance" in this case refers to synonyms like "handy". It's sometimes corresponds to crucial importance, sometimes it doesn't matter practically. Yet, this is certainly an appealing point if a language seeks for "handy and easy", and Ruby does.

## Most of syntactic elements are expressions

This topic is probably difficult to understand instantly without a little supplemental explanation. For example, the following C-language program results in a syntactic error.

```
result = if (cond) { process(val); } else { 0; }
```

Because the C-language syntax defines `if` as a statement. But you

can write it as follows.

```
result = cond ? process(val) : 0;
```

This rewrite is possible because the conditional operator (a?b:c) is defined as an expression.

On the other hand, in Ruby, you can write as follows because `if` is an expression.

```
result = if cond then process(val) else nil end
```

Roughly speaking, if it can be an argument of a function or a method, you can consider it as an expression.

Of course, there are other languages whose syntactic elements are mostly expressions. Lisp is the best example. Because of the characteristic around this, there seems many people who feel like “Ruby is similar to Lisp”.

## Iterators

Ruby has iterators. What is an iterator? Before getting into iterators, I should mention the necessity of using an alternative term, because the word “iterator” is disliked recently. However, I don’t have a good alternative. So let us keep calling it “iterator” for the time being.

Well again, what is an iterator? If you know higher-order function, for the time being, you can regard it as something similar to it. In C-language, the counterpart would be passing a function pointer as an argument. In C++, it would be a method to which the operation part of STL's `Iterator` is enclosed. If you know sh or Perl, it's good to imagine something like a custom `for` statement which we can define.

Yet, the above are merely examples of “similar” concepts. All of them are similar, but they are not identical to Ruby's iterator. I will expand the precise story when it's a good time later.

## Written in C-language

Being written in C-language is not notable these days, but it's still a characteristic for sure. At least it is not written in Haskell or PL/I, thus there's the high possibility that the ordinary people can read it. (Whether it is truly so, I'd like you confirm it by yourself.)

Well, I just said it's in C-language, but the actual language version which ruby is targetting is basically K&R C. Until a little while ago, there were a decent number of – not plenty though – K&R-only-environment. But recently, there are a few environments which do not accept programs written in ANSI C, technically there's no problem to move on to ANSI C. However, also because of the author Matsumoto's personal preference, it is still written in K&R style.

For this reason, the function definition is all in K&R style, and the prototype declarations are not so seriously written. If you carelessly specify `-Wall` option of `gcc`, there would be plenty of warnings shown. If you try to compile it with a C++ compiler, it would warn prototype mismatch and could not compile. ... These kind of stories are often reported to the mailing list.

## Extension library

We can write a Ruby library in C and load it at runtime without recompiling Ruby. This type of library is called “Ruby extension library” or just “Extension library”.

Not only the fact that we can write it in C, but the very small difference in the code expression between Ruby-level and C-level is also a significant trait. As for the operations available in Ruby, we can also use them in C in the almost same way. See the following example.

```
# Method call
obj.method(arg)                                # Ruby
rb_funcall(obj, rb_intern("method"), 1, arg);    # C

# Block call
yield arg          # Ruby
rb_yield(arg);     # C

# Raising exception
raise ArgumentError, 'wrong number of arguments'  # Ruby
rb_raise(rb_eArgError, "wrong number of arguments"); # C

# Generating an object
```

```
arr = Array.new          # Ruby
VALUE arr = rb_ary_new(); # C
```

It's good because it provides easiness in composing an extension library, and actually it makes an indispensable prominence of ruby. However, it's also a burden for ruby implementation. You can see the affects of it in many places. The affects to GC and thread-processing is eminent.

## Thread

Ruby is equipped with thread. Assuming a very few people knowing none about thread these days, I will omit an explanation about the thread itself. I will start a story in detail.

ruby's thread is a user-level thread that is originally written. The characteristic of this implementation is a very high portability in both specification and implementation. Surprisingly a MS-DOS can run the thread. Furthermore, you can expect the same response in any environment. Many people mention that this point is the best feature of ruby.

However, as a trade off for such an extremeness of portability, ruby abandons the speed. It's, say, probably the slowest of all user-level thread implementations in this world. The tendency of ruby implementation may be seen here the most clearly.

# **Technique to read source code**

---

Well. After an introduction of ruby, we are about to start reading source code. But wait.

Any programmer has to read a source code somewhere, but I guess there are not many occasions that someone teaches you the concrete ways how to read. Why? Does it mean you can naturally read a program if you can write a program?

But I can't think reading the program written by other people is so easy. In the same way as writing programs, there must be techniques and theories in reading programs. And they are necessary. Therefore, before starting to ready ruby, I'd like to expand a general summary of an approach you need to take in reading a source code.

## **■ Principles**

At first, I mention the principle.

### **Decide a goal**

An important key to reading the source code is to set a concrete goal.

This is a word by the author of Ruby, Matsumoto. Indeed, his word is very convincing for me. When the motivation is a spontaneous

idea “May be I should read a kernel, at least...”, you would get source code expanded or explanatory books ready on the desk. But not knowing what to do, the studies are to be left untouched. Haven’t you? On the other hand, when you have in mind “I’m sure there is a bug somewhere in this tool. I need to quickly fix it and make it work. Otherwise I will not be able to make the deadline...”, you will probably be able to fix the code in a blink, even if it’s written by someone else. Haven’t you?

The difference in these two cases is motivation you have. In order to know something, you at least have to know what you want to know. Therefore, the first step of all is to figure out what you want to know in explicit words.

However, of course this is not all needed to make it your own “technique”. Because “technique” needs to be a common method that anybody can make use of it by following it. In the following section, I will explain how to bring the first step into the landing place where you achieve the goal finally.

## Visualising the goal

Now let us suppose that our final goal is set “Understand all about ruby”. This is certainly considered as “one set goal”, but apparently it will not be useful for reading the source code actually. It will not be a trigger of any concrete action. Therefore, your first job will be to drag down the vague goal to the level of a concrete thing.

Then how can we do it? The first way is thinking as if you are the

person who wrote the program. You can utilize your knowledge in writing a program, in this case. For example, when you are reading a traditional “structured” programming by somebody, you will analyze it hiring the strategies of structured programming too. That is, you will divide the target into pieces, little by little. If it is something circulating in a event loop such as a GUI program, first roughly browse the event loop then try to find out the role of each event handler. Or, try to investigate the “M” of MVC (Model View Controller) first.

Second, it's good to be aware of the method to analyze. Everybody might have certain analysis methods, but they are often done relying on experience or intuition. In what way can we read source codes well? Thinking about the way itself and being aware of it are crucially important.

Well, what are such methods like? I will explain it in the next section.

## Analysis methods

The methods to read source code can be roughly divided into two; one is a static method and the other is dynamic method. Static method is to read and analyze the source code without running the program. Dynamic method is to watch the actual behavior using tools like a debugger.

It's better to start studying a program by dynamic analysis. That is

because what you can see there is the “fact”. The results from static analysis, due to the fact of not running the program actually, may well be “prediction” to a greater or lesser extent. If you want to know the truth, you should start from watching the fact.

Of course, you don’t know whether the results of dynamic analysis are the fact really. The debugger could run with a bug, or the CPU may not be working properly due to overheat. The conditions of your configuration could be wrong. However, the results of static analysis should at least be closer to the fact than dynamic analysis.

## ■ **Dynamic analysis**

### **Using the target program**

You can’t start without the target program. First of all, you need to know in advance what the program is like, and what are expected behaviors.

### **Following the behavior using the debugger**

If you want to see the paths of code execution and the data structure produced as a result, it’s quicker to look at the result by running the program actually than to emulate the behavior in your brain. In order to do so easily, use the debugger.

I would be more happy if the data structure at runtime can be seen

as a picture, but unfortunately we can nearly scarcely find a tool for that purpose (especially few tools are available for free). If it is about a snapshot of the comparatively simpler structure, we might be able to write it out as a text and convert it to a picture by using a tool like graphviz\footnote{graphviz.....See doc/graphviz.html in the attached CD-ROM}. But it's very difficult to find a way for general purpose and real time analysis.

## Tracer

You can use the tracer if you want to trace the procedures that code goes through. In case of C-language, there is a tool named ctrace\footnote{ctrace.....<http://www.vicente.org/ctrace>}. For tracing a system call, you can use tools like strace\footnote{strace.....<http://www.wi.leidenuniv.nl/~wichert/strace/>}, truss, and ktrace.

## Print everywhere

There is a word “printf debugging”. This method also works for analysis other than debugging. If you are watching the history of one variable, for example, it may be easier to understand to look at the dump of the result of the print statements embed, than to track the variable with a debugger.

## Modifying the code and running it

Say for example, in the place where it's not easy to understand its

behavior, just make a small change in some part of the code or a particular parameter and then re-run the program. Naturally it would change the behavior, thus you would be able to infer the meaning of the code from it.

It goes without saying, you should also have an original binary and do the same thing on both of them.

## ■ **Static analysis**

### **The importance of names**

Static analysis is simply source code analysis. And source code analysis is really an analysis of names. File names, function names, variable names, type names, member names — A program is a bunch of names.

This may seem obvious because one of the most powerful tools for creating abstractions in programming is naming, but keeping this in mind will make reading much more efficient.

Also, we'd like to know about coding rules beforehand to some extent. For example, in C language, `extern` function often uses prefix to distinguish the type of functions. And in object-oriented programs, function names sometimes contain the information about where they belong to in prefixes, and it becomes valuable information (e.g. `rb_str_length`).

# Reading documents

Sometimes a document describes the internal structure is included. Especially be careful of a file named HACKING etc.

## Reading the directory structure

Looking at in what policy the directories are divided. Grasping the overview such as how the program is structured, and what the parts are.

## Reading the file structure

While browsing (the names of) the functions, also looking at the policy of how the files are divided. You should pay attention to the file names because they are like comments whose lifetime is very long.

Additionally, if a file contains some modules in it, for each module the functions to compose it should be grouped together, so you can find out the module structure from the order of the functions.

## Investigating abbreviations

As you encounter ambiguous abbreviations, make a list of them and investigate each of them as early as possible. For example, when it is written “GC”, things will be very different depending on whether it means “Garbage Collection” or “Graphic Context”.

Abbreviations for a program are generally made by the methods like taking the initial letters or dropping the vowels. Especially, popular abbreviations in the fields of the target program are used unconditionally, thus you should be familiar with them at an early stage.

## **Understanding data structure**

If you find both data and code, you should first investigate the data structure. In other words, when exploring code in C, it's better to start with header files. And in this case, let's make the most of our imagination from their filenames. For example, if you find `frame.h`, it would probably be the stack frame definition.

Also, you can understand many things from the member names of a struct and their types. For example, if you find the member `next`, which points to its own type, then it will be a linked list. Similarly, when you find members such as `parent`, `children`, and `sibling`, then it must be a tree structure. When `prev`, it will be a stack.

## **Understanding the calling relationship between functions**

After names, the next most important thing to understand is the relationships between functions. A tool to visualize the calling relationships is especially called a “call graph”, and this is very useful. For this, we'd like to utilize tools.

A text-based tool is sufficient, but it's even better if a tool can generate diagrams. However such tool is seldom available (especially few tools are for free). When I analyzed `ruby` to write this book, I wrote a small command language and a parser in Ruby and generated diagrams half-automatically by passing the results to the tool named `graphviz`.

## Reading functions

Reading how it works to be able to explain things done by the function concisely. It's good to read it part by part as looking at the figure of the function relationships.

What is important when reading functions is not “what to read” but “what not to read”. The ease of reading is decided by how much we can cut out the codes. What should exactly be cut out? It is hard to understand without seeing the actual example, thus it will be explained in the main part.

Additionally, when you don't like its coding style, you can convert it by using the tool like `indent`.

## Experimenting by modifying it as you like

It's a mystery of human body, when something is done using a lot of parts of your body, it can easily persist in your memory. I think the reason why not a few people prefer using manuscript papers to a keyboard is not only they are just nostalgic but such fact is also

related.

Therefore, because merely reading on a monitor is very ineffective to remember with our bodies, rewrite it while reading. This way often helps our bodies get used to the code relatively soon. If there are names or code you don't like, rewrite them. If there's a cryptic abbreviation, substitute it so that it would be no longer abbreviated.

However, it goes without saying but you should also keep the original source aside and check the original one when you think it does not make sense along the way. Otherwise, you would be wondering for hours because of a simple your own mistake. And since the purpose of rewriting is getting used to and not rewriting itself, please be careful not to be enthusiastic very much.

## ■ **Reading the history**

A program often comes with a document which is about the history of changes. For example, if it is a software of GNU, there's always a file named `ChangeLog`. This is the best resource to know about “the reason why the program is as it is”.

Alternatively, when a version control system like CVS or SCCS is used and you can access it, its utility value is higher than `ChangeLog`. Taking CVS as an example, `cvs annotate`, which displays the place which modified a particular line, and `cvs diff`, which takes difference from the specified version, and so on are convenient.

Moreover, in the case when there's a mailing list or a news group for developers, you should get the archives so that you can search over them any time because often there's the information about the exact reason of a certain change. Of course, if you can search online, it's also sufficient.

## The tools for static analysis

Since various tools are available for various purposes, I can't describe them as a whole. But if I have to choose only one of them, I'd recommend `global`. The most attractive point is that its structure allows us to easily use it for the other purposes. For instance, `gctags`, which comes with it, is actually a tool to create tag files, but you can use it to create a list of the function names contained in a file.

```
~/src/ruby % gctags class.c | awk '{print $1}'  
SPECIAL_SINGLETON  
SPECIAL_SINGLETON  
clone_method  
include_class_new  
ins_methods_i  
ins_methods_priv_i  
ins_methods_prot_i  
method_list  
:  
:
```

That said, but this is just a recommendation of this author, you as a reader can use whichever tool you like. But in that case, you should choose a tool equipped with at least the following features.

- list up the function names contained in a file
- find the location from a function name or a variable name (It's more preferable if you can jump to the location)
- function cross-reference

## Build

---

### ■ Target version

The version of `ruby` described in this book is 1.7 (2002-09-12). Regarding `ruby`, it is a stable version if its minor version is an even number, and it is a developing version if it is an odd number. Hence, 1.7 is a developing version. Moreover, 9/12 does not indicate any particular period, thus this version is not distributed as an official package. Therefore, in order to get this version, you can get from the CD-ROM attached to this book or the support site \footnote{The support site of this book.....<http://i.loveruby.net/ja/rhg/>} or you need to use the CVS which will be described later.

There are some reasons why it is not 1.6, which is the stable version, but 1.7. One thing is that, because both the specification and the implementation are organized, 1.7 is easier to deal with. Secondly, it's easier to use CVS if it is the edge of the developing version. Additionally, it is likely that 1.8, which is the next stable version, will be out in the near future. And the last one is,

investigating the edge would make our mood more pleasant.

## Getting the source code

The archive of the target version is included in the attached CD-ROM. In the top directory of the CD-ROM,

```
ruby-rhg.tar.gz  
ruby-rhg.zip  
ruby-rhg.lzh
```

these three versions are placed, so I'd like you to use whichever one that is convenient for you. Of course, whichever one you choose, the content is the same. For example, the archive of `tar.gz` can be extracted as follows.

```
~/src % mount /mnt/cdrom  
~/src % gzip -dc /mnt/cdrom/ruby-rhg.tar.gz | tar xf -  
~/src % umount /mnt/cdrom
```

## Compiling

Just by looking at the source code, you can “read” it. But in order to know about the program, you need to actually use it, remodel it and experiment with it. When experimenting, there’s no meaning if you didn’t use the same version you are looking at, thus naturally you’d need to compile it by yourself.

Therefore, from now on, I’ll explain how to compile. First, let’s start with the case of Unix-like OS. There’s several things to

consider on Windows, so it will be described in the next section altogether. However, Cygwin is on Windows but almost Unix, thus I'd like you to read this section for it.

## Building on a Unix-like OS

When it is a Unix-like OS, because generally it is equipped with a C compiler, by following the below procedures, it can pass in most cases. Let us suppose `~/src/ruby` is the place where the source code is extracted.

```
~/src/ruby % ./configure  
~/src/ruby % make  
~/src/ruby % su  
~/src/ruby # make install
```

Below, I'll describe several points to be careful about.

On some platforms like Cygwin, UX/4800, you need to specify the `--enable-shared` option at the phase of `configure`, or you'd fail to link. `--enable-shared` is an option to put the most of `ruby` out of the command as shared libraries (`libruby.so`).

```
~/src/ruby % ./configure --enable-shared
```

The detailed tutorial about building is included in `doc/build.html` of the attached CD-ROM, I'd like you to try as reading it.

## Building on Windows

If the thing is to build on windows, it becomes way complicated. The source of the problem is, there are multiple building environments.

- Visual C++
- MinGW
- Cygwin
- Borland C++ Compiler

First, the condition of the Cygwin environment is closer to UNIX than Windows, you can follow the building procedures for Unix-like OS.

If you'd like to compile with Visual C++, Visual C++ 5.0 and later is required. There's probably no problem if it is version 6 or .NET.

MinGW or Minimalist GNU for Windows, it is what the GNU compiling environment (Namely, `gcc` and `binutils`) is ported on Windows. Cygwin ports the whole UNIX environment. On the contrary, MinGW ports only the tools to compile. Moreover, a program compiled with MinGW does not require any special DLL at runtime. It means, the `ruby` compiled with MinGW can be treated completely the same as the Visual C++ version.

Alternatively, if it is personal use, you can download the version 5.5 of Borland C++ Compiler for free from the site of Boarland.

\footnote{The Borland site: <http://www.borland.co.jp>} Because `ruby` started to support this environment fairly recently, there's more or less anxiety, but there was not any particular problem on the build

test done before the publication of this book.

Then, among the above four environments, which one should we choose? First, basically the Visual C++ version is the most unlikely to cause a problem, thus I recommend it. If you have experienced with UNIX, installing the whole Cygwin and using it is good. If you have not experienced with UNIX and you don't have Visual C++, using MinGW is probably good.

Below, I'll explain how to build with Visual C++ and MinGW, but only about the outlines. For more detailed explanations and how to build with Borland C++ Compiler, they are included in doc/build.html of the attached CD-ROM, thus I'd like you to check it when it is necessary.

## Visual C++

It is said Visual C++, but usually IDE is not used, we'll build from DOS prompt. In this case, first we need to initialize environment variables to be able to run Visual C++ itself. Since a batch file for this purpose came with Visual C++, let's execute it first.

```
C:\> cd "\Program Files\Microsoft Visual Studio .NET\Vc7\bin"  
C:\Program Files\Microsoft Visual Studio .NET\Vc7\bin> vcvars32
```

This is the case of Visual C++ .NET. If it is version 6, it can be found in the following place.

```
C:\Program Files\Microsoft Visual Studio\VC98\bin\
```

After executing `vcvars32`, all you have to do is to move to the `win32` folder of the source tree of `ruby` and build. Below, let us suppose the source tree is in `C:\src`.

```
C:\> cd src\ruby
C:\src\ruby> cd win32
C:\src\ruby\win32> configure
C:\src\ruby\win32> nmake
C:\src\ruby\win32> nmake DESTDIR="C:\Program Files\ruby" install
```

Then, `ruby` command would be installed in `C:\Program Files\ruby\bin\`, and Ruby libraries would be in `C:\Program Files\ruby\lib\`. Because `ruby` does not use registries and such at all, you can uninstall it by deleting `C:\Program Files\ruby` and below.

## MinGW

As described before, MinGW is only an environment to compile, thus the general UNIX tools like `sed` or `sh` are not available.

However, because they are necessary to build `ruby`, you need to obtain it from somewhere. For this, there are also two methods: Cygwin and MSYS (Minimal SYStem).

However, I can't recommend MSYS because troubles were continuously happened at the building contest performed before the publication of this book. On the contrary, in the way of using Cygwin, it can pass very straightforwardly. Therefore, in this book, I'll explain the way of using Cygwin.

First, install MinGW and the entire developing tools by using setup.exe of Cygwin. Both Cygwin and MinGW are also included in the attached CD-ROM. \footnote{Cygwin and MinGW.....See also doc/win.html of the attached CD-ROM} After that, all you have to do is to type as follows from bash prompt of Cygwin.

```
~/src/ruby % ./configure --with-gcc='gcc -mno-cygwin' \
                      --enable-shared i386-mingw32
~/src/ruby % make
~/src/ruby % make install
```

That's it. Here the line of `configure` spans multi-lines but in practice we'd write it on one line and the backslash is not necessary. The place to install is `\usr\local\` and below of the drive on which it is compiled. Because really complicated things occur around here, the explanation would be fairly long, so I'll explain it comprehensively in `doc/build.html` of the attached CD-ROM.

## Building Details

---

Until here, it has been the `README`-like description. This time, let's look at exactly what is done by what we have been done. However, the talks here partially require very high-level knowledge. If you can't understand, I'd like you to skip this and directly jump to the next section. This should be written so that you can understand by coming back after reading the entire book.

Now, on whichever platform, building ruby is separated into three phases. Namely, `configure`, `make` and `make install`. As considering the explanation about `make install` unnecessary, I'll explain the `configure` phase and the `make` phase.

## configure

First, `configure`. Its content is a shell script, and we detect the system parameters by using it. For example, “whether there’s the header file `setjmp.h`” or “whether `alloca()` is available”, these things are checked. The way to check is unexpectedly simple.

Target to check	Method
commands	execute it actually and then check <code>\$?</code>
header files	<code>if [ -f \$includedir/stdio.h ]</code>
functions	compile a small program and check whether linking is success

When some differences are detected, somehow it should be reported to us. The way to report is, the first way is `Makefile`. If we put a `Makefile.in` in which parameters are embedded in the form of `@param@`, it would generate a `Makefile` in which they are substituted with the actual values. For example, as follows,

```
Makefile.in:  CFLAGS = @CFLAGS@  
              ↓  
Makefile    :  CFLAGS = -g -O2
```

Alternatively, it writes out the information about, for instance,

whether there are certain functions or particular header files, into a header file. Because the output file name can be changed, it is different depending on each program, but it is `config.h` in ruby. I'd like you to confirm this file is created after executing `configure`. Its content is something like this.

### ▼ `config.h`

```
:
:
#define HAVE_SYS_STAT_H 1
#define HAVE_STDLIB_H 1
#define HAVE_STRING_H 1
#define HAVE_MEMORY_H 1
#define HAVE_STRINGS_H 1
#define HAVE_INTTYPES_H 1
#define HAVE_STDINT_H 1
#define HAVE_UNISTD_H 1
#define _FILE_OFFSET_BITS 64
#define HAVE_LONG_LONG 1
#define HAVE_OFF_T 1
#define SIZEOF_INT 4
#define SIZEOF_SHORT 2
:
:
```

Each meaning is easy to understand. `HAVE_xxxx_H` probably indicates whether a certain header file exists, `SIZEOF_SHORT` must indicate the size of the `short` type of C. Likewise, `SIZEOF_INT` indicates the byte length of `int`, `HAVE_OFF_T` indicates whether the `offset_t` type is defined or not.

As we can understand from the above things, `configure` does detect the differences but it does not automatically absorb the differences.

Bridging the difference is left to each programmer. For example, as follows,

## ▼ A typical usage of the HAVE\_ macro

```
24 #ifdef HAVE_STDLIB_H  
25 # include <stdlib.h>  
26 #endif
```

(ruby.h)

## ■ autoconf

configure is not a ruby-specific tool. Whether there are functions, there are header files, ... it is obvious that these tests have regularity. It is wasteful if each person who writes a program wrote each own distinct tool.

Here a tool named autoconf comes in. In the files named `configure.in` or `configure.ac`, write about “I’d like to do these checks”, process it with `autoconf`, then an adequate `configure` would be generated. The `.in` of `configure.in` is probably an abbreviation of input. It’s the same as the relationship between `Makefile` and `Makefile.in`. `.ac` is, of course, an abbreviation of AutoConf.

To illustrate this talk up until here, it would be like Figure 1.

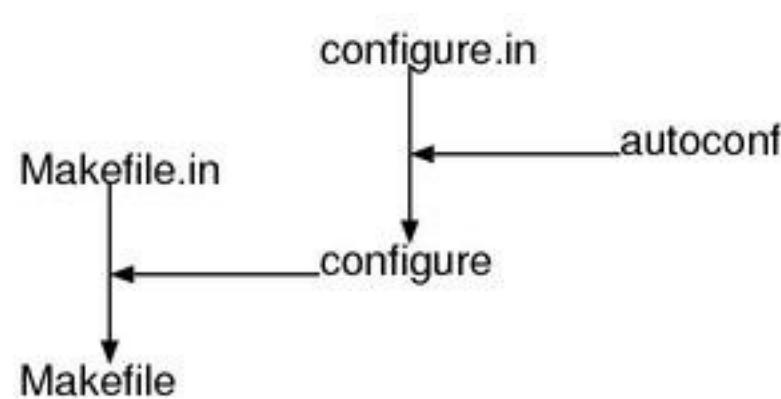


Figure 1: The process until `Makefile` is created

For the readers who want to know more details, I recommend “GNU Autoconf/Automake/Libtool” Gary V.Vaughan, Ben Elliston, Tom Tromey, Ian Lance Taylor.

By the way, ruby’s `configure` is, as said before, generated by using `autoconf`, but not all the `configure` in this world are generated with `autoconf`. It can be written by hand or another tool to automatically generate can be used. Anyway, it’s sufficient if ultimately there are `Makefile` and `config.h` and many others.

## make

At the second phase, `make`, what is done? Of course, it would compile the source code of ruby, but when looking at the output of `make`, I feel like there are many other things it does. I’ll briefly explain the process of it.

1. compile the source code composing ruby itself
2. create the static library `libruby.a` gathering the crucial parts of ruby
3. create “`miniruby`”, which is an always statically-linked ruby

4. create the shared library `libruby.so` when `--enable-shared`
5. compile the extension libraries (under `ext/`) by using `miniruby`
6. At last, generate the real `ruby`

There are two reasons why it creates `miniruby` and `ruby` separately. The first one is that compiling the extension libraries requires `ruby`. In the case when `--enable-shared`, `ruby` itself is dynamically linked, thus there's a possibility not be able to run instantly because of the load paths of the libraries. Therefore, create `miniruby`, which is statically linked, and use it during the building process.

The second reason is, in a platform where we cannot use shared libraries, there's a case when the extension libraries are statically linked to `ruby` itself. In this case, it cannot create `ruby` before compiling all extension libraries, but the extension libraries cannot be compiled without `ruby`. In order to resolve this dilemma, it uses `miniruby`.

## CVS

---

The `ruby` archive included in the attached CD-ROM is, as the same as the official release package, just a snapshot which is an appearance at just a particular moment of `ruby`, which is a continuously changing program. How `ruby` has been changed, why it has been so, these things are not described there. Then what is

the way to see the entire picture including the past. We can do it by using CVS.

## About CVS

CVS is shortly an undo list of editors. If the source code is under the management of CVS, the past appearance can be restored anytime, and we can understand who and where and when and how changed it immediately any time. Generally a program doing such job is called source code management system and CVS is the most famous open-source source code management system in this world.

Since ruby is also managed with CVS, I'll explain a little about the mechanism and usage of CVS. First, the most important idea of CVS is repository and working-copy. I said CVS is something like an undo list of editor, in order to archive this, the records of every changing history should be saved somewhere. The place to store all of them is “CVS repository”.

Directly speaking, repository is what gathers all the past source codes. Of course, this is only a concept, in reality, in order to save spaces, it is stored in the form of one recent appearance and the changing differences (namely, batches). In any ways, it is sufficient if we can obtain the appearance of a particular file of a particular moment any time.

On the other hand, “working copy” is the result of taking files from the repository by choosing a certain point. There's only one

repository, but you can have multiple working copies. (Figure 2)

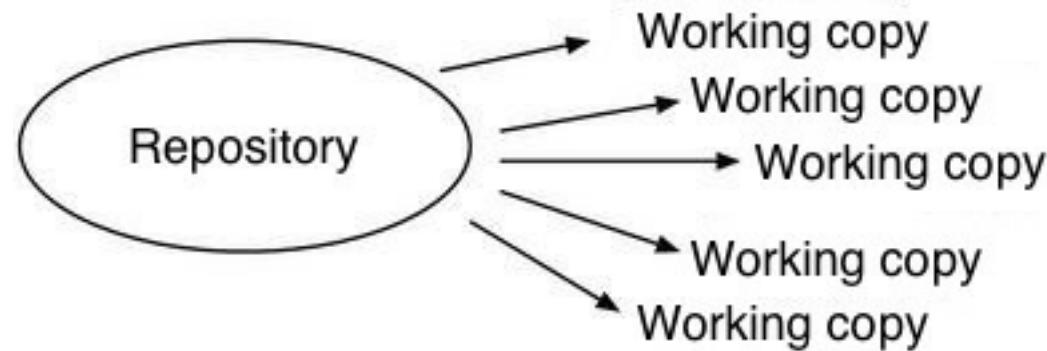


Figure 2: Repository and working copies

When you'd like to modify the source code, first take a working copy, edit it by using editor and such, and "return" it. Then, the change is recorded to the repository. Taking a working copy from the repository is called "checkout", returning is called "checkin" or "commit" (Figure 3). By checking in, the change is recorded to the repository, then we can obtain it any time.

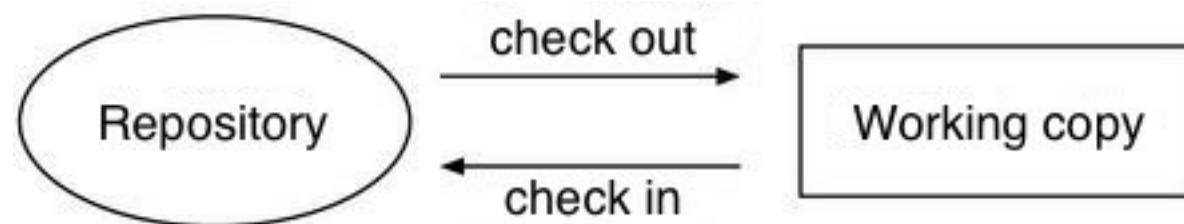


Figure 3: Checkin and Checkout

The biggest trait of CVS is we can access it over the networks. It means, if there's only one server which holds the repository, everyone can checkin/checkout over the internet any time. But generally the access to check in is restricted and we can't do it freely.

# Revision

How can we do to obtain a certain version from the repository? One way is to specify with time. By requiring “give me the edge version of that time”, it would select it. But in practice, we rarely specify with time. Most commonly, we use something named “revision”.

“Revision” and “Version” have the almost same meaning. But usually “version” is attached to the project itself, thus using the word “version” can be confusing. Therefore, the word “revision” is used to indicate a bit smaller unit.

In CVS, the file just stored in the repository is revision 1.1. Checking out it, modifying it, checking in it, then it would be revision 1.2. Next it would be 1.3 then 1.4.

## A simple usage example of CVS

Keeping in mind the above things, I’ll talk about the usage of CVS very very briefly. First, `cvs` command is essential, so I’d like you to install it beforehand. The source code of `cvs` is included in the attached CD-ROM \footnote{`cvs` : archives/cvs-1.11.2.tar.gz}. How to install `cvs` is really far from the main line, thus it won’t be explained here.

After installing it, let’s checkout the source code of `ruby` as an experiment. Type the following commands when you are online.

```
% cvs -d :pserver:anonymous@cvs.ruby-lang.org:/src login
CVS Password: anonymous
% cvs -d :pserver:anonymous@cvs.ruby-lang.org:/src checkout ruby
```

Any options were not specified, thus the edge version would be automatically checked out. The truly edge version of ruby must appear under ruby/.

Additionally, if you'd like to obtain the version of a certain day, you can use -D option of cvs checkout. By typing as follows, you can obtain a working copy of the version which is being explained by this book.

```
% cvs -d :pserver:anonymous@cvs.ruby-lang.org:/src checkout -D2002
```

At this moment, you have to write options immediately after checkout. If you wrote "ruby" first, it would cause a strange error complaining "missing a module".

And, with the anonymous access like this example, we cannot check in. In order to practice checking in, it's good to create a (local) repository and store a "Hello, World!" program in it. The concrete way to store is not explained here. The manual coming with cvs is fairly friendly. Regarding books which you can read in Japanese, I recommend translated "Open Source Development with CVS" Karl Fogel, Moshe Bar.

# The composition of ruby

## The physical structure

Now it is time to start to read the source code, but what is the thing we should do first? It is looking over the directory structure. In most cases, the directory structure, meaning the source tree, directly indicate the module structure of the program. Abruptly searching `main()` by using `grep` and reading from the top in its processing order is not smart. Of course finding out `main()` is also important, but first let's take time to do `ls` or `head` to grasp the whole picture.

Below is the appearance of the top directory immediately after checking out from the CVS repository. What end with a slash are subdirectories.

COPYING	compar.c	gc.c	numeric.c	samp
COPYING.ja	config.guess	hash.c	object.c	sign
CVS/	config.sub	inits.c	pack.c	spri
ChangeLog	configure.in	install-sh	parse.y	st.c
GPL	cygwin/	instruby.rb	prec.c	st.t
LEGAL	defines.h	intern.h	process.c	stri
LGPL	dir.c	io.c	random.c	stru
MANIFEST	djgpp/	keywords	range.c	time
Makefile.in	dln.c	lex.c	re.c	util
README	dln.h	lib/	re.h	util
README.EXT	dmyext.c	main.c	regex.c	vari
README.EXT.ja	doc/	marshal.c	regex.h	vers
README.ja	enum.c	math.c	ruby.1	vers
ToDo	env.h	misc/	ruby.c	vms/
array.c	error.c	missing/	ruby.h	win3
bcc32/	eval.c	missing.h	rubyio.h	x68/

bignum.c  
class.c

ext/  
file.c

mkconfig.rb  
node.h

rubysig.h  
rubystest.rb

Recently the size of a program itself has become larger, and there are many softwares whose subdirectories are divided into pieces, but ruby has been consistently used the top directory for a long time. It becomes problematic if there are too many files, but we can get used to this amount.

The files at the top level can be categorized into six:

- documents
- the source code of ruby itself
- the tool to build ruby
- standard extension libraries
- standard Ruby libraries
- the others

The source code and the build tool are obviously important. Aside from them, I'll list up what seems useful for us.

- ChangeLog

The records of changes on ruby. This is very important when investigating the reason of a certain change.

- README.EXT README.EXT.ja

How to create an extension library is described, but in the course of it, things relating to the implementation of ruby itself are also

written.

# Dissecting Source Code

From now on, I'll further split the source code of `ruby` itself into more tiny pieces. As for the main files, its categorization is described in `README.EXT`, thus I'll follow it. Regarding what is not described, I categorized it by myself.

## Ruby Language Core

<code>class.c</code>	class relating API
<code>error.c</code>	exception relating API
<code>eval.c</code>	evaluator
<code>gc.c</code>	garbage collector
<code>lex.c</code>	reserved word table
<code>object.c</code>	object system
<code>parse.y</code>	parser
<code>variable.c</code>	constants, global variables, class variables
<code>ruby.h</code>	The main macros and prototypes of <code>ruby</code>
<code>intern.h</code>	the prototypes of C API of <code>ruby</code> . <code>intern</code> seems to be an abbreviation of internal, but the functions written here can be used from extension libraries.
<code>rubysig.h</code>	the header file containing the macros relating to signals
<code>node.h</code>	the definitions relating to the syntax tree nodes
<code>env.h</code>	the definitions of the structs to express the context of the evaluator

The parts to compose the core of the `ruby` interpreter. The most of the files which will be explained in this book are contained here. If

you consider the number of the files of the entire `ruby`, it is really only a few. But if you think based on the byte size, 50% of the entire amount is occupied by these files. Especially, `eval.c` is 200KB, `parse.y` is 100KB, these files are large.

## Utility

`dln.c` dynamic loader

`regex.c` regular expression engine

`st.c` hash table

`util.c` libraries for radix conversions and sort and so on

It means utility for `ruby`. However, some of them are so large that you cannot imagine it from the word “utility”. For instance, `regex.c` is 120 KB.

## Implementation of ruby command

`dmyext.c` dummy of the routine to initialize extension libraries ( DumMY EXTension )

`inits.c` the entry point for core and the routine to initialize extension libraries

`main.c` the entry point of `ruby` command (this is unnecessary for `libruby` )

`ruby.c` the main part of `ruby` command (this is also necessary for `libruby` )

`version.c` the version of `ruby`

The implementation of `ruby` command, which is of when typing `ruby` on the command line and execute it. This is the part, for instance, to interpret the command line options. Aside from `ruby`

command, as the commands utilizing ruby core, there are `mod_ruby` and `vim`. These commands are functioning by linking to the `libruby` library (`.a/.so/.dll` and so on).

## Class Libraries

<code>array.c</code>	class <code>Array</code>
<code>bignum.c</code>	class <code>Bignum</code>
<code>compar.c</code>	module <code>Comparable</code>
<code>dir.c</code>	class <code>Dir</code>
<code>enum.c</code>	module <code>Enumerable</code>
<code>file.c</code>	class <code>File</code>
<code>hash.c</code>	class <code>Hash</code> (Its actual body is <code>st.c</code> )
<code>io.c</code>	class <code>IO</code>
<code>marshal.c</code>	module <code>Marshal</code>
<code>math.c</code>	module <code>Math</code>
<code>numeric.c</code>	class <code>Numeric, Integer, Fixnum, Float</code>
<code>pack.c</code>	<code>Array#pack, String#unpack</code>
<code>prec.c</code>	module <code>Precision</code>
<code>process.c</code>	module <code>Process</code>
<code>random.c</code>	<code>Kernel#srand(), rand()</code>
<code>range.c</code>	class <code>Range</code>
<code>re.c</code>	class <code>Regexp</code> (Its actual body is <code>regex.c</code> )
<code>signal.c</code>	module <code>Signal</code>
<code>sprintf.c</code>	<b>ruby-specific</b> <code>sprintf()</code>
<code>string.c</code>	class <code>String</code>
<code>struct.c</code>	class <code>Struct</code>
<code>time.c</code>	class <code>Time</code>

The implementations of the Ruby class libraries. What listed here are basically implemented in the completely same way as the ordinary Ruby extension libraries. It means that these libraries are

also examples of how to write an extension library.

## Files depending on a particular platform

bcc32/ Borland C++ (Win32)

beos/ BeOS

cygwin/ Cygwin (the UNIX simulation layer on Win32)

djgpp/ djgpp (the free developing environment for DOS)

vms/ VMS (an OS had been released by DEC before)

win32/ Visual C++ (Win32)

x86/ Sharp X68000 series (OS is Human68k)

Each platform-specific code is stored.

## fallback functions

missing/

Files to offset the functions which are missing on each platform.

Mainly functions of `libc`.

## Logical Structure

Now, there are the above four groups and the core can be divided further into three: First, “object space” which creates the object world of Ruby. Second, “parser” which converts Ruby programs (in text) to the internal format. Third, “evaluator” which drives Ruby programs. Both parser and evaluator are composed above object space, parser converts a program into the internal format, and evaluator actuates the program. Let me explain them in order.

# Object Space

The first one is object space. This is very easy to understand. It is because all of what dealt with by this are basically on the memory, thus we can directly show or manipulate them by using functions. Therefore, in this book, the explanation will start with this part. Part 1 is from chapter 2 to chapter 7.

# Parser

The second one is parser. Probably some preliminary explanations are necessary for this.

`ruby` command is the interpreter of Ruby language. It means that it analyzes the input which is a text on invocation and executes it by following it. Therefore, `ruby` needs to be able to interpret the meaning of the program written as a text, but unfortunately text is very hard to understand for computers. For computers, text files are merely byte sequences and nothing more than that. In order to comprehend the meaning of text from it, some special gimmick is necessary. And the gimmick is parser. By passing through parser, (a text as) a Ruby program would be converted into the `ruby`-specific internal expression which can be easily handled from the program.

The internal expression is called “syntax tree”. Syntax tree expresses a program by a tree structure, for instance, figure 4 shows how an `if` statement is expressed.

```
if i < 10 then
  var = i + 3
end
```

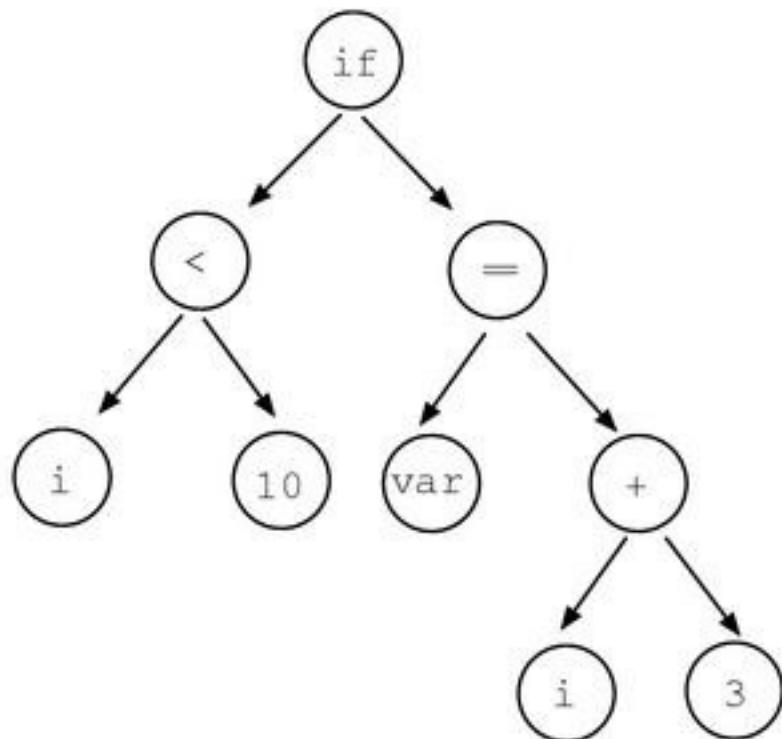


Figure 4: an `if` statement and its corresponding syntax tree

Parser will be described in Part 2 “Syntactic Analysis”. Part 2 is from chapter 10 to chapter 12. Its target file is only `parse.y`.

## Evaluator

Objects are easy to understand because they are tangible. Also regarding parser, What it does is ultimately converting a data format into another one, so it's reasonably easy to understand. However, the third one, evaluator, this is completely elusive.

What evaluator does is “executing” a program by following a syntax tree. This sounds easy, but what is “executing”? To answer this question precisely is fairly difficult. What is “executing an `if` statement”? What is “executing a `while` statement”? What does “assigning to a local variable” mean? We cannot understand evaluator without answering all of such questions clearly and

precisely.

In this book, evaluator will be discussed in Part 3 “Evaluate”. Its target file is `eval.c`. `eval` is an abbreviation of “evaluator”.

Now, I’ve described briefly about the structure of `ruby`, however even though the ideas were explained, it does not so much help us understand the behavior of program. In the next chapter, we’ll start with actually using `ruby`.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](#)

# Ruby Hacking Guide

Translated by Sebastian Krause

# Chapter 1: Introduction

## A Minimal Introduction to Ruby

---

Here the Ruby prerequisites are explained, which one needs to know in order to understand the first section. I won't point out programming techniques or points one should be careful about. So don't think you'll be able to write Ruby programs just because you read this chapter. Readers who have prior experience with Ruby can skip this chapter.

We will talk about grammar extensively in the second section, hence I won't delve into the finer points of grammar here. From hash literals and such I'll show only the most widely used notations. On principle I won't omit things even if I can. This way the syntax becomes more simple. I won't always say "We can omit this".

## Objects

---

### ■ Strings

Everything that can be manipulated in a Ruby program is an object. There are no primitives as Java's `int` and `long`. For instance if we write as below it denotes a string object with content `content`.

```
"content"
```

I casually called it a string object but to be precise this is an expression which generates a string object. Therefore if we write it several times each time another string object is generated.

```
"content"  
"content"  
"content"
```

Here three string objects with content `content` are generated.

By the way, objects just existing there can't be seen by programmers. Let's show how to print them on the terminal.

```
p("content")  # Shows "content"
```

Everything after an `#` is a comment. From now on, I'll put the result of an expression in a comment behind.

`p(.....)` calls the function `p`. It displays arbitrary objects "as such". It's basically a debugging function.

Precisely speaking, there are no functions in Ruby, but just for now we can think of it as a function. You can use functions wherever you are.

# Various Literals

Now, let's explain some more the expressions which directly generate objects, the so-called literals. First the integers and floating point numbers.

```
# Integer
1
2
100
99999999999999999999999999999999 # Arbitrarily big integers

# Float
1.0
99.999
1.3e4      # 1.3×10^4
```

Don't forget that these are all expressions which generate objects. I'm repeating myself but there are no primitives in Ruby.

Below an array object is generated.

```
[1, 2, 3]
```

This program generates an array which consists of the three integers 1, 2 and 3 in that order. As the elements of an array can be arbitrary objects the following is also possible.

```
[1, "string", 2, ["nested", "array"]]
```

And finally, a hash table is generated by the expression below.

```
{"key"=>"value", "key2"=>"value2", "key3"=>"value3"}
```

A hash table is a structure which expresses one-to-one relationships between arbitrary objects. The above line creates a table which stores the following relationships.

```
"key"    →  "value"  
"key2"   →  "value2"  
"key3"   →  "value3"
```

If we ask a hash table created in this way “What’s corresponding to key?”, it’ll answer “That’s value.” How can we ask? We use methods.

## Method Calls

We can call methods on an object. In C++ Jargon they are member functions. I don’t think it’s necessary to explain what a method is. I’ll just explain the notation.

```
"content".upcase()
```

Here the `upcase` method is called on a string object ( with content `content`). As `upcase` is a method which returns a new string with the small letters replaced by capital letters, we get the following result.

```
p("content".upcase())  # Shows "CONTENT"
```

Method calls can be chained.

```
"content".upcase().downcase()
```

Here the method `downcase` is called on the return value of `"content".upcase()`.

There are no public fields (member variables) as in Java or C++. The object interface consists of methods only.

## The Program

---

### Top Level

In Ruby we can just write expressions and it becomes a program. One doesn't need to define a `main()` as in C++ or Java.

```
p("content")
```

This is a complete Ruby program. If we put this into a file called `first.rb` we can execute it from the command line as follows.

```
% ruby first.rb
"content"
```

With the `-e` option of the `ruby` program we don't even need to create a file.

```
% ruby -e 'p("content")'
"content"
```

By the way, the place where `p` is written is the lowest nesting level of the program, it means the highest level from the program's standpoint, thus it's called "top-level". Having top-level is a characteristic trait of Ruby as a scripting language.

In Ruby, one line is usually one statement. A semicolon at the end isn't necessary. Therefore the program below is interpreted as three statements.

```
p("content")
p("content".upcase())
p("CONTENT".downcase())
```

When we execute it it looks like this.

```
% ruby second.rb
"content"
"CONTENT"
"content"
```

## Local Variables

In Ruby all variables and constants store references to objects. That's why one can't copy the content by assigning one variable to another variable. Variables of type `Object` in Java or pointers to objects in C++ are good to think of. However, you can't change the value of each pointer itself.

In Ruby one can tell the classification (scope) of a variable by the beginning of the name. Local variables start with a small letter or

an underscore. One can write assignments by using “=”.

```
str = "content"  
arr = [1,2,3]
```

An initial assignment serves as declaration, an explicit declaration is not necessary. Because variables don’t have types, we can assign any kind of objects indiscriminately. The program below is completely legal.

```
lvar = "content"  
lvar = [1,2,3]  
lvar = 1
```

But even if we can, we don’t have to do it. If different kind of objects are put in one variable, it tends to become difficult to read. In a real world Ruby program one doesn’t do this kind of things without a good reason. The above was just an example for the sake of it.

Variable reference has also a pretty sensible notation.

```
str = "content"  
p(str)           # Shows "content"
```

In addition let’s check the point that a variable hold a reference by taking an example.

```
a = "content"  
b = a  
c = b
```

After we execute this program all three local variables `a` `b` `c` point to the same object, a string object with content "content" created on the first line (Figure 1).

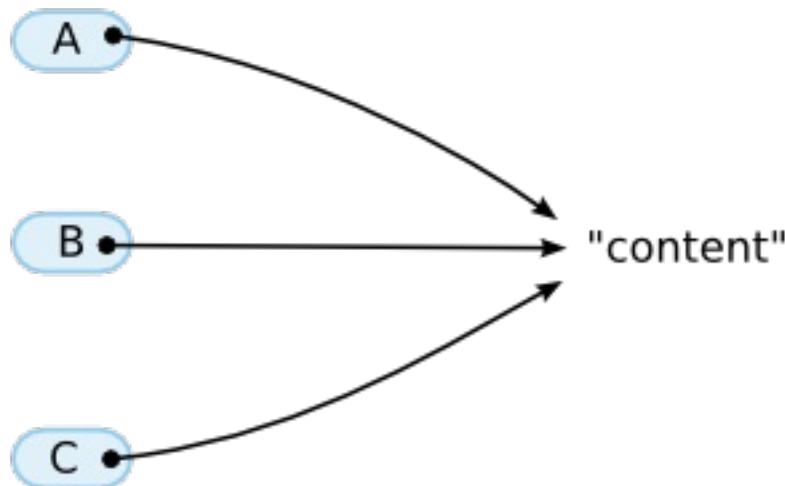


Figure 1: Ruby variables store references to objects

By the way, as these variables are called local, they should be local to somewhere, but we cannot talk about this scope without reading a bit further. Let's say for now that the top level is one local scope.

## Constants

Constants start with a capital letter. They can only be assigned once (at their creation).

```
Const = "content"  
PI = 3.1415926535
```

```
p(Const) # Shows "content"
```

I'd like to say that if we assign twice an error occurs. But there is just a warning, not an error. It is in this way in order to avoid raising an error even when the same file is loaded twice in applications that manipulate Ruby program itself, for instance in development environments. Therefore, it is allowed due to practical requirements and there's no other choice, but essentially there should be an error. In fact, up until version 1.1 there really was an error.

```
C = 1
```

```
C = 2 # There is a warning but ideally there should be an error
```

A lot of people are fooled by the word constant. A constant only does not switch objects once it is assigned. But it does not mean the pointed object itself won't change. The term "read only" might capture the concept better than "constant".

By the way, to indicate that an object itself shouldn't be changed another means is used: `freeze`.

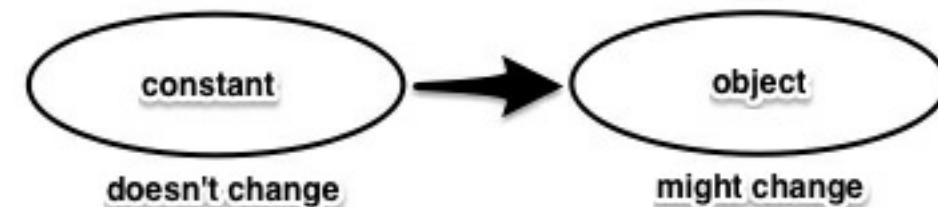


Figure 2: constant means read only

And the scope of constants is actually also cannot be described yet. It will be discussed later in the next section mixing with classes.

# Control Structures

Since Ruby has a wide abundance of control structures, just lining up them can be a huge task. For now, I just mention that there are `if` and `while`.

```
if i < 10 then
  # body
end
```

```
while i < 10 do
  # body
end
```

In a conditional expression, only the two objects, `false` and `nil`, are false and all other various objects are true. `0` or the empty string are also true of course.

It wouldn't be wise if there were just `false`, there is also `true`. And it is of course true.

# Classes and Methods

---

## Classes

In object oriented system, essentially methods belong to objects. It can hold only in a ideal world, though. In a normal program there are a lot of objects which have the same set of methods, it would be

an enormous work if each object remember the set of callable methods. Usually a mechanism like classes or multimethods is used to get rid of the duplication of definitions.

In Ruby, as the traditional way to bind objects and methods together, the concept of classes is used. Namely every object belongs to a class, the methods which can be called are determined by the class. And in this way, an object is called “an instance of the XX class”.

For example the string "str" is an instance of the `String` class. And on this `String` class the methods `upcase`, `downcase`, `strip` and many others are defined. So it looks as if each string object can respond to all these methods.

```
# They all belong to the String class,  
# hence the same methods are defined  
    "content".upcase()  
"This is a pen.".upcase()  
    "chapter II".upcase()  
  
    "content".length()  
"This is a pen.".length()  
    "chapter II".length()
```

By the way, what happens if the called method isn't defined? In a static language a compiler error occurs but in Ruby there is a runtime exception. Let's try it out. For this kind of programs the `-e` option is handy.

```
% ruby -e '"str".bad_method()'  
-e:1: undefined method `bad_method' for "str":String (NoMethodError)
```

When the method isn't found there's apparently a `NoMethodError`.

Always saying "the `upcase` method of `String`" and such is cumbersome. Let's introduce a special notation `String#upcase` refers to the method `upcase` defined in the class `String`.

By the way, if we write `String.upcase` it has a completely different meaning in the Ruby world. What could that be? I explain it in the next paragraph.

## Class Definition

Up to now we talked about already defined classes. We can of course also define our own classes. To define classes we use the `class` statement.

```
class C
end
```

This is the definition of a new class `C`. After we defined it we can use it as follows.

```
class C
end
c = C.new()    # create an instance of C and assign it to the var
```

Note that the notation for creating a new instance is not `new C`. The astute reader might think: Hmm, this `C.new()` really looks like a method call. In Ruby the object generating expressions are indeed

just methods.

In Ruby class names and constant names are the same. Then, what is stored in the constant whose name is the same as a class name? In fact, it's the class. In Ruby all things which a program can manipulate are objects. So of course classes are also expressed as objects. Let's call these *class objects*. Every class is an instance of the class `Class`.

In other words a `class` statement creates a new class object and it assigns a constant named with the classname to the class. On the other hand the generation of an instance references this constant and calls a method on this object ( usually `new`). If we look at the example below, it's pretty obvious that the creation of an instance doesn't differ from a normal method call.

```
S = "content"  
class C  
end  
  
S.upcase() # Get the object the constant S points to and call its upcase method  
C.new() # Get the object the constant C points to and call its new method
```

So `new` is not a reserved word in Ruby.

And we can also use `p` for an instance of a class even immediately after its creation.

```
class C  
end  
  
c = C.new()
```

```
p(c)      # #<C:0x2acbd7e4>
```

It won't display as nicely as a string or an integer but it shows its respective class and it's internal ID. This ID is the pointer value which points to the object.

Oh, I completely forgot to mention about the notation of method names: `Object.new` means the class object `Object` and the `new` method called on the class itself. So `Object#new` and `Object.new` are completely different things, we have to separate them strictly.

```
obj = Object.new()    # Object.new
obj.new()            # Object#new
```

In practice a method `Object#new` is almost never defined so the second line will return an error. Please regard this as an example of the notation.

## Method Definition

Even if we can define classes, it is useless if we cannot define methods. Let's define a method for our class `C`.

```
class C
  def myupcase( str )
    return str.upcase()
  end
end
```

To define a method we use the `def` statement. In this example we

defined the method `myupcase`. The name of the only parameter is `str`. As with variables, it's not necessary to write parameter types or the return type. And we can use any number of parameters.

Let's use the defined method. Methods are usually called from the outside by default.

```
c = C.new()
result = c.myupcase("content")
p(result)  # Shows "CONTENT"
```

Of course if you get used to it you don't need to assign every time. The line below gives the same result.

```
p(C.new().myupcase("content"))  # Also shows "CONTENT"
```

## ■ `self`

During the execution of a method the information about who is itself (the instance on which the method was called) is always saved and can be picked up in `self`. Like the `this` in C++ or Java. Let's check this out.

```
class C
  def get_self()
    return self
  end
end

c = C.new()
p(c)          # #<C:0x40274e44>
p(c.get_self()) # #<C:0x40274e44>
```

As we see, the above two expressions return the exact same object. We could confirm that `self` is `c` during the method call on `c`.

Then what is the way to call a method on itself? What first comes to mind is calling via `self`.

```
class C
  def my_p( obj )
    self.real_my_p(obj)    # called a method against oneself
  end

  def real_my_p( obj )
    p(obj)
  end
end

C.new().my_p(1)    # Output 1
```

But always adding the `self` when calling an own method is tedious. Hence, it is designed so that one can omit the called method (the receiver) whenever one calls a method on `self`.

```
class C
  def my_p( obj )
    real_my_p(obj)    # You can call without specifying the receiver
  end

  def real_my_p( obj )
    p(obj)
  end
end

C.new().my_p(1)    # Output 1
```

## Instance Variables

As there are a saying “Objects are data and code”, just being able to define methods alone would be not so useful. Each object must also be able to store data. In other words instance variables. Or in C++ jargon member variables.

In the fashion of Ruby’s variable naming convention, the variable type can be determined by the first a few characters. For instance variables it’s an @.

```
class C
  def set_i(value)
    @i = value
  end

  def get_i()
    return @i
  end
end

c = C.new()
c.set_i("ok")
p(c.get_i())  # Shows "ok"
```

Instance variables differ a bit from the variables seen before: We can reference them without assigning (defining) them. To see what happens we add the following lines to the code above.

```
c = C.new()
p(c.get_i())  # Shows nil
```

Calling `get` without `set` gives `nil`. `nil` is the object which indicates “nothing”. It’s mysterious that there’s really an object but it means nothing, but that’s just the way it is.

We can use `nil` like a literal as well.

```
p(nil)  # Shows nil
```

## ■ initialize

As we saw before, when we call ‘`new`’ on a freshly defined class, we can create an instance. That’s sure, but sometimes we might want to have a peculiar instantiation. In this case we don’t change the `new` method, we define the `initialize` method. When we do this, it gets called within `new`.

```
class C
  def initialize()
    @i = "ok"
  end
  def get_i()
    return @i
  end
end
c = C.new()
p(c.get_i())  # Shows "ok"
```

Strictly speaking this is the specification of the `new` method but not the specification of the language itself.

## ■ Inheritance

Classes can inherit from other classes. For instance `String` inherits from `Object`. In this book, we’ll indicate this relation by a vertical arrow as in Fig.3.



superclass

subclass

Figure 3: Inheritance

In the case of this illustration, the inherited class (`Object`) is called superclass or superior class. The inheriting class (`String`) is called subclass or inferior class. This point differs from C++ jargon, be careful. But it's the same as in Java.

Anyway let's try it out. Let our created class inherit from another class. To inherit from another class ( or designate a superclass) write the following.

```
class C < SuperClassName
end
```

When we leave out the superclass like in the cases before the class `Object` becomes tacitly the superclass.

Now, why should we want to inherit? Of course to hand over methods. Handing over means that the methods which were defined in the superclass also work in the subclass as if they were defined in there once more. Let's check it out.

```
class C
  def hello()
    return "hello"
  end
end
```

```
class Sub < C
end

sub = Sub.new()
p(sub.hello())  # Shows "hello"
```

hello was defined in the class C but we could call it on an instance of the class Sub as well. Of course we don't need to assign variables. The above is the same as the line below.

```
p(Sub.new().hello())
```

By defining a method with the same name, we can overwrite the method. In C++ and Object Pascal (Delphi) it's only possible to overwrite functions explicitly defined with the keyword `virtual` but in Ruby every method can be overwritten unconditionally.

```
class C
  def hello()
    return "Hello"
  end
end

class Sub < C
  def hello()
    return "Hello from Sub"
  end
end

p(Sub.new().hello())  # Shows "Hello from Sub"
p(C.new().hello())  # Shows "Hello"
```

We can inherit over several steps. For instance as in Fig.4 Fixnum inherits every method from Object, Numeric and Integer. When there

are methods with the same name the nearer classes take preference. As type overloading isn't there at all the requisites are extremely straightforward.

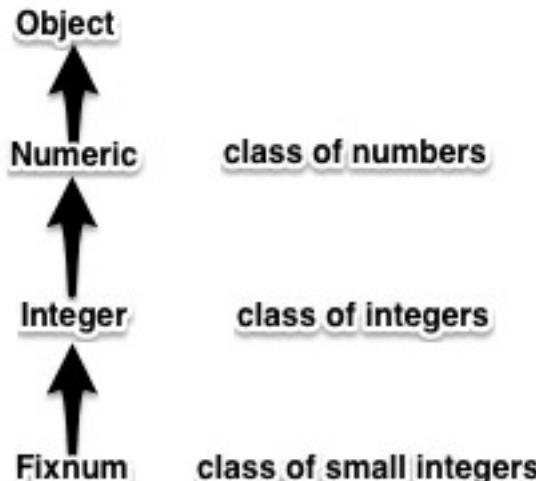


Figure 4: Inheritance over multiple steps

In C++ it's possible to create a class which inherits nothing. While in Ruby one has to inherit from the `Object` class either directly or indirectly. In other words when we draw the inheritance relations it becomes a single tree with `Object` at the top. For example, when we draw a tree of the inheritance relations among the important classes of the basic library, it would look like Fig.5.

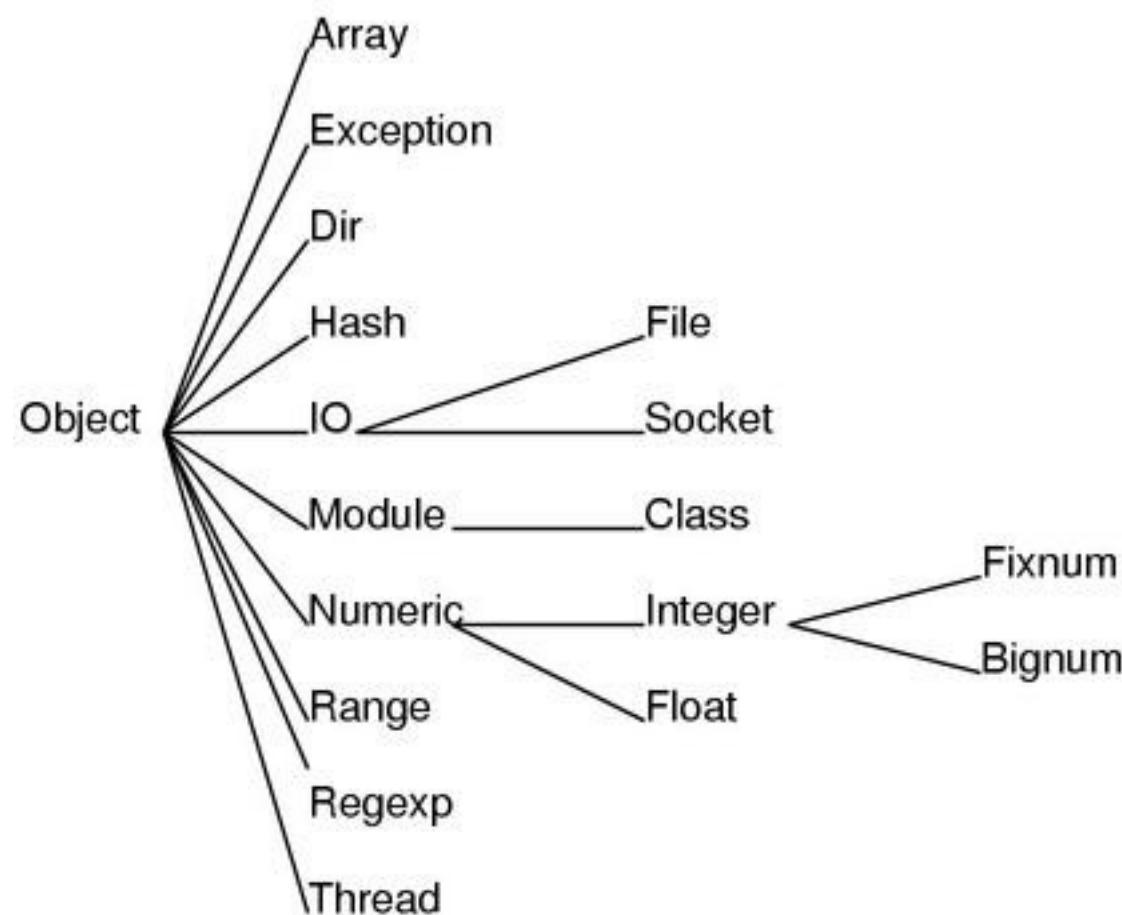


Figure 5: Ruby's class tree

Once the superclass is appointed ( in the definition statement ) it's impossible to change it. In other words, one can add a new class to the class tree but cannot change a position or delete a class.

## ■ Inheritance of Variables.....?

In Ruby (instance) variables aren't inherited. Even though trying to inherit, a class does not know about what variables are going to be used.

But when an inherited method is called ( in an instance of a subclass), assignment of instance variables happens. Which means they become defined. Then, since the namespace of instance variables is completely flat based on each instance, it can be

accessed by a method of whichever class.

```
class A
  def initialize()    # called from when processing new()
    @i = "ok"
  end
end

class B < A
  def print_i()
    p(@i)
  end
end

B.new().print_i()    # Shows "ok"
```

If you can't agree with this behavior, let's forget about classes and inheritance. When there's an instance `obj` of the class `c`, then think as if all the methods of the superclass of `c` are defined in `c`. Of course we keep the overwrite rule in mind. Then the methods of `c` get attached to the instance `obj` (Fig.6). This strong palpability is a specialty of Ruby's object orientation.

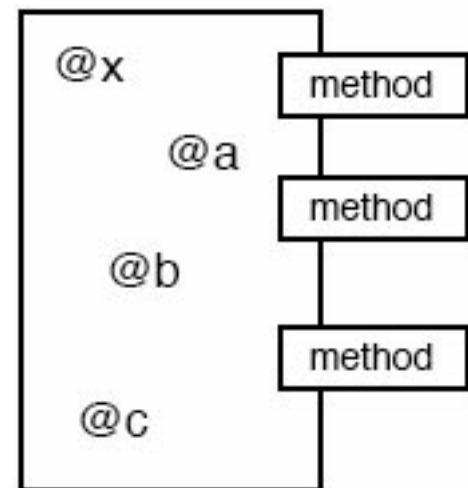


Figure 6: A conception of a Ruby object

# Modules

Only a single superclass can be designated. So Ruby looks like single inheritance. But because of *modules* it has in practice the ability which is identical to multiple inheritance. Let's explain these modules next.

In short, modules are classes for which a superclass cannot be designated and instances cannot be created. For the definition we write as follows.

```
module M
end
```

Here the module `M` was defined. Methods are defined exactly the same way as for classes.

```
module M
  def myupcase( str )
    return str.upcase()
  end
end
```

But because we cannot create instances, we cannot call them directly. To do that, we use the module by “including” it into other classes. Then we become to be able to deal with it as if a class inherited the module.

```
module M
  def myupcase( str )
    return str.upcase()
  end
```

```
end

class C
  include M
end

p(C.new().myupcase("content")) # "CONTENT" is shown
```

Even though no method was defined in the class `C` we can call the method `myupcase`. It means it “inherited” the method of the module `M`. Inclusion is functionally completely the same as inheritance. There’s no limit on defining methods or accessing instance variables.

I said we cannot specify any superclass of a module, but other modules can be included.

```
module M
end

module M2
  include M
end
```

In other words it’s functionally the same as appointing a superclass. But a class cannot come above a module. Only modules are allowed above modules.

The example below also contains the inheritance of methods.

```
module OneMore
  def method_OneMore()
    p("OneMore")
  end
```

```

end

module M
  include OneMore

  def method_M()
    p("M")
  end
end

class C
  include M
end

C.new().method_M()      # Output "M"
C.new().method_OneMore() # Output "OneMore"

```

As with classes when we sketch inheritance it looks like Fig.7

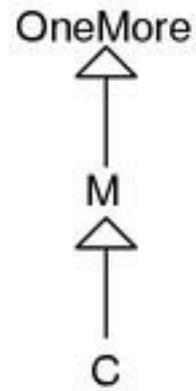


Figure 7: multilevel inclusion

Besides, the class `c` also has a superclass. How is its relationship to modules? For instance, let's think of the following case.

```

# modcls.rb

class Cls
  def test()
    return "class"
  end

```

```

end

module Mod
  def test()
    return "module"
  end
end

class C < Cls
  include Mod
end

p(B.new().test())  # "class"? "module"?

```

`C` inherits from `Cls` and includes `Mod`. Which will be shown in this case, "class" or "module"? In other words, which one is "closer", class or module? We'd better ask Ruby about Ruby, thus let's execute it:

```
% ruby modcls.rb
"module"
```

Apparently a module takes preference before the superclass.

In general, in Ruby when a module is included, it would be inherited by going in between the class and the superclass. As a picture it might look like Fig.8.

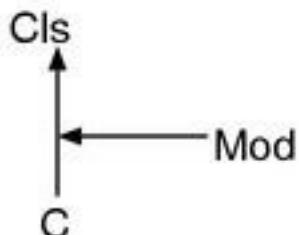


Figure 8: The relation between modules and classes

And if we also taking the modules included in the module into accounts, it would look like Fig.9.

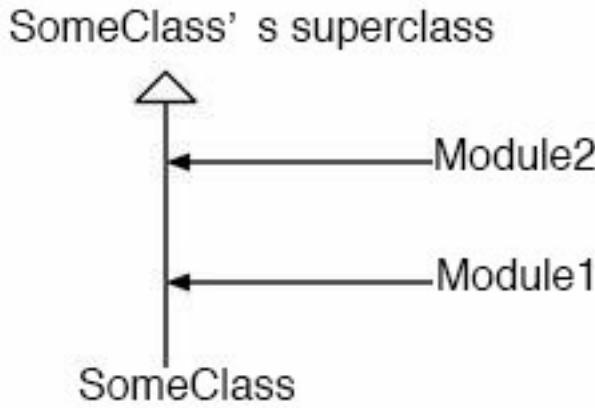


Figure 9: The relation between modules and classes(2)

## **The Program revisited**

---

Caution. This section is extremely important and explaining the elements which are not easy to mix with for programmers who have only used static languages before. For other parts just skimming is sufficient, but for only this part I'd like you to read it carefully. The explanation will also be relatively attentive.

### **Nesting of Constants**

First a repetition of constants. As a constant begins with a capital letter the definition goes as follows.

```
Const = 3
```

Now we reference the constant in this way.

```
p(Const) # Shows 3
```

Actually we can also write this.

```
p(::Const) # Shows 3 in the same way.
```

The `::` in front shows that it's a constant defined at the top level. You can think of the path in a filesystem. Assume there is a file `vmunix` in the root directory. Being at `/` one can write `vmunix` to access the file. One can also write `/vmunix` as its full path. It's the same with `Const` and `::Const`. At top level it's okay to write only `Const` or to write the full path `::Const`

And what corresponds to a filesystem's directories in Ruby? That should be class and module definition statements. However mentioning both is cumbersome, so I'll just subsume them under class definition. When one enters a class definition the level for constants rises ( as if entering a directory).

```
class SomeClass
  Const = 3
end

p(::SomeClass::Const) # Shows 3
p( SomeClass::Const) # The same. Shows 3
```

`SomeClass` is defined at toplevel. Hence one can reference it by writing either `SomeClass` or `::SomeClass`. And as the constant `Const`

nested in the class definition is a Const “inside SomeClass”, It becomes ::SomeClass::Const.

As we can create a directory in a directory, we can create a class inside a class. For instance like this:

```
class C      # ::C
  class C2    # ::C::C2
    class C3  # ::C::C2::C3
    end
  end
end
```

By the way, for a constant defined in a class definition statement, should we always write its full name? Of course not. As with the filesystem, if one is inside the same class definition one can skip the ::. It becomes like that:

```
class SomeClass
  Const = 3
  p(Const)  # Shows 3.
end
```

“What?” you might think. Surprisingly, even if it is in a class definition statement, we can write a program which is going to be executed. People who are used to only static languages will find this quite exceptional. I was also flabbergasted the first time I saw it.

Let’s add that we can of course also view a constant inside a method. The reference rules are the same as within the class definition (outside the method).

```
class C
  Const = "ok"
  def test()
    p(Const)
  end
end

C.new().test()  # Shows "ok"
```

## Everything is executed

Looking at the big picture I want to write one more thing. In Ruby almost the whole parts of program is “executed”. Constant definitions, class definitions and method definitions and almost all the rest is executed in the apparent order.

Look for instance at the following code. I used various constructions which have been used before.

```
1: p("first")
2:
3: class C < Object
4:   Const = "in C"
5:
6:   p(Const)
7:
8:   def myupcase(str)
9:     return str.upcase()
10:  end
11: end
12:
13: p(C.new().myupcase("content"))
```

This program is executed in the following order:

1: p("first")	Shows "first"
3: < Object	The constant Object is referenced and the class object Object is gained
3: class C	A new class object with superclass Object is generated, and assigned to the constant C
4: Const = "in C"	Assigning the value "in C" to the constant ::C::Const
6: p(Const)	Showing the constant ::C::Const hence "in C"
8: def myupcase(...)...end	Define C#myupcase
13: C.new().myupcase(...)	Refer the constant C, call the method new on it, and then myupcase on the return value
9: return str.upcase()	Returns "CONTENT"
13: p(...)	Shows "CONTENT"

## The Scope of Local Variables

At last we can talk about the scope of local variables.

The toplevel, the interior of a class definition, the interior of a module definition and a method body are all have each completely independent local variable scope. In other words, the `lvar` variables in the following program are all different variables, and they do not influence each other.

```

lvar = 'toplevel'

class C
  lvar = 'in C'
  def method()
    lvar = 'in C#method'
  end
end

```

```
end
end

p(lvar)  # Shows "toplevel"

module M
  lvar = 'in M'
end

p(lvar)  # Shows "toplevel"
```

## self as context

Previously, I said that during method execution oneself (an object on which the method was called) becomes `self`. That's true but only half true. Actually during the execution of a Ruby program, `self` is always set wherever it is. It means there's `self` also at the top level or in a class definition statement.

For instance the `self` at the toplevel is `main`. It's an instance of the `Object` class which is nothing special. `main` is provided to set up `self` for the time being. There's no deeper meaning attached to it.

Hence the toplevel's `self` i.e. `main` is an instance of `Object`, such that one can call the methods of `Object` there. And in `Object` the module `Kernel` is included. In there the function-flavor methods like `p` and `puts` are defined (Fig.10). That's why one can call `puts` and `p` also at the toplevel.

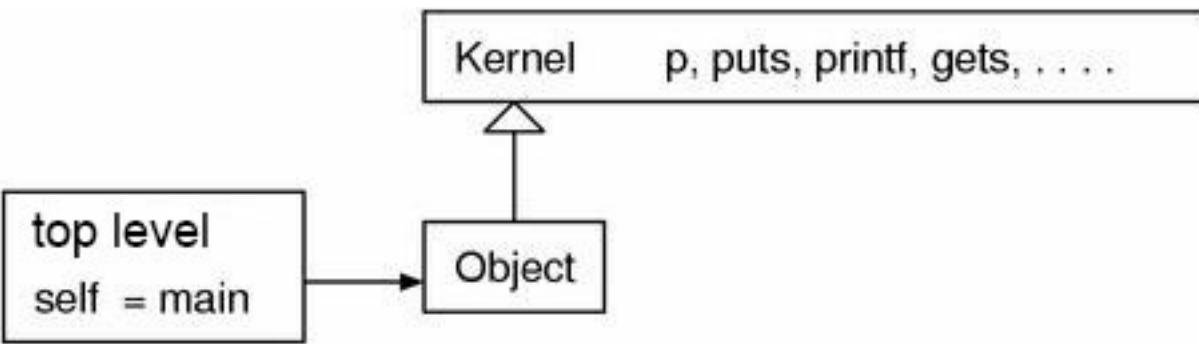


Figure 10: `main`, `Object` and `Kernel`

Thus `p` isn't a function, it's a method. Just because it is defined in `Kernel` and thus can be called like a function as "its own" method wherever it is or no matter what the class of `self` is. Therefore, there aren't functions in the true sense, there are only methods.

By the way, besides `p` and `puts` there are the function-flavor methods `print`, `puts`, `printf`, `sprintf`, `gets`, `fork`, and `exec` and many more with somewhat familiar names. When you look at the choice of names you might be able to imagine Ruby's character.

Well, since `self` is setup everywhere, `self` should also be in a class definition in the same way. The `self` in the class definition is the class itself (the class object). Hence it would look like this.

```

class C
  p(self)  # C
end

```

What should this be good for? In fact, we've already seen an example in which it is very useful. This one.

```

module M
end

```

```
class C
  include M
end
```

This `include` is actually a method call to the class object `c`. I haven't mentioned it yet but the parentheses around arguments can be omitted for method calls. And I omitted the parentheses around `include` such that it doesn't look like a method call because we have not finished the talk about class definition statement.

## ■ Loading

In Ruby the loading of libraries also happens at runtime. Normally one writes this.

```
require("library_name")
```

The impression isn't false, `require` is a method. It's not even a reserved word. When it is written this way, loading is executed on the line it is written, and the execution is handed over to (the code of) the library. As there is no concept like Java packages in Ruby, when we'd like to separate namespaces, it is done by putting files into a directory.

```
require("somelib/file1")
require("somelib/file2")
```

And in the library usually classes and such are defined with `class` statements or `module` statements. The constant scope of the top

level is flat without the distinction of files, so one can see classes defined in another file without any special preparation. To partition the namespace of class names one has to explicitly nest modules as shown below.

```
# example of the namespace partition of net library
module Net
  class SMTP
    # ...
  end
  class POP
    # ...
  end
  class HTTP
    # ...
  end
end
```

## More about Classes

---

### ▀ The talk about Constants still goes on

Up to now we used the filesystem metaphor for the scope of constants, but I want you to completely forget that.

There is more about constants. Firstly one can also see constants in the “outer” class.

```
Const = "ok"
class C
  p(Const)  # Shows "ok"
```

```
end
```

The reason why this is designed in this way is because this becomes useful when modules are used as namespaces. Let's explain this by adding a few things to the previous example of `net` library.

```
module Net
  class SMTP
    # Uses Net::SMTPHelper in the methods
  end
  class SMTPHelper # Supports the class Net::SMTP
  end
end
```

In such case, it's convenient if we can refer to it also from the `SMTP` class just by writing `SMTPHelper`, isn't it? Therefore, it is concluded that “it's convenient if we can see the outer classes”.

The outer class can be referenced no matter how many times it is nesting. When the same name is defined on different levels, the one which will first be found from within will be referred to.

```
Const = "far"
class C
  Const = "near" # This one is closer than the one above
  class C2
    class C3
      p(Const) # "near" is shown
    end
  end
end
```

There's another way of searching constants. If the toplevel is

reached when going further and further outside then the own superclass is searched for the constant.

```
class A
  Const = "ok"
end
class B < A
  p(Const)  # "ok" is shown
end
```

Really, that's pretty complicated.

Let's summarize. When looking up a constant, first the outer classes is searched then the superclasses. This is quite contrived, but let's assume a class hierarchy as follows.

```
class A1
end
class A2 < A1
end
class A3 < A2
  class B1
  end
  class B2 < B1
  end
  class B3 < B2
    class C1
    end
    class C2 < C1
    end
    class C3 < C2
      p(Const)
    end
  end
end
```

When the constant `Const` in `C3` is referenced, it's looked up in the

order depicted in Fig.11.

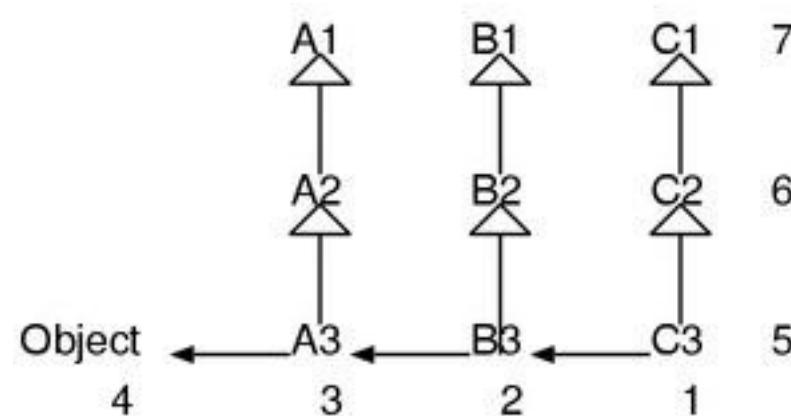


Figure 11: Search order for constants

Be careful about one point. The superclasses of the classes outside, for instance A1 and B2, aren't searched at all. If it's outside once it's always outside and if it's superclass once it's always superclass. Otherwise, the number of classes searched would become too big and the behavior of such complicated thing would become unpredictable.

## ■ Metaclasses

I said that a method can be called on if it is an object. I also said that the methods that can be called are determined by the class of an object. Then shouldn't there be a class for class objects? (Fig.12)

"string" —————→ String —————→ ????

Figure 12: A class of classes?

In this kind of situation, in Ruby, we can check in practice. It's because there's "a method which returns the class (class object) to

which an object itself belongs”, `Object#class`.

```
p("string".class())      # String is shown
p(String.class())        # Class is shown
p(Object.class())        # Class is shown
```

Apparently `String` belongs to the class named `class`. Then what's the class of `class`?

```
p(Class.class())      # Class is shown
```

Again `Class`. In other words, whatever object it is, by following like `.class().class().class() ...`, it would reach `Class` in the end, then it will stall in the loop (Fig.13).

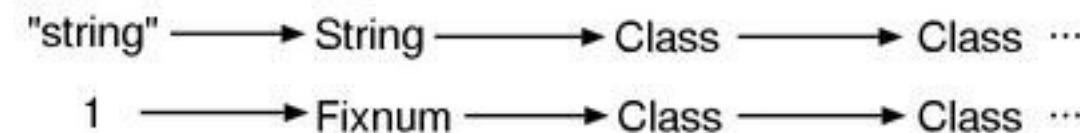


Figure 13: The class of the class of the class...

Class is the class of classes. And what has a recursive structure as “X of X” is called a meta-X. Hence Class is a metaclass.

# Metaobjects

Let's change the target and think about modules. As modules are also objects, there also should be a class for them. Let's see.

```
module M
end
```

```
p(M.class()) # Module is shown
```

The class of a module seems to be `Module`. And what should be the class of the class `Module`?

```
p(Module.class()) # Class
```

It's again `Class`

Now we change the direction and examine the inheritance relationships. What's the superclass of `Class` and `Module`? In Ruby, we can find it out with `Class#superclass`.

```
p(Class.superclass()) # Module
p(Module.superclass()) # Object
p(Object.superclass()) # nil
```

So `Class` is a subclass of `Module`. Based on these facts, Figure 14 shows the relationships between the important classes of Ruby.

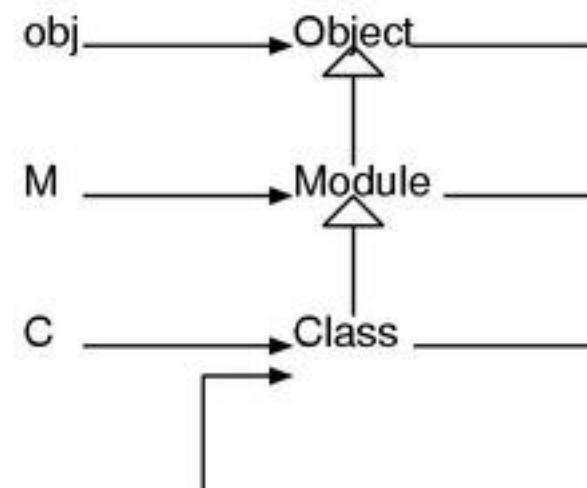


Figure 14: The class relationship between the important Ruby classes

Up to now we used `new` and `include` without any explanation, but finally I can explain their true form. `new` is really a method defined for the class `Class`. Therefore on whatever class, (because it is an instance of `Class`), `new` can be used immediately. But `new` isn't defined in `Module`. Hence it's not possible to create instances in a module. And since `include` is defined in the `Module` class, it can be called on both modules and classes.

These three classes `Object`, `Module` and `Class` are objects that support the foundation of Ruby. We can say that these three objects describe the Ruby's object world itself. Namely they are objects which describe objects. Hence, `Object` `Module` `Class` are Ruby's "meta-objects".

## ■ Singleton Methods

I said that methods can be called if it is an object. I also said that the methods that can be called are determined by the object's class. However I think I also said that ideally methods belong to objects. Classes are just a means to eliminate the effort of defining the same method more than once.

Actually In Ruby there's also a means to define methods for individual objects (instances) not depending on the class. To do this, you can write this way.

```
obj = Object.new()
def obj.my_first()
  puts("My first singleton method")
```

```
end
obj.my_first()  # Shows My first singleton method
```

As you already know `Object` is the root for every class. It's very unlikely that a method whose name is so weird like `my_first` is defined in such important class. And `obj` is an instance of `Object`. However the method `my_first` can be called on `obj`. Hence we have created without doubt a method which has nothing to do with the class the object belongs to. These methods which are defined for each object individually are called singleton methods.

When are singleton methods used? First, it is used when defining something like static methods of Java or C++. In other words methods which can be used without creating an instance. These methods are expressed in Ruby as singleton methods of a class object.

For example in UNIX there's a system call `unlink`. This command deletes a file entry from the filesystem. In Ruby it can be used directly as the singleton method `unlink` of the `File` class. Let's try it out.

```
File.unlink("core")  # deletes the coredump
```

It's cumbersome to say “the singleton method `unlink` of the object `File`”. We simply write `File.unlink`. Don't mix it up and write `File#unlink`, or vice versa don't write `File.write` for the method `write` defined in `File`.

## ▼ A summary of the method notation

notation	the target object	example
File.unlink	the Fileclass itself	File.unlink("core")
File#write	an instance of File	f.write("str")

## Class Variables

Class variables were added to Ruby from 1.6 on, they are a relatively new mechanism. As with constants, they belong to a class, and they can be referenced and assigned from both the class and its instances. Let's look at an example. The beginning of the name is @@.

```
class C
  @@cvar = "ok"
  p(@@cvar)      # "ok" is shown

  def print_cvar()
    p(@@cvar)
  end
end

C.new().print_cvar()  # "ok" is shown
```

As the first assignment serves as the definition, a reference before an assignment like the one shown below leads to a runtime error. There is an '@' in front but the behavior differs completely from instance variables.

```
% ruby -e '
class C
  @@cvar
```

```
end
'
-e:3: uninitialized class variable @@cvar in C (NameError)
```

Here I was a bit lazy and used the -e option. The program is the three lines between the single quotes.

Class variables are inherited. Or saying it differently, a variable in a superior class can be assigned and referenced in the inferior class.

```
class A
  @@cvar = "ok"
end

class B < A
  p(@@cvar)          # Shows "ok"
  def print_cvar()
    p(@@cvar)
  end
end

B.new().print_cvar()  # Shows "ok"
```

## Global Variables

---

At last there are also global variables. They can be referenced from everywhere and assigned everywhere. The first letter of the name is a \$.

```
$gvar = "global variable"
p($gvar)  # Shows "global variable"
```

As with instance variables, all kinds of names can be considered defined for global variables before assignments. In other words a reference before an assignment gives a `nil` and doesn't raise an error.

---

Copyright © 2002-2004 Minero Aoki, All rights reserved.

English Translation: Sebastian Krause <[skra@pantolog.de](mailto:skra@pantolog.de)>

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

Translated by Vincent ISAMBART

# Chapter 2: Objects

## Structure of Ruby objects

---

### Guideline

From this chapter, we will begin actually exploring the `ruby` source code. First, as declared at the beginning of this book, we'll start with the object structure.

What are the necessary conditions for objects to be objects? There could be many ways to explain about object itself, but there are only three conditions that are truly indispensable.

1. The ability to differentiate itself from other objects (an identity)
2. The ability to respond to messages (methods)
3. The ability to store internal state (instance variables)

In this chapter, we are going to confirm these three features one by one.

The target file is mainly `ruby.h`, but we will also briefly look at other files such as `object.c`, `class.c` or `variable.c`.

# VALUE and object struct

In ruby, the body of an object is expressed by a struct and always handled via a pointer. A different struct type is used for each class, but the pointer type will always be `VALUE` (figure 1).

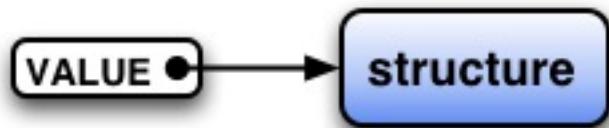


Figure 1: `VALUE` and `struct`

Here is the definition of `VALUE`:

## ▼ `VALUE`

```
71  typedef unsigned long VALUE;
```

```
(ruby.h)
```

In practice, when using a `VALUE`, we cast it to the pointer to each object struct. Therefore if an `unsigned long` and a pointer have a different size, ruby will not work well. Strictly speaking, it will not work if there's a pointer type that is bigger than `sizeof(unsigned long)`. Fortunately, systems which could not meet this requirement is unlikely recently, but some time ago it seems there were quite a few of them.

The structs, on the other hand, have several variations, a different

struct is used based on the class of the object.

struct RObject	all things for which none of the following applies
struct RClass	class object
struct RFloat	small numbers
struct RString	string
struct RArray	array
struct RRegexp	regular expression
struct RHash	hash table
struct RFile	IO, File, Socket, etc...
struct RData	all the classes defined at C level, except the ones mentioned above
struct RStruct	Ruby's Struct class
struct RBignum	big integers

For example, for an string object, `struct RString` is used, so we will have something like the following.

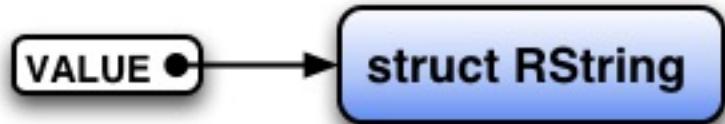


Figure 2: String object

Let's look at the definition of a few object structs.

## ▼ Examples of object struct

```
  /* struct for ordinary objects */
295 struct RObject {
296     struct RBasic basic;
297     struct st_table *iv_tbl;
298 };

  /* struct for strings (instance of String) */
314 struct RString {
315     struct RBasic basic;
316     long len;
317     char *ptr;
318     union {
319         long capa;
320         VALUE shared;
321     } aux;
322 };

  /* struct for arrays (instance of Array) */
324 struct RArray {
325     struct RBasic basic;
326     long len;
327     union {
328         long capa;
329         VALUE shared;
330     } aux;
331     VALUE *ptr;
332 };

(ruby.h)
```

Before looking at every one of them in detail, let's begin with something more general.

First, as `VALUE` is defined as `unsigned long`, it must be cast before being used when it is used as a pointer. That's why `Rxxxx()` macros have been made for each object struct. For example, for `struct`

RString there is RSTRING(), for struct RArray there is RARRAY(), etc...

These macros are used like this:

```
VALUE str = ....;
VALUE arr = ....;
RSTRING(str)->len; /* ((struct RString*)str)->len */
RARRAY(arr)->len; /* ((struct RArray*)arr)->len */
```

Another important point to mention is that all object structs start with a member `basic` of type `struct RBasic`. As a result, if you cast this `VALUE` to `struct RBasic*`, you will be able to access the content of `basic`, regardless of the type of struct pointed to by `VALUE`.

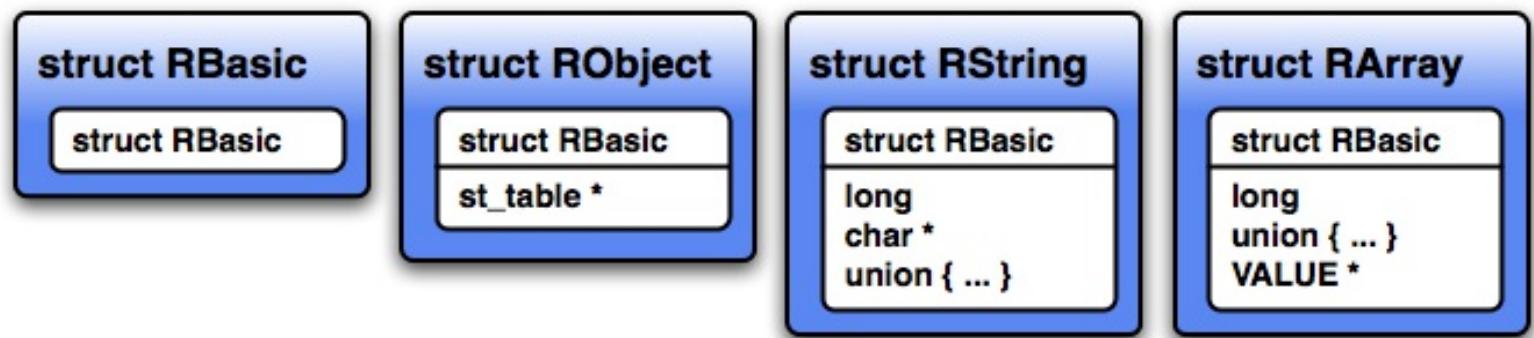


Figure 3: struct RBasic

Because it is purposefully designed this way, `struct RBasic` must contain very important information for Ruby objects. Here is the definition for `struct RBasic`:

#### ▼ struct RBasic

```
290 struct RBasic {
291     unsigned long flags;
292     VALUE klass;
```

```
293 };
```

```
(ruby.h)
```

flags are multipurpose flags, mostly used to register the struct type (for instance struct RObject). The type flags are named `T_xxxx`, and can be obtained from a `VALUE` using the macro `TYPE()`. Here is an example:

```
VALUE str;  
str = rb_str_new();      /* creates a Ruby string (its struct is F  
TYPE(str);           /* the return value is T_STRING */
```

The all flags are named as `T_xxxx`, like `T_STRING` for struct RString and `T_ARRAY` for struct RArray. They are very straightforwardly corresponded to the type names.

The other member of struct RBasic, `klass`, contains the class this object belongs to. As the `klass` member is of type `VALUE`, what is stored is (a pointer to) a Ruby object. In short, it is a class object.

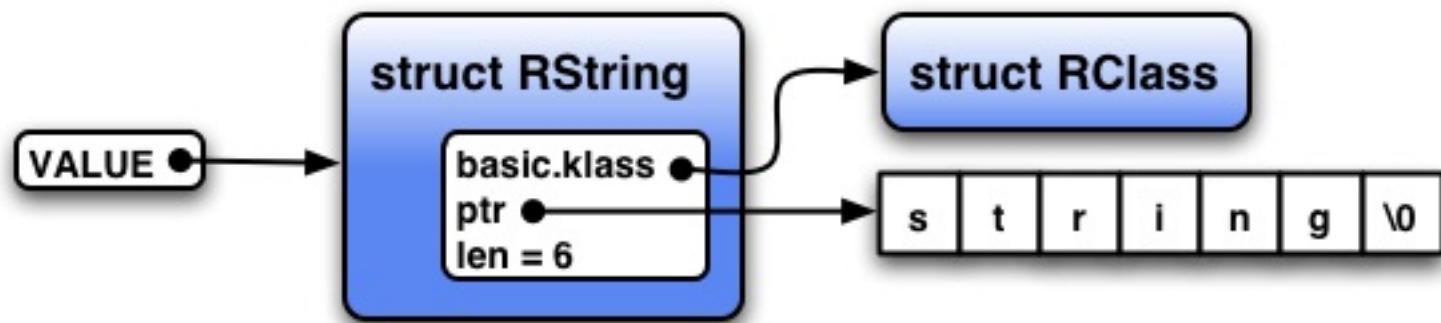


Figure 4: object and class

The relation between an object and its class will be detailed in the

“Methods” section of this chapter.

By the way, this member is named `klass` so as not to conflict with the reserved word `class` when the file is processed by a C++ compiler.

## About struct types

I said that the type of struct is stored in the `flags` member of `struct Basic`. But why do we have to store the type of struct? It’s to be able to handle all different types of struct via `VALUE`. If you cast a pointer to a struct to `VALUE`, as the type information does not remain, the compiler won’t be able to help. Therefore we have to manage the type ourselves. That’s the consequence of being able to handle all the struct types in a unified way.

OK, but the used struct is defined by the class so why are the struct type and class are stored separately? Being able to find the struct type from the class should be enough. There are two reasons for not doing this.

The first one is (I’m sorry for contradicting what I said before), in fact there are structs that do not have a `struct RBasic` (i.e. they have no `klass` member). For example `struct RNode` that will appear in the second part of the book. However, `flags` is guaranteed to be in the beginning members even in special structs like this. So if you put the type of struct in `flags`, all the object structs can be differentiated in one unified way.

The second reason is that there is no one-to-one correspondence between class and struct. For example, all the instances of classes defined at the Ruby level use `struct RObject`, so finding a struct from a class would require to keep the correspondence between each class and struct. That's why it's easier and faster to put the information about the type in the struct.

## The use of `basic.flags`

Regarding the use of `basic.flags`, because I feel bad to say it is the struct type “and such”, I’ll illustrate it entirely here. (Figure 5) There is no need to understand everything right away, because this is prepared for the time when you will be wondering about it later.

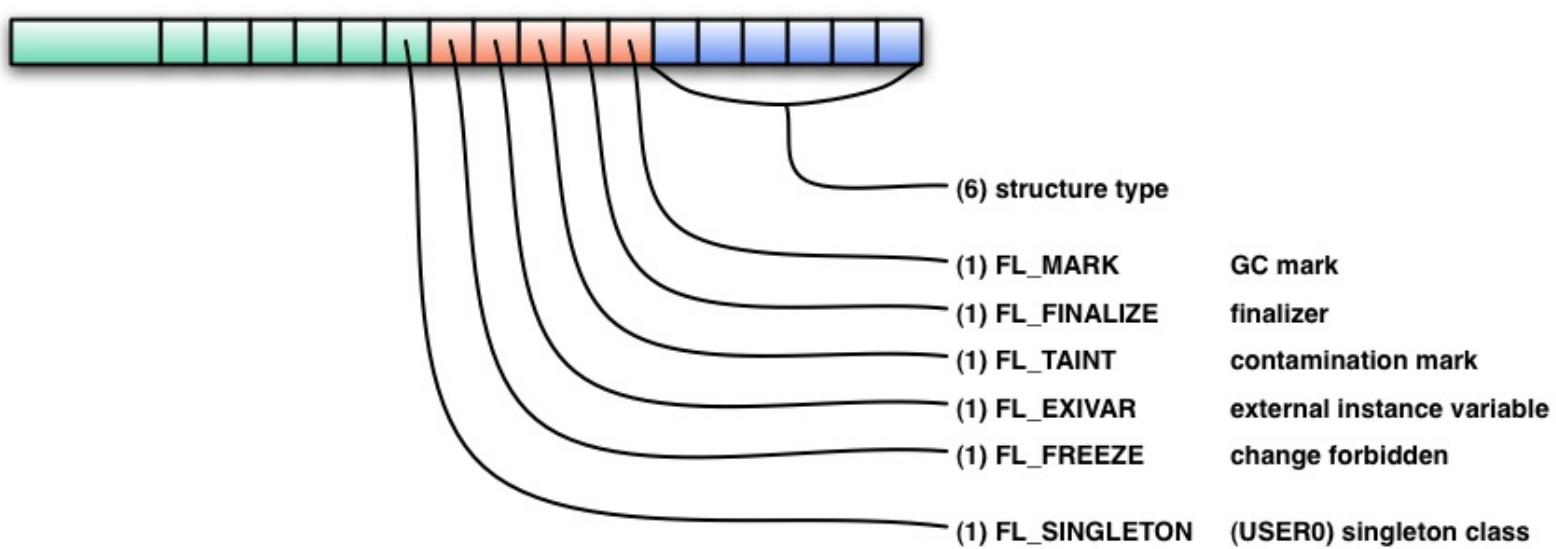


Figure 5: Use of `flags`

When looking at the diagram, it looks like that 21 bits are not used on 32 bit machines. On these additional bits, the flags `FL_USER0` to `FL_USER8` are defined, and are used for a different purpose for each

struct. In the diagram I also put `FL_USER0` (`FL_SINGLETON`) as an example.

## Objects embedded in VALUE

As I said, `VALUE` is an `unsigned long`. As `VALUE` is a pointer, it may look like `void*` would also be all right, but there is a reason for not doing this. In fact, `VALUE` can also not be a pointer. The 6 cases for which `VALUE` is not a pointer are the following:

1. small integers
2. symbols
3. `true`
4. `false`
5. `nil`
6. `Qundef`

I'll explain them one by one.

### Small integers

All data are objects in Ruby, thus integers are also objects. But since there are so many kind of integer objects, if each of them is expressed as a struct, it would risk slowing down execution significantly. For example, when incrementing from 0 to 50000, we would hesitate to create 50000 objects for only that purpose.

That's why in ruby, integers that are small to some extent are

treated specially and embedded directly into `VALUE`. “Small” means signed integers that can be held in `sizeof(VALUE)*8-1` bits. In other words, on 32 bits machines, the integers have 1 bit for the sign, and 30 bits for the integer part. Integers in this range will belong to the `Fixnum` class and the other integers will belong to the `Bignum` class.

Let’s see in practice the `INT2FIX()` macro that converts from a C `int` to a `Fixnum`, and confirm that `Fixnum` are directly embedded in `VALUE`.

## ▼ INT2FIX

```
123 #define INT2FIX(i) ((VALUE)((long)(i))<<1 | FIXNUM_FLAG))  
122 #define FIXNUM_FLAG 0x01
```

(`ruby.h`)

In brief, shift 1 bit to the left, and bitwise or it with 1.

`110100001000` before conversion

`1101000010001` after conversion

That means that `Fixnum` as `VALUE` will always be an odd number. On the other hand, as Ruby object structs are allocated with `malloc()`, they are generally arranged on addresses multiple of 4. So they do not overlap with the values of `Fixnum` as `VALUE`.

Also, to convert `int` or `long` to `VALUE`, we can use macros like `INT2NUM()` or `LONG2NUM()`. Any conversion macro `xxxx2xxxx` with a name containing `NUM` can manage both `Fixnum` and `Bignum`. For example if `INT2NUM()` can’t convert an integer into a `Fixnum`, it will

automatically convert it to `Bignum`. `NUM2INT()` will convert both `Fixnum` and `Bignum` to `int`. If the number can't fit in an `int`, an exception will be raised, so there is no need to check the value range.

# Symbols

What are symbols?

As this question is quite troublesome to answer, let's start with the reasons why symbols were necessary. In the first place, there's a type named `ID` used inside `ruby`. Here it is.

## ▼ `ID`

```
72  typedef unsigned long ID;  
(ruby.h)
```

This `ID` is a number having a one-to-one association with a string. However, it's not possible to have an association between all strings in this world and numerical values. It is limited to the one to one relationships inside one `ruby` process. I'll speak of the method to find an `ID` in the next chapter "Names and name tables".

In language processor, there are a lot of names to handle. Method names or variable names, constant names, file names, class names... It's troublesome to handle all of them as strings (`char*`), because of memory management and memory management and memory management... Also, lots of comparisons would certainly

be necessary, but comparing strings character by character will slow down the execution. That's why strings are not handled directly, something will be associated and used instead. And generally that "something" will be integers, as they are the simplest to handle.

These ID are found as symbols in the Ruby world. Up to ruby 1.4, the values of ID converted to Fixnum were used as symbols. Even today these values can be obtained using `Symbol#to_i`. However, as real use results came piling up, it was understood that making Fixnum and Symbol the same was not a good idea, so since 1.6 an independent class Symbol has been created.

Symbol objects are used a lot, especially as keys for hash tables. That's why Symbol, like Fixnum, was made embedded in VALUE. Let's look at the `ID2SYM()` macro converting ID to Symbol object.

## ▼ ID2SYM

```
158 #define SYMBOL_FLAG 0x0e
160 #define ID2SYM(x) (((VALUE)((long)(x))<<8|SYMBOL_FLAG))
(ruby.h)
```

When shifting 8 bits left, x becomes a multiple of 256, that means a multiple of 4. Then after with a bitwise or (in this case it's the same as adding) with `0x0e` (14 in decimal), the VALUE expressing the symbol is not a multiple of 4. Or even an odd number. So it does not overlap the range of any other VALUE. Quite a clever trick.

Finally, let's see the reverse conversion of `ID2SYM()`, `SYM2ID()`.

## ▼ `SYM2ID()`

```
161 #define SYM2ID(x) RSHIFT((long)x,8)  
(ruby.h)
```

`RSHIFT` is a bit shift to the right. As right shift may keep or not the sign depending of the platform, it became a macro.

## `true` `false` `nil`

These three are Ruby special objects. `true` and `false` represent the boolean values. `nil` is an object used to denote that there is no object. Their values at the C level are defined like this:

## ▼ `true` `false` `nil`

```
164 #define Qfalse 0          /* Ruby's false */  
165 #define Qtrue  2          /* Ruby's true */  
166 #define Qnil   4          /* Ruby's nil */  
(ruby.h)
```

This time it's even numbers, but as `0` or `2` can't be used by pointers, they can't overlap with other `VALUE`. It's because usually the first block of virtual memory is not allocated, to make the programs dereferencing a `NULL` pointer crash.

And as `Qfalse` is `0`, it can also be used as `false` at C level. In practice,

in ruby, when a function returns a boolean value, it's often made to return an `int` or `VALUE`, and returns `Qtrue/Qfalse`.

For `Qnil`, there is a macro dedicated to check if a `VALUE` is `Qnil` or not, `NIL_P()`.

### ▼ `NIL_P()`

```
170 #define NIL_P(v) (((VALUE)(v) == Qnil)  
(ruby.h)
```

The name ending with `p` is a notation coming from Lisp denoting that it is a function returning a boolean value. In other words, `NIL_P` means “is the argument `nil`?”. It seems the “`p`” character comes from “predicate.” This naming rule is used at many different places in ruby.

Also, in Ruby, `false` and `nil` are false (in conditional statements) and all the other objects are true. However, in C, `nil` (`Qnil`) is true. That's why there's the `RTEST()` macro to do Ruby-style test in C.

### ▼ `RTEST()`

```
169 #define RTEST(v) (((VALUE)(v) & ~Qnil) != 0)  
(ruby.h)
```

As in `Qnil` only the third lower bit is `1`, in `~Qnil` only the third lower bit is `0`. Then only `Qfalse` and `Qnil` become `0` with a bitwise and.

`!=0` has been added to be certain to only have 0 or 1, to satisfy the requirements of the glib library that only wants 0 or 1 ([\[ruby-dev:11049\]](#)).

By the way, what is the ‘0’ of `Qnil`? ‘R’ I would have understood but why ‘0’? When I asked, the answer was “Because it’s like that in Emacs.” I did not have the fun answer I was expecting...

## Qundef

### ▼ Qundef

```
167 #define Qundef 6 /* undefined value for place
(ruby.h)
```

This value is used to express an undefined value in the interpreter. It can’t (must not) be found at all at the Ruby level.

## Methods

---

I already brought up the three important points of a Ruby object: having an identity, being able to call a method, and keeping data for each instance. In this section, I’ll explain in a simple way the structure linking objects and methods.

struct RClass

In Ruby, classes exist as objects during the execution. Of course. So there must be a struct for class objects. That struct is struct `RClass`. Its struct type flag is `T_CLASS`.

As classes and modules are very similar, there is no need to differentiate their content. That's why modules also use the struct `RClass` struct, and are differentiated by the `T_MODULE` struct flag.

## ▼ struct `RClass`

```
300 struct RClass {  
301     struct RBasic basic;  
302     struct st_table *iv_tbl;  
303     struct st_table *m_tbl;  
304     VALUE super;  
305 };
```

(`ruby.h`)

First, let's focus on the `m_tbl` (Method TaBLE) member. `struct st_table` is an hashtable used everywhere in `ruby`. Its details will be explained in the next chapter “Names and name tables”, but basically, it is a table mapping names to objects. In the case of `m_tbl`, it keeps the correspondence between the name (`ID`) of the methods possessed by this class and the methods entity itself. As for the structure of the method entity, it will be explained in Part 2 and Part 3.

The fourth member `super` keeps, like its name suggests, the superclass. As it's a `VALUE`, it's (a pointer to) the class object of the superclass. In Ruby there is only one class that has no superclass

(the root class): `Object`.

However I already said that all `Object` methods are defined in the `Kernel` module, `Object` just includes it. As modules are functionally similar to multiple inheritance, it may seem having just `super` is problematic, but in `ruby` some clever conversions are made to make it look like single inheritance. The details of this process will be explained in the fourth chapter “Classes and modules”.

Because of this conversion, `super` of the struct of `Object` points to struct `RClass` which is the entity of `Kernel` object and the `super` of `Kernel` is `NULL`. So to put it conversely, if `super` is `NULL`, its `RClass` is the entity of `Kernel` (figure 6).

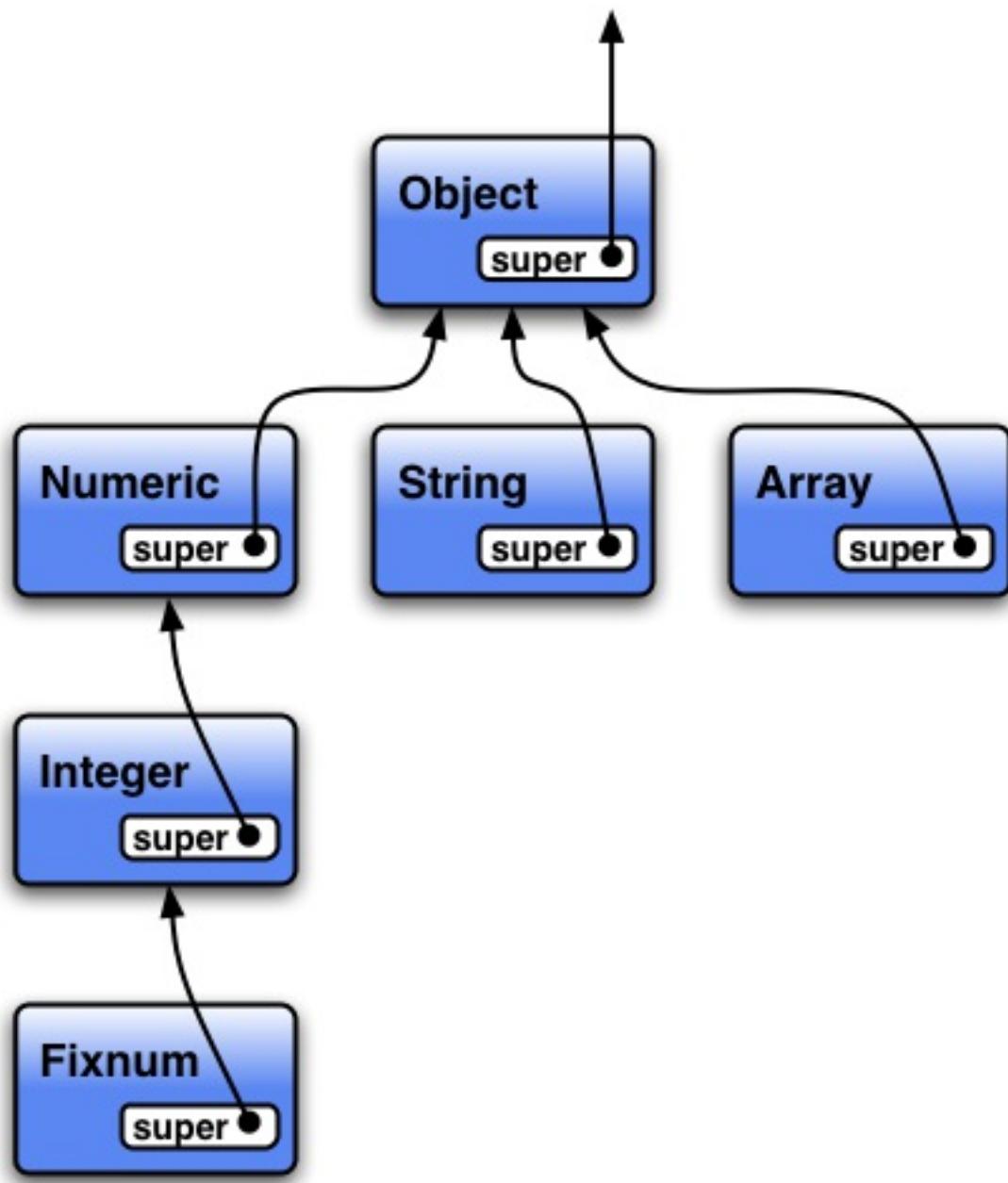


Figure 6: Class tree at the C level

## Methods search

With classes structured like this, you can easily imagine the method call process. The `m_tbl` of the object's class is searched, and if the method was not found, the `m_tbl` of super is searched, and so on. If there is no more super, that is to say the method was not found even in `Object`, then it must not be defined.

The sequential search process in `m_tbl` is done by `search_method()`.

▼ `search_method()`

```
256 static NODE*
257 search_method(klass, id, origin)
258     VALUE klass, *origin;
259     ID id;
260 {
261     NODE *body;
262
263     if (!klass) return 0;
264     while (!st_lookup(RCLASS(klass)->m_tbl, id, &body)) {
265         klass = RCLASS(klass)->super;
266         if (!klass) return 0;
267     }
268
269     if (origin) *origin = klass;
270     return body;
271 }
```

`(eval.c)`

This function searches the method named `id` in the class object `klass`.

`RCLASS(value)` is the macro doing:

```
((struct RClass*)(value))
```

`st_lookup()` is a function that searches in `st_table` the value corresponding to a key. If the value is found, the function returns true and puts the found value at the address given in third parameter (`&body`).

Nevertheless, doing this search each time whatever the circumstances would be too slow. That's why in reality, once called, a method is cached. So starting from the second time it will be found without following `super` one by one. This cache and its search will be seen in the 15th chapter “Methods”.

## Instance variables

---

In this section, I will explain the implementation of the third essential condition, instance variables.

### ■ `rb_ivar_set()`

Instance variable is the mechanism that allows each object to hold its specific data. Since it is specific to each object, it seems good to store it in each object itself (i.e. in its object struct), but is it really so? Let's look at the function `rb_ivar_set()`, which assigns an object to an instance variable.

### ▼ `rb_ivar_set()`

```
      /* assign val to the id instance variable of obj */
984  VALUE
985  rb_ivar_set(obj, id, val)
986      VALUE obj;
987      ID id;
988      VALUE val;
989  {
```

```

990     if ( !OBJ_TAINTED(obj) && rb_safe_level() >= 4)
991         rb_raise(rb_eSecurityError,
992                 "Insecure: can't modify instance variable")
993     if (OBJ_FROZEN(obj)) rb_error_frozen("object");
994     switch (TYPE(obj)) {
995         case T_OBJECT:
996         case T_CLASS:
997         case T_MODULE:
998             if ( !ROBJECT(obj)->iv_tbl)
999                 ROBJECT(obj)->iv_tbl = st_init_numtable();
1000             st_insert(ROBJECT(obj)->iv_tbl, id, val);
1001             break;
1002         default:
1003             generic_ivar_set(obj, id, val);
1004             break;
1005     }
1006     return val;
1007 }
```

(variable.c)

`rb_raise()` and `rb_error_frozen()` are both error checks. This can always be said hereafter: Error checks are necessary in reality, but it's not the main part of the process. Therefore, we should wholly ignore them at first read.

After removing the error handling, only the `switch` remains, but

```

switch (TYPE(obj)) {
    case T_aaaa:
    case T_bbbb:
    ...
}
```

this form is an idiom of ruby. `TYPE()` is the macro returning the type flag of the object struct (`T_OBJECT`, `T_STRING`, etc.). In other words as

the type flag is an integer constant, we can branch depending on it with a switch. Fixnum or Symbol do not have structs, but inside TYPE() a special treatment is done to properly return T\_FIXNUM and T\_SYMBOL, so there's no need to worry.

Well, let's go back to rb\_ivar\_set(). It seems only the treatments of T\_OBJECT, T\_CLASS and T\_MODULE are different. These 3 have been chosen on the basis that their second member is iv\_tbl. Let's confirm it in practice.

## ▼ Structs whose second member is iv\_tbl

```
295  /* TYPE(val) == T_OBJECT */
296  struct RObject {
297      struct RBasic basic;
298      struct st_table *iv_tbl;
299  };

300  /* TYPE(val) == T_CLASS or T_MODULE */
301  struct RClass {
302      struct RBasic basic;
303      struct st_table *iv_tbl;
304      struct st_table *m_tbl;
305      VALUE super;
306  };

(ruby.h)
```

iv\_tbl is the Instance Variable TaBLE. It records the correspondences between the instance variable names and their values.

In rb\_ivar\_set(), let's look again the code for the structs having

iv\_tbl.

```
if (!ROBJECT(obj)->iv_tbl)
  ROBJECT(obj)->iv_tbl = st_init_numtable();
st_insert(ROBJECT(obj)->iv_tbl, id, val);
break;
```

`ROBJECT()` is a macro that casts a `VALUE` into a `struct RObject*`. It's possible that what `obj` points to is actually a `struct RClass`, but when accessing only the second member, no problem will occur.

`st_init_numtable()` is a function creating a new `st_table`. `st_insert()` is a function doing associations in a `st_table`.

In conclusion, this code does the following: if `iv_tbl` does not exist, it creates it, then stores the `[variable name → object]` association.

There's one thing to be careful about. As `struct RClass` is the struct of a class object, its instance variable table is for the class object itself. In Ruby programs, it corresponds to something like the following:

```
class C
  @ivar = "content"
end
```

## generic\_ivar\_set()

What happens when assigning to an instance variable of an object whose struct is not one of `T_OBJECT T_MODULE T_CLASS`?

## ▼ rb\_ivar\_set() in the case there is no iv\_tbl

```
1000 default:  
1001     generic_ivar_set(obj, id, val);  
1002     break;  
  
(variable.c)
```

This is delegated to `generic_ivar_set()`. Before looking at this function, let's first explain its general idea.

Structs that are not `T_OBJECT`, `T_MODULE` or `T_CLASS` do not have an `iv_tbl` member (the reason why they do not have it will be explained later). However, even if it does not have the member, if there's another method linking an instance to a struct `st_table`, it would be able to have instance variables. In `ruby`, these associations are solved by using a global `st_table`, `generic_iv_table` (figure 7).

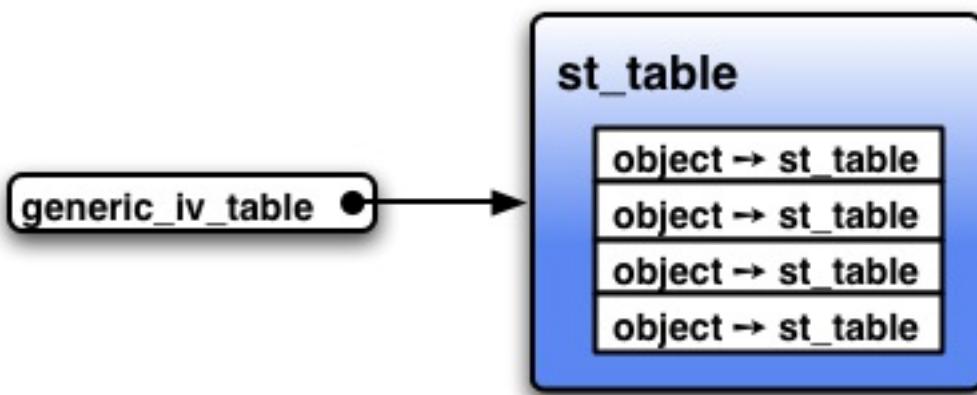


Figure 7: `generic_iv_table`

Let's see this in practice.

## ▼ `generic_ivar_set()`

```
801 static st_table *generic_iv_tbl;
830 static void
831 generic_ivar_set(obj, id, val)
832     VALUE obj;
833     ID id;
834     VALUE val;
835 {
836     st_table *tbl;
837
838     /* for the time being you can ignore this */
839     if (rb_special_const_p(obj)) {
840         special_generic_ivar = 1;
841     }
842     /* initialize generic_iv_tbl if it does not exist */
843     if (!generic_iv_tbl) {
844         generic_iv_tbl = st_init_numtable();
845     }
846     /* the process itself */
847     if (!st_lookup(generic_iv_tbl, obj, &tbl)) {
848         FL_SET(obj, FL_EXIVAR);
849         tbl = st_init_numtable();
850         st_add_direct(generic_iv_tbl, obj, tbl);
851         st_add_direct(tbl, id, val);
852         return;
853     }
854     st_insert(tbl, id, val);
855 }
```

(variable.c)

`rb_special_const_p()` is true when its parameter is not a pointer. However, as this `if` part requires knowledge of the garbage collector, we'll skip it for now. I'd like you to check it again after reading the chapter 5 “Garbage collection”.

`st_init_numtable()` already appeared some time ago. It creates a

new hash table.

`st_lookup()` searches a value corresponding to a key. In this case it searches for what's attached to `obj`. If an attached value can be found, the whole function returns true and stores the value at the address (`&tbl`) given as third parameter. In short, `!st_lookup(...)` can be read “if a value can't be found”.

`st_insert()` was also already explained. It stores a new association in a table.

`st_add_direct()` is similar to `st_insert()`, but it does not check if the key was already stored before adding an association. It means, in the case of `st_add_direct()`, if a key already registered is being used, two associations linked to this same key will be stored. We can use `st_add_direct()` only when the check for existence has already been done, or when a new table has just been created. And this code would meet these requirements.

`FL_SET(obj, FL_EXIVAR)` is the macro that sets the `FL_EXIVAR` flag in the `basic.flags` of `obj`. The `basic.flags` flags are all named `FL_xxxx` and can be set using `FL_SET()`. These flags can be unset with `FL_UNSET()`. The `EXIVAR` from `FL_EXIVAR` seems to be the abbreviation of EXternal Instance VARiable.

This flag is set to speed up the reading of instance variables. If `FL_EXIVAR` is not set, even without searching in `generic_iv_tbl`, we can see the object does not have any instance variables. And of

course a bit check is way faster than searching a struct `st_table`.

## Gaps in structs

Now you understood the way to store the instance variables, but why are there structs without `iv_tbl`? Why is there no `iv_tbl` in struct `RString` or struct `RArray`? Couldn't `iv_tbl` be part of `RBasic`?

To tell the conclusion first, we can do such thing, but should not. As a matter of fact, this problem is deeply linked to the way ruby manages objects.

In ruby, the memory used for string data (`char[]`) and such is directly allocated using `malloc()`. However, the object structs are handled in a particular way. ruby allocates them by clusters, and then distribute them from these clusters. And in this way, if the types (or rather their sizes) were diverse, it's hard to manage, thus `RVALUE`, which is the union of the all structs, is defined and the array of the unions is managed.

The size of a union is the same as the size of the biggest member, so for instance, if one of the structs is big, a lot of space would be wasted. Therefore, it's preferable that each struct size is as similar as possible.

The most used struct might be usually struct `RString`. After that, depending on each program, there comes struct `RArray` (array), `RHash` (hash), `RObject` (user defined object), etc. However, this struct

RObject only uses the space of struct RBasic + 1 pointer. On the other hand, struct RString, RArray and RHash take the space of struct RBasic + 3 pointers. In other words, when the number of struct RObject is being increased, the memory space of the two pointers for each object are wasted. Furthermore, if the size of RString was as much as 4 pointers, RObject would use less than the half size of the union, and this is too wasteful.

So the received merit for iv\_tbl is more or less saving memory and speeding up. Furthermore we do not know if it is used often or not. In fact, generic\_iv\_tbl was not introduced before ruby 1.2, so it was not possible to use instance variables in String or Array at that time. Nevertheless, it was not much of a problem. Making large amounts of memory useless just for such functionality looks stupid.

If you take all this into consideration, you can conclude that increasing the size of object structs for iv\_tbl does not do any good.

## ■ rb\_ivar\_get()

We saw the rb\_ivar\_set() function that sets variables, so let's see quickly how to get them.

### ▼ rb\_ivar\_get()

```
960 VALUE
961 rb_ivar_get(obj, id)
962     VALUE obj;
963     ID id;
964 {
```

```
965     VALUE val;
966
967     switch (TYPE(obj)) {
968     /* (A) */
969     case T_OBJECT:
970     case T_CLASS:
971     case T_MODULE:
972     if (ROBJECT(obj)->iv_tbl &&
973         st_lookup(ROBJECT(obj)->iv_tbl, id, &val))
974         return val;
975     break;
976
977     /* (B) */
978     default:
979     if (FL_TEST(obj, FL_EXIVAR) || rb_special_const_p(obj))
980         return generic_ivar_get(obj, id);
981     break;
982     }
983
984     /* (C) */
985     rb_warning("instance variable %s not initialized", rb_id);
986
987     return Qnil;
988 }
```

(variable.c)

The structure is completely the same.

(A) For struct RObject or RClass, we search the variable in `iv_tbl`. As `iv_tbl` can also be `NULL`, we must check it before using it. Then if `st_lookup()` finds the relation, it returns true, so the whole `if` can be read as “If the instance variable has been set, return its value”.

(C) If no correspondence could be found, in other words if we read an instance variable that has not been set, we first leave the `if` then the `switch`. `rb_warning()` will then issue a warning and `nil` will be returned. That’s because you can read instance variables that have

not been set in Ruby.

(B) On the other hand, if the struct is neither `RObjet` nor `RClass`, the instance variable table is searched in `generic_iv_tbl`. What `generic_ivar_get()` does can be easily guessed, so I won't explain it. I'd rather want you to focus on the condition of the `if` statement.

I already told you that the `FL_EXIRVAR` flag is set to the object on which `generic_ivar_set()` is used. Here, that flag is utilized to make the check faster.

And what is `rb_special_const_p()`? This function returns true when its parameter `obj` does not point to a struct. As no struct means no `basic.flags`, no flag can be set in the first place. Thus `FL_xxxx()` is designed to always return false for such object. Hence, objects that are `rb_special_const_p()` should be treated specially here.

## Object Structs

---

In this section, about the important ones among object structs, we'll briefly see their concrete appearances and how to deal with them.

### ■ `struct RString`

struct `RString` is the struct for the instances of the `String` class and its subclasses.

## ▼ struct `RString`

```
314 struct RString {  
315     struct RBasic basic;  
316     long len;  
317     char *ptr;  
318     union {  
319         long capa;  
320         VALUE shared;  
321     } aux;  
322 };
```

(`ruby.h`)

`ptr` is a pointer to the string, and `len` the length of that string. Very straightforward.

Rather than a string, Ruby's string is more a byte array, and can contain any byte including `NUL`. So when thinking at the Ruby level, ending the string with `NUL` does not mean anything. But as C functions require `NUL`, for convenience the ending `NUL` is there. However, its size is not included in `len`.

When dealing with a string from the interpreter or an extension library, you can access `ptr` and `len` by writing `RSTRING(str)->ptr` or `RSTRING(str)->len`, and it is allowed. But there are some points to pay attention to.

1. you have to check if `str` really points to a struct `RString` by

yourself beforehand

2. you can read the members, but you must not modify them
3. you can't store `RSTRING(str)→ptr` in something like a local variable and use it later

Why is that? First, there is an important software engineering principle: Don't arbitrarily tamper with someone's data. When there are interface functions, we should use them. However, there are also concrete reasons in ruby's design why you should not refer to or store a pointer, and that's related to the fourth member `aux`. However, to explain properly how to use `aux`, we have to explain first a little more of Ruby's strings' characteristics.

Ruby's strings can be modified (are mutable). By mutable I mean after the following code:

```
s = "str"      # create a string and assign it to s
s.concat("ing") # append "ing" to this string object
p(s)          # show "string"
```

the content of the object pointed by `s` will become "string". It's different from Java or Python string objects. Java's `StringBuffer` is closer.

And what's the relation? First, mutable means the length (`len`) of the string can change. We have to increase or decrease the allocated memory size each time the length changes. We can of course use `realloc()` for that, but generally `malloc()` and `realloc()` are heavy operations. Having to `realloc()` each time the string

changes is a huge burden.

That's why the memory pointed by `ptr` has been allocated with a size a little bigger than `len`. Because of that, if the added part can fit into the remaining memory, it's taken care of without calling `realloc()`, so it's faster. The struct member `aux.capa` contains the length including this additional memory.

So what is this other `aux.shared`? It's to speed up the creation of literal strings. Have a look at the following Ruby program.

```
while true do  # repeat indefinitely
  a = "str"      # create a string with "str" as content and a
  a.concat("ing") # append "ing" to the object pointed by a
  p(a)          # show "string"
end
```

Whatever the number of times you repeat the loop, the fourth line's `p` has to show "string". And to do so, the expression "str" must every time create an object that holds a distinct `char[]`. But there must be also the high possibility that strings are not modified at all, and a lot of useless copies of `char[]` would be created in such situation. If possible, we'd like to share one common `char[]`.

The trick to share is `aux.shared`. Every string object created with a literal uses one shared `char[]`. And after a change occurs, the object-specific memory is allocated. When using a shared `char[]`, the flag `ELTS_SHARED` is set in the object struct's `basic.flags`, and `aux.shared` contains the original object. `ELTS` seems to be the

Then, let's return to our talk about `RSTRING(str)->ptr`. Though referring to a pointer is OK, you must not assign to it. This is first because the value of `len` or `capa` will no longer agree with the actual body, and also because when modifying strings created as literals, `aux.shared` has to be separated.

Before ending this section, I'll write some examples of dealing with `RString`. I'd like you to regard `str` as a `VALUE` that points to `RString` when reading this.

```
RSTRING(str)->len;          /* length */
RSTRING(str)->ptr[0];        /* first character */
str = rb_str_new("content", 7); /* create a string with "conter
                                the second parameter is the
                                its length is calculated wit
                                /* Concatenate a C string to a
```

## ■ struct RArray

`struct RArray` is the struct for the instances of Ruby's array class `Array`.

### ▼ struct RArray

```
324 struct RArray {
325     struct RBasic basic;
326     long len;
327     union {
328         long capa;
329         VALUE shared;
```

```
330     } aux;
331     VALUE *ptr;
332 }

(ruby.h)
```

Except for the type of `ptr`, this structure is almost the same as `struct RString`. `ptr` points to the content of the array, and `len` is its length. `aux` is exactly the same as in `struct RString`. `aux.capa` is the “real” length of the memory pointed by `ptr`, and if `ptr` is shared, `aux.shared` stores the shared original array object.

From this structure, it’s clear that Ruby’s `Array` is an array and not a list. So when the number of elements changes in a big way, a `realloc()` must be done, and if an element must be inserted at an other place than the end, a `memmove()` will occur. But even if it does it, it’s moving so fast that we don’t notice about that. Recent machines are really impressive.

And the way to access to its members is similar to the way of `RString`. With `RARRAY(arr)->ptr` and `RARRAY(arr)->len`, you can refer to the members, and it is allowed, but you must not assign to them, etc. We’ll only look at simple examples:

```
/* manage an array from C */
VALUE ary;
ary = rb_ary_new();           /* create an empty array */
rb_ary_push(ary, INT2FIX(9)); /* push a Ruby 9 */
RARRAY(ary)->ptr[0];        /* look what's at index 0 */
rb_p(RARRAY(ary)->ptr[0]);  /* do p on ary[0] (the result is

# manage an array from Ruby
```

```
ary = []      # create an empty array
ary.push(9)   # push 9
ary[0]        # look what's at index 0
p(ary[0])    # do p on ary[0] (the result is 9)
```

## struct RRegexp

It's the struct for the instances of the regular expression class `Regexp`.

### ▼ struct RRegexp

```
334 struct RRegexp {
335     struct RBasic basic;
336     struct re_pattern_buffer *ptr;
337     long len;
338     char *str;
339 };
```

(`ruby.h`)

`ptr` is the compiled regular expression. `str` is the string before compilation (the source code of the regular expression), and `len` is this string's length.

As any code to handle `Regexp` objects doesn't appear in this book, we won't see how to use it. Even if you use it in extension libraries, as long as you do not want to use it a very particular way, the interface functions are enough.

## struct RHash

struct RHash is the struct for Hash object, which is Ruby's hash table.

## ▼ struct RHash

```
341 struct RHash {  
342     struct RBasic basic;  
343     struct st_table *tbl;  
344     int iter_lev;  
345     VALUE ifnone;  
346 };
```

(ruby.h)

It's a wrapper for struct st\_table. st\_table will be detailed in the next chapter “Names and name tables”.

ifnone is the value when a key does not have an associated value, its default is nil. iter\_lev is to make the hashtable reentrant (multithread safe).

## ■ struct RFile

struct RFile is a struct for instances of the built-in IO class and its subclasses.

## ▼ struct RFile

```
348 struct RFile {  
349     struct RBasic basic;  
350     struct OpenFile *fptr;  
351 };
```

(ruby.h)

## ▼ OpenFile

```
19  typedef struct OpenFile {  
20      FILE *f;                      /* stdio ptr for read/write  
21      FILE *f2;                     /* additional ptr for rw pip  
22      int mode;                    /* mode flags */  
23      int pid;                     /* child's pid (for pipes) */  
24      int lineno;                  /* number of lines read */  
25      char *path;                  /* pathname for file */  
26      void (*finalize) _((struct OpenFile*)); /* finalize proc  
27  } OpenFile;
```

(rubyio.h)

All members have been transferred in `struct OpenFile`. As there aren't many instances of `IO` objects, it's OK to do it like this. The purpose of each member is written in the comments. Basically, it's a wrapper around C's `stdio`.

## ■ struct RData

`struct RData` has a different tenor from what we saw before. It is the struct for implementation of extension libraries.

Of course structs for classes created in extension libraries are necessary, but as the types of these structs depend on the created class, it's impossible to know their size or struct in advance. That's why a "struct for managing a pointer to a user defined struct" has been created on `ruby`'s side to manage this. This struct is `struct RData`.

## ▼ struct RData

```
353 struct RData {  
354     struct RBasic basic;  
355     void (*dmark) _((void*));  
356     void (*dfree) _((void*));  
357     void *data;  
358 };
```

(ruby.h)

data is a pointer to the user defined struct, dfree is the function used to free that user defined struct, and dmark is the function to do “mark” of the mark and sweep.

Because explaining struct RData is still too complicated, for the time being let's just look at its representation (figure 8). The detailed explanation of its members will be introduced after we'll finish chapter 5 “Garbage collection”.

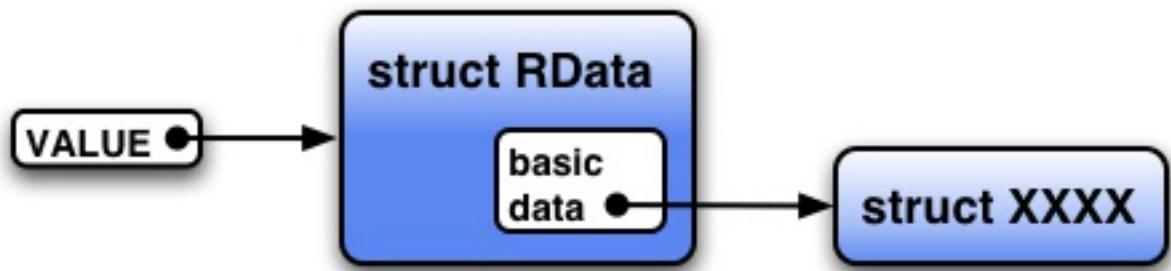


Figure 8: Representation of struct RData

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

# License

# Ruby Hacking Guide

Translated by Clifford Escobar CAOILE

# Chapter 3: Names and Name Table

## st\_table

---

`st_table` has already appeared several times as a method table and an instance table. In this chapter let's look at the structure of the `st_table` in detail.

### ■ Summary

I previously mentioned that the `st_table` is a hash table. What is a hash table? It is a data structure that records one-to-one relations, for example, a variable name and its value, or a function name and its body, etc.

However, data structures other than hash tables can, of course, record one-to-one relations. For example, a list of the following structs will suffice for this purpose.

```
struct entry {  
    ID key;  
    VALUE val;
```

```
    struct entry *next; /* point to the next entry */  
};
```

However, this method is slow. If the list contains a thousand items, in the worst case, it is necessary to traverse a thousand links. In other words, the search time increases in proportion to the number of elements. This is bad. Since ancient times, various speed improvement methods have been conceived. The hash table is one of those improved methods. In other words, the point is not that the hash table is necessary but that it can be made faster.

Now then, let us examine the `st_table`. As it turns out, this library is not created by Matsumoto, rather:

### ▼ `st.c` credits

```
1 /* This is a public domain general purpose hash table package  
   written by Peter Moore @ UCB. */
```

(`st.c`)

as shown above.

By the way, when I searched Google and found another version, it mentioned that `st_table` is a contraction of “STring TABLE”. However, I find it contradictory that it has both “general purpose” and “string” aspects.

## What is a hash table?

A hash table can be thought as the following: Let us think of an array with  $n$  items. For example, let us make  $n=64$  (figure 1).



Figure 1: Array

Then let us specify a function  $f$  that takes a key and produces an integer  $i$  from 0 to  $n-1$  (0-63). We call this  $f$  a hash function.  $f$  when given the same key always produces the same  $i$ . For example, if we can assume that the key is limited to positive integers, when the key is divided by 64, the remainder should always fall between 0 and 63. Therefore, this calculating expression has a possibility of being the function  $f$ .

When recording relationships, given a key, function  $f$  generates  $i$ , and places the value into index  $i$  of the array we have prepared. Index access into an array is very fast. The key concern is changing a key into an integer.



Figure 2: Array assignment

However, in the real world it isn't that easy. There is a critical problem with this idea. Because  $n$  is only 64, if there are more than 64 relationships to be recorded, it is certain that there will be the same index for two different keys. It is also possible that with fewer than 64, the same thing can occur. For example, given the previous hash function "key % 64", keys 65 and 129 will both have a hash value of 1. This is called a hash value collision. There are many ways to resolve such a collision.

One solution is to insert into the next element when a collision occurs. This is called open addressing. (Figure 3).

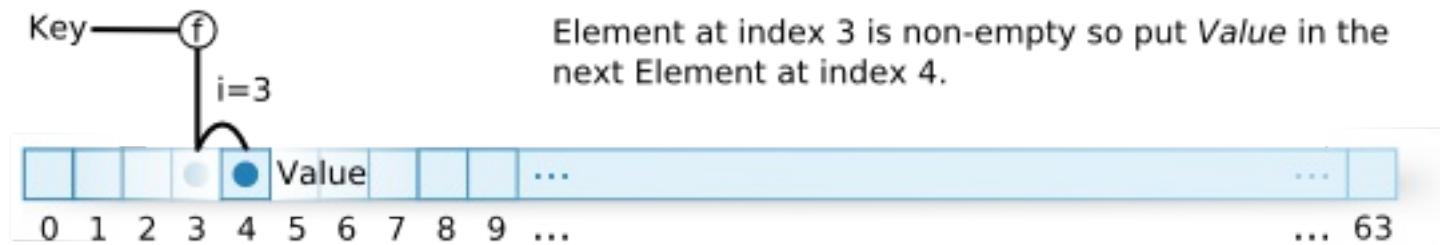


Figure 3: Open addressing

Other than using the array like this, there are other possible approaches, like using a pointer to a respective linked list in each element of the array. Then when a collision occurs, grow the linked list. This is called chaining. (Figure 4) `st_table` uses this chaining method.

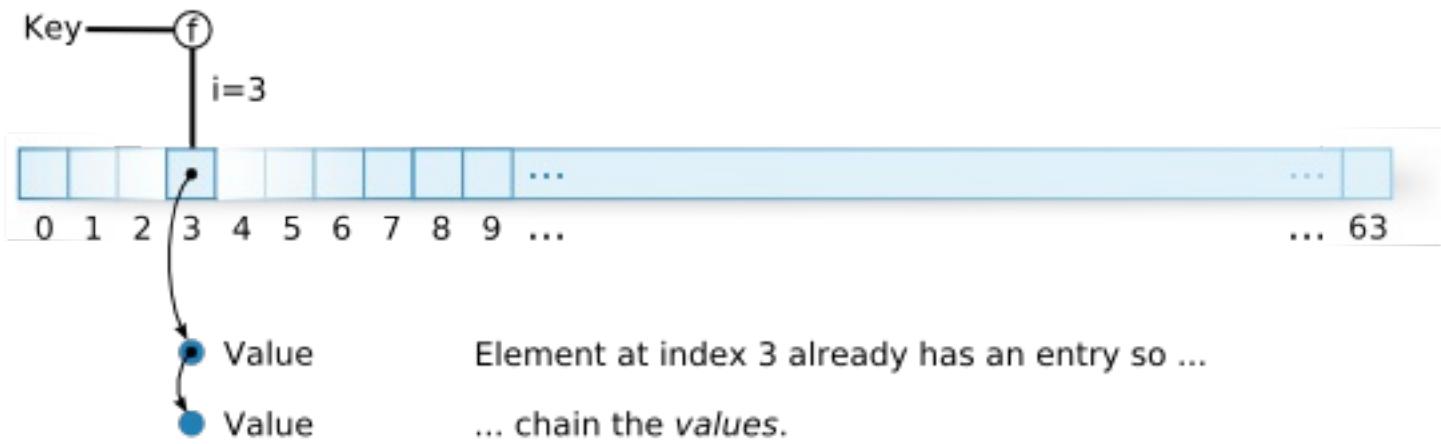


Figure 4: Chaining

However, if it can be determined a priori what set of keys will be used, it is possible to imagine a hash function that will never create collisions. This type of function is called a “perfect hash function”. Actually, there are tools which create a perfect hash function given a set of arbitrary strings. GNU gperf is one of those. ruby’s parser implementation uses GNU gperf but... this is not the time to discuss it. We’ll discuss this in the second part of the book.

## Data Structure

Let us start looking at the source code. As written in the introductory chapter, if there is data and code, it is better to read the data first. The following is the data type of `st_table`.

### ▼ `st_table`

```

9  typedef struct st_table st_table;
16 struct st_table {
17     struct st_hash_type *type;
18     int num_bins;                      /* slot count */

```

```
19     int num_entries;           /* total number of entries
20     struct st_table_entry **bins; /* slot */
21 };
```

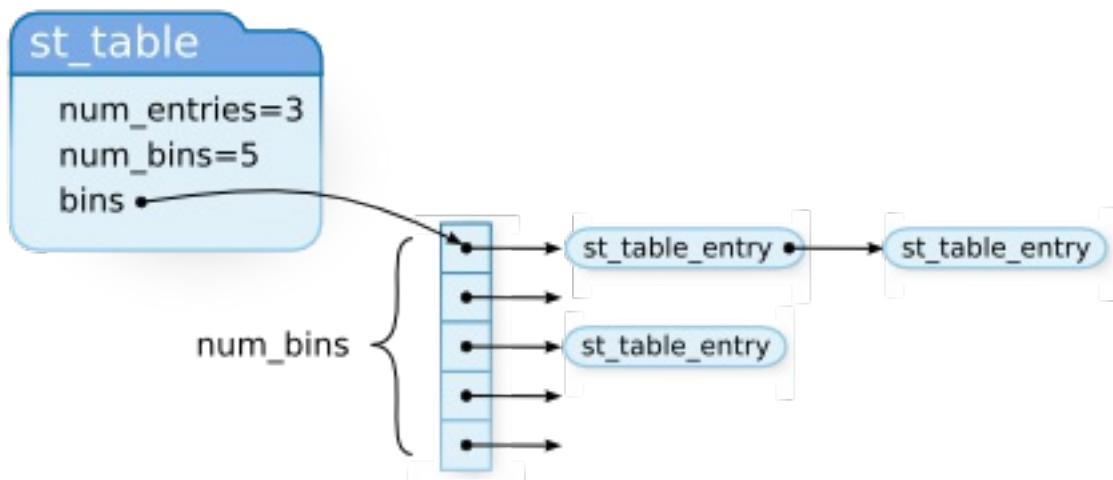
(st.h)

## ▼ struct st\_table\_entry

```
16 struct st_table_entry {
17     unsigned int hash;
18     char *key;
19     char *record;
20     st_table_entry *next;
21 };
```

(st.c)

st\_table is the main table structure. st\_table\_entry is a holder that stores one value. st\_table\_entry contains a member called next which of course is used to make st\_table\_entry into a linked list. This is the chain part of the chaining method. The st\_hash\_type data type is used, but I will explain this later. First let me explain the other parts so you can compare and understand the roles.



## Figure 5: st\_table data structure

So, let us comment on `st_hash_type`.

### ▼ `struct st_hash_type`

```
11 struct st_hash_type {  
12     int (*compare)(); /* comparison function */  
13     int (*hash)(); /* hash function */  
14 };
```

`(st.h)`

This is still Chapter 3 so let us examine it attentively.

```
int (*compare)()
```

This part shows, of course, the member `compare` which has a data type of “a pointer to a function that returns an `int`”. `hash` is also of the same type. This variable is substituted in the following way:

```
int  
great_function(int n)  
{  
    /* ToDo: Do something great! */  
    return n;  
}  
  
{  
    int (*f)();  
    f = great_function;
```

And it is called like this:

```
(*f)(7);  
}
```

Here let us return to the `st_hash_type` commentary. Of the two members `hash` and `compare`, `hash` is the hash function `f` explained previously.

On the other hand, `compare` is a function that evaluates if the key is actually the same or not. With the chaining method, in the spot with the same hash value `n`, multiple elements can be inserted. To know exactly which element is being searched for, this time it is necessary to use a comparison function that we can absolutely trust. `compare` will be that function.

This `st_hash_type` is a good generalized technique. The hash table itself cannot determine what the stored keys' data type will be. For example, in `ruby`, `st_table`'s keys are `ID` or `char*` or `VALUE`, but to write the same kind of hash for each (data type) is foolish. Usually, the things that change with the different key data types are things like the hash function. For things like memory allocation and collision detection, typically most of the code is the same. Only the parts where the implementation changes with a differing data type will be bundled up into a function, and a pointer to that function will be used. In this fashion, the majority of the code that makes up the hash table implementation can use it.

In object-oriented languages, in the first place, you can attach a procedure to an object and pass it (around), so this mechanism is

not necessary. Perhaps it more correct to say that this mechanism is built-in as a language's feature.

## ■ **st\_hash\_type example**

The usage of a data structure like `st_hash_type` is good as an abstraction. On the other hand, what kind of code it actually passes through may be difficult to understand. If we do not examine what sort of function is used for `hash` or `compare`, we will not grasp the reality. To understand this, it is probably sufficient to look at `st_init_numtable()` introduced in the previous chapter. This function creates a table for integer data type keys.

### ▼ `st_init_numtable()`

```
182 st_table*  
183 st_init_numtable()  
184 {  
185     return st_init_table(&type_numhash);  
186 }
```

(st.c)

`st_init_table()` is the function that allocates the table memory and so on. `type_numhash` is an `st_hash_type` (it is the member named “type” of `st_table`). Regarding this `type_numhash`:

### ▼ `type_numhash`

```
37 static struct st_hash_type type_numhash = {  
38     numcmp,
```

```
39     numhash,
40 };

552 static int
553 numcmp(x, y)
554     long x, y;
555 {
556     return x != y;
557 }

559 static int
560 numhash(n)
561     long n;
562 {
563     return n;
564 }
```

(st.c)

Very simple. The table that the ruby interpreter uses is by and large this `type_numhash`.

## ■ `st_lookup()`

Now then, let us look at the function that uses this data structure. First, it's a good idea to look at the function that does the searching. Shown below is the function that searches the hash table, `st_lookup()`.

## ▼ `st_lookup()`

```
247 int
248 st_lookup(table, key, value)
249     st_table *table;
250     register char *key;
251     char **value;
```

```
252  {
253      unsigned int hash_val, bin_pos;
254      register st_table_entry *ptr;
255
256      hash_val = do_hash(key, table);
257      FIND_ENTRY(table, ptr, hash_val, bin_pos);
258
259      if (ptr == 0) {
260          return 0;
261      }
262      else {
263          if (value != 0)  *value = ptr->record;
264          return 1;
265      }
266 }
```

(st.c)

The important parts are pretty much in `do_hash()` and `FIND_ENTRY()`. Let us look at them in order.

## ▼ `do_hash()`

```
68 #define do_hash(key,table) (unsigned int)(*(table)->type->ha
```

(st.c)

Just in case, let us write down the macro body that is difficult to understand:

`(table)->type->hash`

is a function pointer where the `key` is passed as a parameter. This is the syntax for calling the function. `*` is not applied to `table`. In other words, this macro is a hash value generator for a key, using the

prepared hash function type->hash for each data type.

Next, let us examine FIND\_ENTRY().

## ▼ FIND\_ENTRY()

```
235 #define FIND_ENTRY(table, ptr, hash_val, bin_pos) do {\n236     bin_pos = hash_val%(table)->num_bins;\n237     ptr = (table)->bins[bin_pos];\n238     if (PTR_NOT_EQUAL(table, ptr, hash_val, key)) {\n239         COLLISION;\n240         while (PTR_NOT_EQUAL(table, ptr->next, hash_val, key)) {\n241             ptr = ptr->next;\n242         }\n243         ptr = ptr->next;\n244     }\n245 } while (0)\n\n227 #define PTR_NOT_EQUAL(table, ptr, hash_val, key) ((ptr) != 0\n228             (ptr->hash != (hash_val) || !EQUAL((table), (key), (ptr))\n\n66 #define EQUAL(table,x,y) \
229             ((x)==(y) || (*table->type->compare)((x),(y)) == 0)\n\n(st.c)
```

COLLISION is a debug macro so we will (should) ignore it.

The parameters of FIND\_ENTRY(), starting from the left are:

1. st\_table
2. the found entry will be pointed to by this parameter
3. hash value
4. temporary variable

And, the second parameter will point to the found `st_table_entry*`.

At the outermost level, a `do .. while(0)` is used to safely wrap up a multiple expression macro. This is `ruby`'s, or rather, C language's preprocessor idiom. In the case of `if(1)`, there may be a danger of adding an `else` part. In the case of `while(1)`, it becomes necessary to add a `break` at the very end.

Also, there is no semicolon added after the `while(0)`.

```
FIND_ENTRY();
```

This is so that the semicolon that is normally written at the end of an expression will not go to waste.

## ■ `st_add_direct()`

Continuing on, let us examine `st_add_direct()` which is a function that adds a new relationship to the hash table. This function does not check if the key is already registered. It always adds a new entry. This is the meaning of `direct` in the function name.

## ▼ `st_add_direct()`

```
308 void
309 st_add_direct(table, key, value)
310     st_table *table;
311     char *key;
312     char *value;
313 {
314     unsigned int hash_val, bin_pos;
```

```
315     hash_val = do_hash(key, table);
316     bin_pos = hash_val % table->num_bins;
317     ADD_DIRECT(table, key, value, hash_val, bin_pos);
318 }
319 }
```

(st.c)

Just as before, the `do_hash()` macro that obtains a value is called here. After that, the next calculation is the same as at the start of `FIND_ENTRY()`, which is to exchange the hash value for a real index.

Then the insertion operation seems to be implemented by `ADD_DIRECT()`. Since the name is all uppercase, we can anticipate that is a macro.

## ▼ ADD\_DIRECT()

```
268 #define ADD_DIRECT(table, key, value, hash_val, bin_pos) \
269 do { \
270     st_table_entry *entry; \
271     if (table->num_entries / (table->num_bins) \
272         > ST_DEFAULT_MAX_DENSITY) { \
273         rehash(table); \
274         bin_pos = hash_val % table->num_bins; \
275     } \
276     /* (A) */ \
277     entry = alloc(st_table_entry); \
278     entry->hash = hash_val; \
279     entry->key = key; \
280     entry->record = value; \
281     /* (B) */ \
282     entry->next = table->bins[bin_pos]; \
283     table->bins[bin_pos] = entry; \
284     table->num_entries++; \
285 }
```

```
284 } while (0)
```

```
(st.c)
```

The first `if` is an exception case so I will explain it afterwards.

(A) Allocate and initialize a `st_table_entry`.

(B) Insert the `entry` into the start of the list. This is the idiom for handling the list. In other words,

```
entry->next = list_beg;  
list_beg = entry;
```

makes it possible to insert an entry to the front of the list. This is similar to “cons-ing” in the Lisp language. Check for yourself that even if `list_beg` is `NULL`, this code holds true.

Now, let me explain the code I left aside.

## ▼ ADD\_DIRECT() -rehash

```
271     if (table->num_entries / (table->num_bins)  
272         > ST_DEFAULT_MAX_DENSITY) {  
273         rehash(table);  
274         bin_pos = hash_val % table->num_bins;  
275     }
```

```
(st.c)
```

`DENSITY` is “concentration”. In other words, this conditional checks if the hash table is “crowded” or not. In the `st_table`, as the number

of values that use the same `bin_pos` increases, the longer the link list becomes. In other words, search becomes slower. That is why for a given `bin` count, when the average elements per bin become too many, `bin` is increased and the crowding is reduced.

The current `ST_DEFAULT_MAX_DENSITY` is

### ▼ `ST_DEFAULT_MAX_DENSITY`

```
23 #define ST_DEFAULT_MAX_DENSITY 5  
(st.c)
```

Because of this setting, if in all `bin_pos` there are 5 `st_table_entries`, then the size will be increased.

## ■ `st_insert()`

`st_insert()` is nothing more than a combination of `st_add_direct()` and `st_lookup()`, so if you understand those two, this will be easy.

### ▼ `st_insert()`

```
286 int  
287 st_insert(table, key, value)  
288     register st_table *table;  
289     register char *key;  
290     char *value;  
291 {  
292     unsigned int hash_val, bin_pos;  
293     register st_table_entry *ptr;  
294  
295     hash_val = do_hash(key, table);
```

```
296     FIND_ENTRY(table, ptr, hash_val, bin_pos);  
297  
298     if (ptr == 0) {  
299         ADD_DIRECT(table, key, value, hash_val, bin_pos);  
300         return 0;  
301     }  
302     else {  
303         ptr->record = value;  
304         return 1;  
305     }  
306 }
```

(st.c)

It checks if the element is already registered in the table. Only when it is not registered will it be added. If there is a insertion, return 0. If there is no insertion, return a 1.

## ID and Symbols

I've already discussed what an ID is. It is a correspondence between an arbitrary string of characters and a value. It is used to declare various names. The actual data type is `unsigned int`.

### From `char*` to ID

The conversion from string to ID is executed by `rb_intern()`. This function is rather long, so let's omit the middle.

#### ▼ `rb_intern()` (simplified)

```

5451 static st_table *sym_tbl;          /* char* to ID */
5452 static st_table *sym_rev_tbl;      /* ID to char* */

5469 ID
5470 rb_intern(name)
5471     const char *name;
5472 {
5473     const char *m = name;
5474     ID id;
5475     int last;
5476
5477     /* If for a name, there is a corresponding ID that is already
5478      registered, then return that ID */
5479     if (st_lookup(sym_tbl, name, &id))
5480         return id;

5481     /* omitted ... create a new ID */

5482     /* register the name and ID relation */
5483     id_register:
5484         name = strdup(name);
5485         st_add_direct(sym_tbl, name, id);
5486         st_add_direct(sym_rev_tbl, id, name);
5487         return id;
5488     }

(parse.y)

```

The string and ID correspondence relationship can be accomplished by using the `st_table`. There probably isn't any especially difficult part here.

What is the omitted section doing? It is treating global variable names and instance variables names as special and flagging them. This is because in the parser, it is necessary to know the variable's classification from the ID. However, the fundamental part of ID is unrelated to this, so I won't explain it here.

# From ID to char\*

The reverse of `rb_intern()` is `rb_id2name()`, which takes an ID and generates a `char*`. You probably know this, but the 2 in `id2name` is “to”. “To” and “two” have the same pronunciation, so “2” is used for “to”. This syntax is often seen.

This function also sets the ID classification flags so it is long. Let me simplify it.

## ▼ `rb_id2name()` (simplified)

```
char *
rb_id2name(id)
    ID id;
{
    char *name;

    if (st_lookup(sym_rev_tbl, id, &name))
        return name;
    return 0;
}
```

Maybe it seems that it is a little over-simplified, but in reality if we remove the details it really becomes this simple.

The point I want to emphasize is that the found `name` is not copied. The ruby API does not require (or rather, it forbids) the `free()`-ing of the return value. Also, when parameters are passed, it always copies them. In other words, the creation and release is completed by one side, either by the user or by ruby.

So then, when creation and release cannot be accomplished (when passed it is not returned) on a value, then a Ruby object is used. I have not yet discussed it, but a Ruby object is automatically released when it is no longer needed, even if we are not taking care of the object.

## Converting VALUE and ID

ID is shown as an instance of the `Symbol` class at the Ruby level. And it can be obtained like so: `"string".intern`. The implementation of `String#intern` is `rb_str_intern()`.

### ▼ `rb_str_intern()`

```
2996 static VALUE
2997 rb_str_intern(VALUE str)
2998 {
2999     ID id;
3000
3001     if (!RSTRING(str)->ptr || RSTRING(str)->len == 0) {
3002         rb_raise(rb_eArgError, "interning empty string");
3003     }
3004     if (strlen(RSTRING(str)->ptr) != RSTRING(str)->len)
3005         rb_raise(rb_eArgError, "string contains '\\0'");
3006     id = rb_intern(RSTRING(str)->ptr);
3007     return ID2SYM(id);
3008 }
3009 }
```

(string.c)

This function is quite reasonable as a ruby class library code example. Please pay attention to the part where `RSTRING()` is used

and casted, and where the data structure's member is accessed.

Let's read the code. First, `rb_raise()` is merely error handling so we ignore it for now. The `rb_intern()` we previously examined is here, and also `ID2SYM` is here. `ID2SYM()` is a macro that converts `ID` to `Symbol`.

And the reverse operation is accomplished using `Symbol#to_s` and such. The implementation is in `sym_to_s`.

### ▼ `sym_to_s()`

```
522 static VALUE
523 sym_to_s(sym)
524     VALUE sym;
525 {
526     return rb_str_new2(rb_id2name(SYM2ID(sym)));
527 }
```

`(object.c)`

`SYM2ID()` is the macro that converts `Symbol` (`VALUE`) to an `ID`.

It looks like the function is not doing anything unreasonable. However, it is probably necessary to pay attention to the area around the memory handling. `rb_id2name()` returns a `char*` that must not be `free()`. `rb_str_new2()` copies the parameter's `char*` and uses the copy (and does not change the parameter). In this way the policy is consistent, which allows the line to be written just by chaining the functions.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

Translated by Vincent ISAMBART

# Chapter 4: Classes and modules

In this chapter, we'll see the details of the data structures created by classes and modules.

## Classes and methods definition

---

First, I'd like to have a look at how Ruby classes are defined at the C level. This chapter investigates almost only particular cases, so I'd like you to know first the way used most often.

The main API to define classes and modules consists of the following 6 functions:

- `rb_define_class()`
- `rb_define_class_under()`
- `rb_define_module()`
- `rb_define_module_under()`
- `rb_define_method()`
- `rb_define_singleton_method()`

There are a few other versions of these functions, but the extension libraries and even most of the core library is defined using just this API. I'll introduce to you these functions one by one.

## Class definition

`rb_define_class()` defines a class at the top-level. Let's take the Ruby array class, `Array`, as an example.

### ▼ Array class definition

```
19  VALUE rb_cArray;  
  
1809 void  
1810 Init_Array()  
1811 {  
1812     rb_cArray = rb_define_class("Array", rb_cObject);  
  
(array.c)
```

`rb_cObject` and `rb_cArray` correspond respectively to `Object` and `Array` at the Ruby level. The added prefix `rb` shows that it belongs to `ruby` and the `c` that it is a class object. These naming rules are used everywhere in `ruby`.

This call to `rb_define_class()` defines a class called `Array`, which inherits from `Object`. At the same time as `rb_define_class()` creates the class object, it also defines the constant. That means that after this you can already access `Array` from a Ruby program. It corresponds to the following Ruby program:

```
class Array < Object
```

I'd like you to note the fact that there is no end. It was written like this on purpose. It is because with `rb_define_class()` the body of the class has not been executed.

## ▀ Nested class definition

After that, there's `rb_define_class_under()`. This function defines a class nested in an other class or module. This time the example is what is returned by `stat(2)`, `File::Stat`.

### ▼ Definition of `File::Stat`

```
78 VALUE rb_cFile;
80 static VALUE rb_cStat;

2581     rb_cFile = rb_define_class("File", rb_cIO);
2674     rb_cStat = rb_define_class_under(rb_cFile, "Stat", rb_c0
(file.c)
```

This code corresponds to the following Ruby program;

```
class File < IO
  class Stat < Object
```

This time again I omitted the `end` on purpose.

## ▀ Module definition

`rb_define_module()` is simple so let's end this quickly.

## ▼ Definition of Enumerable

```
17 VALUE rb_mEnumerable;  
492     rb_mEnumerable = rb_define_module("Enumerable");  
(enum.c)
```

The `m` in the beginning of `rb_mEnumerable` is similar to the `c` for classes: it shows that it is a module. The corresponding Ruby program is:

```
module Enumerable
```

`rb_define_module_under()` is not used much so we'll skip it.

## Method definition

This time the function is the one for defining methods, `rb_define_method()`. It's used very often. We'll take once again an example from `Array`.

## ▼ Definition of `Array#to_s`

```
1818 rb_define_method(rb_cArray, "to_s", rb_ary_to_s, 0);  
(array.c)
```

With this the `to_s` method is defined in `Array`. The method body is

given by a function pointer (`rb_ary_to_s`). The fourth parameter is the number of parameters taken by the method. As `to_s` does not take any parameters, it's 0. If we write the corresponding Ruby program, we'll have this:

```
class Array < Object
  def to_s
    # content of rb_ary_to_s()
  end
end
```

Of course the `class` part is not included in `rb_define_method()` and only the `def` part is accurate. But if there is no `class` part, it will look like the method is defined like a function, so I also wrote the enclosing `class` part.

One more example, this time taking a parameter:

## ▼ Definition of `Array#concat`

```
1835 rb_define_method(rb_cArray, "concat", rb_ary_concat, 1);
(array.c)
```

The class for the definition is `rb_cArray` (`Array`), the method name is `concat`, its body is `rb_ary_concat()` and the number of parameters is 1. It corresponds to writing the corresponding Ruby program:

```
class Array < Object
  def concat( str )
    # content of rb_ary_concat()
  end
```

end

# Singleton methods definition

We can define methods that are specific to a single object instance. They are called singleton methods. As I used `File.unlink` as an example in chapter 1 “Ruby language minimum”, I first wanted to show it here, but for a particular reason we’ll look at `File.link` instead.

## ▼ Definition of `File.link`

```
2624 rb_define_singleton_method(rb_cFile, "link", rb_file_s_link,  
(file.c)
```

It’s used like `rb_define_method()`. The only difference is that here the first parameter is just the “object” where the method is defined. In this case, it’s defined in `rb_cFile`.

# Entry point

Being able to make definitions like before is great, but where are these functions called from, and by what means are they executed? These definitions are grouped in functions named `Init_xxxx()`. For instance, for `Array` a function `Init_Array()` like this has been made:

## ▼ `Init_Array`

```
1809 void
1810 Init_Array()
1811 {
1812     rb_cArray = rb_define_class("Array", rb_cObject);
1813     rb_include_module(rb_cArray, rb_mEnumerable);
1814
1815     rb_define_singleton_method(rb_cArray, "allocate",
1816                               rb_ary_s_alloc, 0);
1816     rb_define_singleton_method(rb_cArray, "[", rb_ary_s_cre
1817     rb_define_method(rb_cArray, "initialize", rb_ary_initialize,
1818     rb_define_method(rb_cArray, "to_s", rb_ary_to_s, 0);
1819     rb_define_method(rb_cArray, "inspect", rb_ary_inspect, 0
1820     rb_define_method(rb_cArray, "to_a", rb_ary_to_a, 0);
1821     rb_define_method(rb_cArray, "to_ary", rb_ary_to_a, 0);
1822     rb_define_method(rb_cArray, "frozen?", rb_ary_frozen_p,
```

(array.c)

The `Init` for the built-in functions are explicitly called during the startup of ruby. This is done in `inits.c`.

## ▼ `rb_call_inits()`

```
47 void
48 rb_call_inits()
49 {
50     Init_sym();
51     Init_var_tables();
52     Init_Object();
53     Init_Comparable();
54     Init_Enumerable();
55     Init_Precision();
56     Init_eval();
57     Init_String();
58     Init_Exception();
59     Init_Thread();
60     Init_Numeric();
61     Init_Bignum();
62     Init_Array();
```

This way, `Init_Array()` is called properly.

That explains it for the built-in libraries, but what about extension libraries? In fact, for extension libraries the convention is the same. Take the following code:

```
require "myextension"
```

With this, if the loaded extension library is `myextension.so`, at load time, the (`extern`) function named `Init_myextension()` is called. How they are called is beyond the scope of this chapter. For that, you should read chapter 18, “Load”. Here we’ll just end this with an example of `Init`.

The following example is from `stringio`, an extension library provided with `ruby`, that is to say not from a built-in library.

## ▼ `Init_stringio()` (beginning)

```
895 void
896 Init_stringio()
897 {
898     VALUE StringIO = rb_define_class("StringIO", rb_cData);
899     rb_define_singleton_method(StringIO, "allocate",
900                               stro_s_allocate, 0);
901     rb_define_singleton_method(StringIO, "open", stro_s_ope
902     rb_define_method(StringIO, "initialize", stro_initialize
903     rb_enable_super(StringIO, "initialize");
904     rb_define_method(StringIO, "become", stro_become, 1);
905     rb_define_method(StringIO, "reopen", stro_reopen, -1);
```

# Singleton classes

## rb\_define\_singleton\_method()

You should now be able to more or less understand how normal methods are defined. Somehow making the body of the method, then registering it in `m_tbl` will do. But what about singleton methods? We'll now look into the way singleton methods are defined.

### ▼ rb\_define\_singleton\_method()

```
721 void
722 rb_define_singleton_method(obj, name, func, argc)
723     VALUE obj;
724     const char *name;
725     VALUE (*func)();
726     int argc;
727 {
728     rb_define_method(rb_singleton_class(obj), name, func, ar
729 }
```

(class.c)

As I explained, `rb_define_method()` is a function used to define normal methods, so the difference from normal methods is only `rb_singleton_class()`. But what on earth are singleton classes?

In brief, singleton classes are virtual classes that are only used to execute singleton methods. Singleton methods are functions defined in singleton classes. Classes themselves are in the first place (in a way) the “implementation” to link objects and methods, but singleton classes are even more on the implementation side. In the Ruby language way, they are not formally included, and don’t appear much at the Ruby level.

## ■ `rb_singleton_class()`

Well, let’s confirm what the singleton classes are made of. It’s too simple to just show you the code of a function each time so this time I’ll use a new weapon, a call graph.

```
rb_define_singleton_method
  rb_define_method
    rb_singleton_class
      SPECIAL_SINGLETON
        rb_make_metaclass
          rb_class_boot
            rb_singleton_class_attached
```

Call graphs are graphs showing calling relationships among functions (or more generally procedures). The call graphs showing all the calls written in the source code are called static call graphs. The ones expressing only the calls done during an execution are called dynamic call graphs.

This diagram is a static call graph and the indentation expresses which function calls which one. For instance,

`rb_define_singleton_method()` calls `rb_define_method()` and

`rb_singleton_class()`. And this `rb_singleton_class()` itself calls `SPECIAL_SINGLETON()` and `rb_make_metaclass()`. In order to obtain call graphs, you can use `cflow` and such. {`cflow`: see also `doc/callgraph.html` in the attached CD-ROM}

In this book, because I wanted to obtain call graphs that contain only functions, I created a `ruby`-specific tool by myself. Perhaps it can be generalized by modifying its code analyzing part, thus I'd like to somehow make it until around the publication of this book. These situations are also explained in `doc/callgraph.html` of the attached CD-ROM.

Let's go back to the code. When looking at the call graph, you can see that the calls made by `rb_singleton_class()` go very deep. Until now all call levels were shallow, so we could simply look at the functions without getting too lost. But at this depth, I easily forget what I was doing. In such situation you must bring a call graph to keep aware of where it is when reading. This time, as an example, we'll decode the procedures below `rb_singleton_class()` in parallel. We should look out for the following two points:

- What exactly are singleton classes?
- What is the purpose of singleton classes?

## ■ **Normal classes and singleton classes**

Singleton classes are special classes: they're basically the same as normal classes, but there are a few differences. We can say that

finding these differences is explaining concretely singleton classes.

What should we do to find them? We should find the differences between the function creating normal classes and the one creating singleton classes. For this, we have to find the function for creating normal classes. That is as normal classes can be defined by `rb_define_class()`, it must call in a way or another a function to create normal classes. For the moment, we'll not look at the content of `rb_define_class()` itself. I have some reasons to be interested in something that's deeper. That's why we will first look at the call graph of `rb_define_class()`.

```
rb_define_class
  rb_class_inherited
  rb_define_class_id
    rb_class_new
      rb_class_boot
    rb_make_metaclass
      rb_class_boot
    rb_singleton_class_attached
```

I'm interested by `rb_class_new()`. Doesn't this name means it creates a new class? Let's confirm that.

## ▼ `rb_class_new()`

```
37  VALUE
38  rb_class_new(super)
39    VALUE super;
40  {
41    Check_Type(super, T_CLASS);
42    if (super == rb_cClass) {
43      rb_raise(rb_eTypeError, "can't make subclass of Clas
```

```
44      }
45      if (FL_TEST(super, FL_SINGLETON)) {
46          rb_raise(rb_eTypeError, "can't make subclass of virt
47      }
48      return rb_class_boot(super);
49  }
```

(class.c)

Check\_Type() is checks the type of object structure, so we can ignore it. rb\_raise() is error handling so we can ignore it. Only rb\_class\_boot() remains. So let's look at it.

## ▼ rb\_class\_boot()

```
21  VALUE
22  rb_class_boot(super)
23      VALUE super;
24  {
25      NEWOBJ(klass, struct RClass);      /* allocates struct
26      OBJSETUP(klass, rb_cClass, T_CLASS); /* initialization o
27
28      klass->super = super;           /* (A) */
29      klass->iv_tbl = 0;
30      klass->m_tbl = 0;
31      klass->m_tbl = st_init_numtable();
32
33      OBJ_INFECT(klass, super);
34      return (VALUE)klass;
35  }
```

(class.c)

NEWHOBJ() and OBJSETUP() are fixed expressions used when creating Ruby objects that possess one of the built-in structure types (struct Rxxxx). They are both macros. In NEWOBJ(), struct RClass is created

and the pointer is put in its first parameter `klass`. In `OBJSETUP()`, the struct `RBasic` member of the `RClass` (and thus `basic.klass` and `basic.flags`) is initialized.

`OBJ_INFECT()` is a macro related to security. From now on, we'll ignore it.

At (A), the `super` member of `klass` is set to the `super` parameter. It looks like `rb_class_boot()` is a function that creates a class inheriting from `super`.

So, as `rb_class_boot()` is a function that creates a class, and `rb_class_new()` is almost identical.

Then, let's once more look at `rb_singleton_class()`'s call graph:

```
rb_singleton_class
  SPECIAL_SINGLETON
  rb_make_metaclass
    rb_class_boot
  rb_singleton_class_attached
```

Here also `rb_class_boot()` is called. So up to that point, it's the same as in normal classes. What's going on after is what's different between normal classes and singleton classes, in other words the characteristics of singleton classes. If everything's clear so far, we just need to read `rb_singleton_class()` and `rb_make_metaclass()`.

## Compressed `rb_singleton_class()`

rb\_singleton\_class() is a little long so we'll first remove its non-essential parts.

## ▼ rb\_singleton\_class()

```
678 #define SPECIAL_SINGLETON(x,c) do {\  
679     if (obj == (x)) {\\  
680         return c;\\  
681     }\\  
682 } while (0)  
  
684 VALUE  
685 rb_singleton_class(VALUE obj)  
686     VALUE obj;  
687 {  
688     VALUE klass;  
689  
690     if (FIXNUM_P(obj) || SYMBOL_P(obj)) {  
691         rb_raise(rb_eTypeError, "can't define singleton");  
692     }  
693     if (rb_special_const_p(obj)) {  
694         SPECIAL_SINGLETON(Qnil, rb_cNilClass);  
695         SPECIAL_SINGLETON(Qfalse, rb_cFalseClass);  
696         SPECIAL_SINGLETON(Qtrue, rb_cTrueClass);  
697         rb_bug("unknown immediate %ld", obj);  
698     }  
699  
700     DEFER_INTS;  
701     if (FL_TEST(RBASIC(obj)->klass, FL_SINGLETON) &&  
702         (BUILTIN_TYPE(obj) == T_CLASS ||  
703          rb_iv_get(RBASIC(obj)->klass, "__attached__") == ob  
704          klass = RBASIC(obj)->klass;  
705     }  
706     else {  
707         klass = rb_make_metaklass(obj, RBASIC(obj)->klass);  
708     }  
709     if (OBJ_TAINTED(obj)) {  
710         OBJ_TAINT(klass);  
711     }  
712     else {
```

```
713             FL_UNSET(klass, FL_TAINT);
714         }
715         if (OBJ_FROZEN(obj)) OBJ_FREEZE(klass);
716         ALLOW_INTS;
717
718         return klass;
719     }

(class.c)
```

The first and the second half are separated by a blank line. The first half handles special cases and the second half handles the general case. In other words, the second half is the trunk of the function. That's why we'll keep it for later and talk about the first half.

Everything that is handled in the first half are non-pointer `VALUES`, it means their object structs do not exist. First, `Fixnum` and `Symbol` are explicitly picked. Then, `rb_special_const_p()` is a function that returns true for non-pointer `VALUES`, so there only `Qtrue`, `Qfalse` and `Qnil` should get caught. Other than that, there are no valid non-pointer `VALUE` so it would be reported as a bug with `rb_bug()`.

`DEFER_INTS()` and `ALLOW_INTS()` both end with the same `INTS` so you should see a pair in them. That's the case, and they are macros related to signals. Because they are defined in `rubysig.h`, you can guess that `INTS` is the abbreviation of interrupts. You can ignore them.

## Compressed `rb_make_metaclass()`

▼ `rb_make_metaclass()`

```

142 VALUE
143 rb_make_metaclass(obj, super)
144     VALUE obj, super;
145 {
146     VALUE klass = rb_class_boot(super);
147     FL_SET(klass, FL_SINGLETON);
148     RBASIC(obj)->klass = klass;
149     rb_singleton_class_attached(klass, obj);
150     if (BUILTIN_TYPE(obj) == T_CLASS) {
151         RBASIC(klass)->klass = klass;
152         if (FL_TEST(obj, FL_SINGLETON)) {
153             RCLASS(klass)->super =
154                 RBASIC(rb_class_real(RCLASS(obj)->super))
155         }
156     }
157     return klass;
158 }
```

(class.c)

We already saw `rb_class_boot()`. It creates a (normal) class using the `super` parameter as its superclass. After that, the `FL_SINGLETON` of this class is set. This is clearly suspicious. The name of the function makes us think that it is the indication of a singleton class.

## ▀ What are singleton classes?

Finishing the above process, furthermore, we'll throw away the declarations because parameters, return values and local variables are all `VALUE`. That makes us able to compress to the following:

▼ `rb_singleton_class()` `rb_make_metaclass()` (after compression)

`rb_singleton_class(obj)`

```

{
    if (FL_TEST(RBASIC(obj)->klass, FL_SINGLETON) &&
        (BUILTIN_TYPE(obj) == T_CLASS || BUILTIN_TYPE(obj) == T_M0
        rb_iv_get(RBASIC(obj)->klass, "__attached__") == obj) {
        klass = RBASIC(obj)->klass;
    }
    else {
        klass = rb_make_metaclass(obj, RBASIC(obj)->klass);
    }
    return klass;
}

rb_make_metaclass(obj, super)
{
    klass = create a class with super as superclass;
    FL_SET(klass, FL_SINGLETON);
    RBASIC(obj)->klass = klass;
    rb_singleton_class_attached(klass, obj);
    if (BUILTIN_TYPE(obj) == T_CLASS) {
        RBASIC(klass)->klass = klass;
        if (FL_TEST(obj, FL_SINGLETON)) {
            RCLASS(klass)->super =
                RBASIC(rb_class_real(RCLASS(obj)->super))->kla
        }
    }
    return klass;
}

```

The condition of the `if` statement of `rb_singleton_class()` seems quite complicated. However, this condition is not connected to `rb_make_metaclass()`, which is the mainstream, so we'll see it later. Let's first think about what happens on the false branch of the `if`.

The `BUILTIN_TYPE()` of `rb_make_metaclass()` is similar to `TYPE()` as it is a macro to get the structure type flag (`T_xxxx`). That means this check in `rb_make_metaclass` means “if `obj` is a class”. For the moment

we assume that `obj` is a class, so we'll remove it.

With these simplifications, we get the following:

▼ `rb_singleton_class()` `rb_make_metaclass()` (after recompression)

```
rb_singleton_class(obj)
{
    klass = create a class with RBASIC(obj)->klass as superclass;
    FL_SET(klass, FL_SINGLETON);
    RBASIC(obj)->klass = klass;
    return klass;
}
```

But there is still a quite hard to understand side to it. That's because `klass` is used too often. So let's rename the `klass` variable to `sclass`.

▼ `rb_singleton_class()` `rb_make_metaclass()` (variable substitution)

```
rb_singleton_class(obj)
{
    sclass = create a class with RBASIC(obj)->klass as superclass;
    FL_SET(sclass, FL_SINGLETON);
    RBASIC(obj)->klass = sclass;
    return sclass;
}
```

Now it should be very easy to understand. To make it even simpler, I've represented what is done with a diagram (figure 1). In the horizontal direction is the “instance – class” relation, and in the vertical direction is inheritance (the superclasses are above).

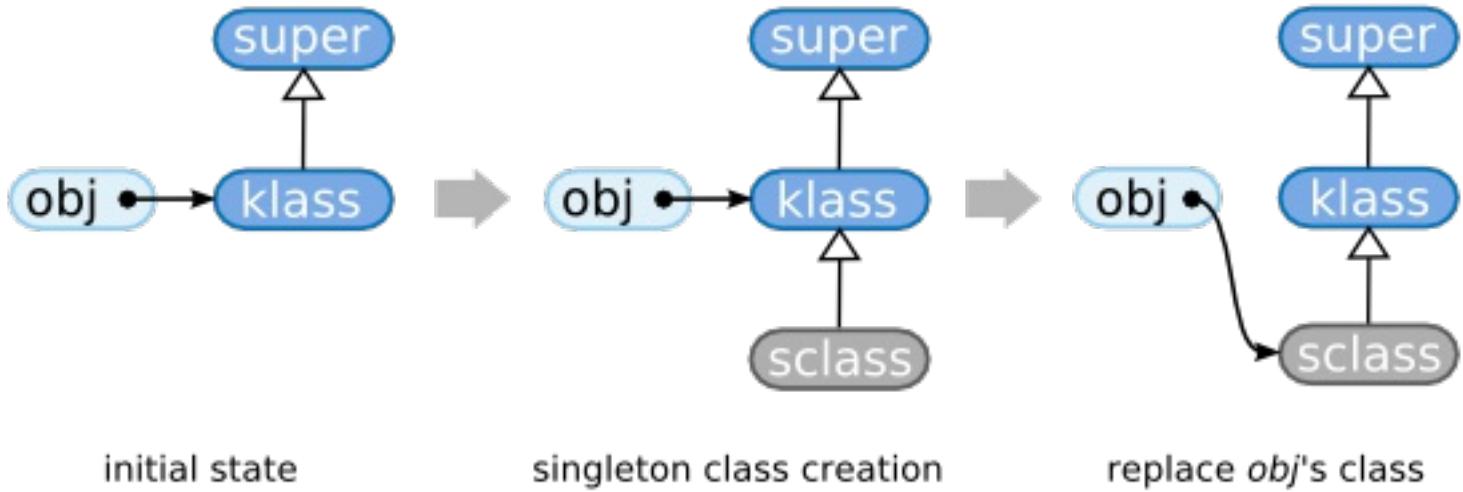


Figure 1: `rb_singleton_class`

When comparing the first and last part of this diagram, you can understand that `sclass` is inserted without changing the structure. That's all there is to singleton classes. In other words the inheritance is increased one step. By defining methods there, we can define methods which have completely nothing to do with other instances of `klass`.

## Singleton classes and instances

By the way, did you notice about, during the compression process, the call to `rb_singleton_class_attached()` was stealthily removed? Here:

```
rb_make_metaclass(obj, super)
{
    klass = create a class with super as superclass;
    FL_SET(klass, FL_SINGLETON);
    RBASIC(obj)->klass = klass;
    rb_singleton_class_attached(klass, obj); /* THIS */
```

Let's have a look at what it does.

## ▼ rb\_singleton\_class\_attached()

```
130 void
131 rb_singleton_class_attached(klass, obj)
132     VALUE klass, obj;
133 {
134     if (FL_TEST(klass, FL_SINGLETON)) {
135         if (!RCLASS(klass)->iv_tbl) {
136             RCLASS(klass)->iv_tbl = st_init_numtable();
137         }
138         st_insert(RCLASS(klass)->iv_tbl,
139                     rb_intern("__attached__"), obj);
140     }
141 }
```

(class.c)

If the `FL_SINGLETON` flag of `klass` is set... in other words if it's a singleton class, put the `__attached__` → `obj` relation in the instance variable table of `klass` (`iv_tbl`). That's how it looks like (in our case `klass` is always a singleton class... in other words its `FL_SINGLETON` flag is always set).

`__attached__` does not have the `@` prefix, but it's stored in the instance variables table so it's still an instance variable. Such an instance variable can never be read at the Ruby level so it can be used to keep values for the system's exclusive use.

Let's now think about the relationship between `klass` and `obj`. `klass` is the singleton class of `obj`. In other words, this “invisible” instance variable allows the singleton class to remember the

instance it was created from. Its value is used when the singleton class is changed, notably to call hook methods on the instance (i.e. `obj`). For example, when a method is added to a singleton class, the `obj`'s `singleton_method_added` method is called. There is no logical necessity to doing it, it was done because that's how it was defined in the language.

But is it really all right? Storing the instance in `__attached__` will force one singleton class to have only one attached instance. For example, by getting (in some way or an other) the singleton class and calling `new` on it, won't a singleton class end up having multiple instances?

This cannot be done because the proper checks are done to prevent the creation of an instance of a singleton class.

Singleton classes are in the first place for singleton methods. Singleton methods are methods existing only on a particular object. If singleton classes could have multiple instances, they would be the same as normal classes. Hence, each singleton class has only one instance ... or rather, it must be limited to one.

## ■ Summary

We've done a lot, maybe made a real mayhem, so let's finish and put everything in order with a summary.

What are singleton classes? They are classes that have the `FL_SINGLETON` flag set and that can only have one instance.

What are singleton methods? They are methods defined in the singleton class of an object.

## Metaclasses

---

### Inheritance of singleton methods

#### Infinite chain of classes

Even a class has a class, and it's `Class`. And the class of `Class` is again `Class`. We find ourselves in an infinite loop (figure 2).



Figure 2: Infinite loop of classes

Up to here it's something we've already gone through. What's going after that is the theme of this chapter. Why do classes have to make a loop?

First, in Ruby all data are objects. And classes are data in Ruby so they have to be objects.

As they are objects, they must answer to methods. And setting the rule "to answer to methods you must belong to a class" made

processing easier. That's where comes the need for a class to also have a class.

Let's base ourselves on this and think about the way to implement it. First, we can try first with the most naïve way, `Class`'s class is `ClassClass`, `ClassClass`'s class is `ClassClassClass`..., chaining classes of classes one by one. But whichever the way you look at it, this can't be implemented effectively. That's why it's common in object oriented languages where classes are objects that `Class`'s class is to `Class` itself, creating an endless virtual instance-class relationship.

((errata:

This structure is implemented efficiently in recent Ruby 1.8, thus it can be implemented efficiently. ))

I'm repeating myself, but the fact that `Class`'s class is `Class` is only to make the implementation easier, there's nothing important in this logic.

## “Class is also an object”

“Everything is an object” is often used as advertising statement when speaking about Ruby. And as a part of that, “Classes are also objects!” also appears. But these expressions often go too far. When thinking about these sayings, we have to split them in two:

- all data are objects
- classes are data

Talking about data or code makes a discussion much harder to understand. That's why here we'll restrict the meaning of "data" to "what can be put in variables in programs".

Being able to manipulate classes from programs gives programs the ability to manipulate themselves. This is called reflection. In Ruby, which is a object oriented language and furthermore has classes, it is equivalent to be able to directly manipulate classes.

Nevertheless, there's also a way in which classes are not objects. For example, there's no problem in providing a feature to manipulate classes as function-style methods (functions defined at the top-level). However, as inside the interpreter there are data structures to represent the classes, it's more natural in object oriented languages to make them available directly. And Ruby did this choice.

Furthermore, an objective in Ruby is for all data to be objects. That's why it's appropriate to make them objects.

By the way, there is also a reason not linked to reflection why in Ruby classes had to be made objects. That is to prepare the place to define methods which are independent from instances (what are called static methods in Java and C++).

And to implement static methods, another thing was necessary: singleton methods. By chain reaction, that also makes singleton classes necessary. Figure 3 shows these dependency relationships.

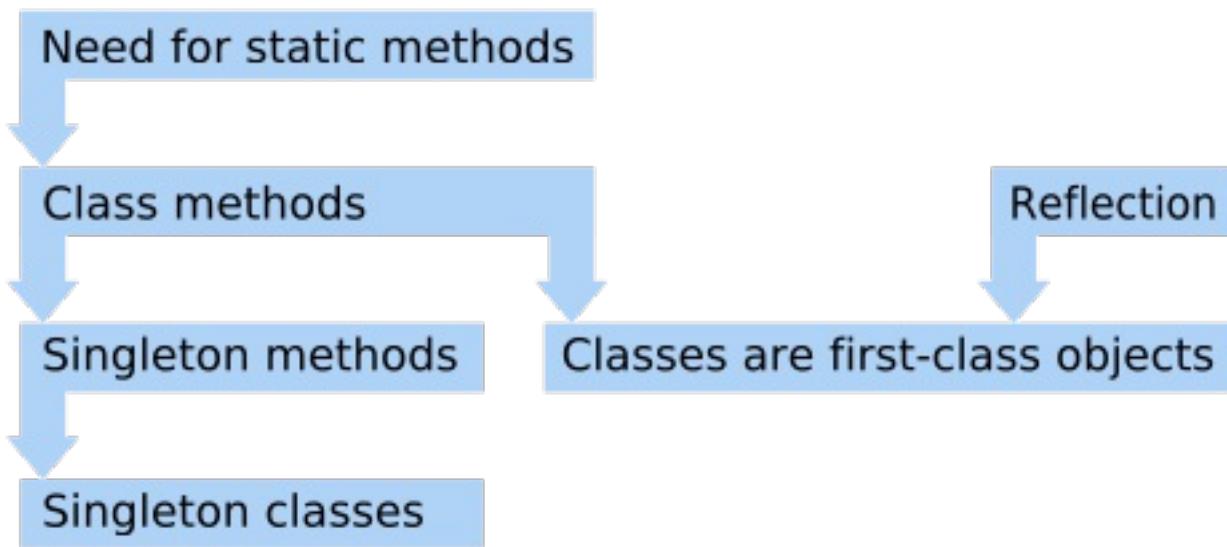


Figure 3: Requirements dependencies

## Class methods inheritance

In Ruby, singleton methods defined in a class are called class methods. However, their specification is a little strange. For some reasons, class methods are inheritable.

```

class A
  def A.test    # defines a singleton method in A
    puts("ok")
  end
end

class B < A
end

B.test() # calls it
  
```

This can't occur with singleton methods from objects that are not classes. In other words, classes are the only ones handled specially. In the following section we'll see how class methods are inherited.

# Singleton class of a class

Assuming that class methods are inherited, where is this operation done? It must be done either at class definition (creation) or at singleton method definition. Then let's first look at the code defining classes.

Class definition means of course `rb_define_class()`. Now let's take the call graph of this function.

```
rb_define_class
  rb_class_inherited
  rb_define_class_id
    rb_class_new
      rb_class_boot
    rb_make_metaclass
      rb_class_boot
    rb_singleton_class_attached
```

If you're wondering where you've seen it before, we looked at it in the previous section. At that time you did not see it but if you look closely, somehow `rb_make_metaclass()` appeared. As we saw before, this function introduces a singleton class. This is very suspicious. Why is this called even if we are not defining a singleton function? Furthermore, why is the lower level `rb_make_metaclass()` used instead of `rb_singleton_class()`? It looks like we have to check these surroundings again.

## `rb_define_class_id()`

Let's first start our reading with its caller, `rb_define_class_id()`.

## ▼ rb\_define\_class\_id()

```
160  VALUE
161  rb_define_class_id(id, super)
162      ID id;
163      VALUE super;
164  {
165      VALUE klass;
166
167      if (!super) super = rb_cObject;
168      klass = rb_class_new(super);
169      rb_name_class(klass, id);
170      rb_make_metaclass(klass, RBASIC(super)->klass);
171
172      return klass;
173 }
```

(class.c)

`rb_class_new()` was a function that creates a class with `super` as its superclass. `rb_name_class()`‘s name means it names a class, but for the moment we do not care about names so we’ll skip it. After that there’s the `rb_make_metaclass()` in question. I’m concerned by the fact that when called from `rb_singleton_class()`, the parameters were different. Last time was like this:

```
rb_make_metaclass(obj, RBASIC(obj)->klass);
```

But this time is like this:

```
rb_make_metaclass(klass, RBASIC(super)->klass);
```

So as you can see it’s slightly different. How do the results change depending on that? Let’s have once again a look at a simplified

```
rb_make_metaclass().
```

## rb\_make\_metaclass (once more)

### ▼ rb\_make\_metaclass (after first compression)

```
rb_make_metaclass(obj, super)
{
    klass = create a class with super as superclass;
    FL_SET(klass, FL_SINGLETON);
    RBASIC(obj)->klass = klass;
    rb_singleton_class_attached(klass, obj);
    if (BUILTIN_TYPE(obj) == T_CLASS) {
        RBASIC(klass)->klass = klass;
        if (FL_TEST(obj, FL_SINGLETON)) {
            RCLASS(klass)->super =
                RBASIC(rb_class_real(RCLASS(obj)->super))->kla
        }
    }
    return klass;
}
```

Last time, the `if` statement was wholly skipped, but looking once again, something is done only for `T_CLASS`, in other words classes. This clearly looks important. In `rb_define_class_id()`, as it's called like this:

```
rb_make_metaclass(klass, RBASIC(super)->klass);
```

Let's expand `rb_make_metaclass()`'s parameter variables with the actual values.

### ▼ rb\_make\_metaclass (recompression)

```

rb_make_metaclass(klass, super_klass /* == RBASIC(super)->klass */
{
    sclass = create a class with super_class as superclass;
    RBASIC(klass)->klass = sclass;
    RBASIC(sclass)->klass = sclass;
    return sclass;
}

```

Doing this as a diagram gives something like figure 4. In it, the names between parentheses are singleton classes. This notation is often used in this book so I'd like you to remember it. This means that `obj`'s singleton class is written as `(obj)`. And `(klass)` is the singleton class for `klass`. It looks like the singleton class is caught between a class and this class's superclass's class.

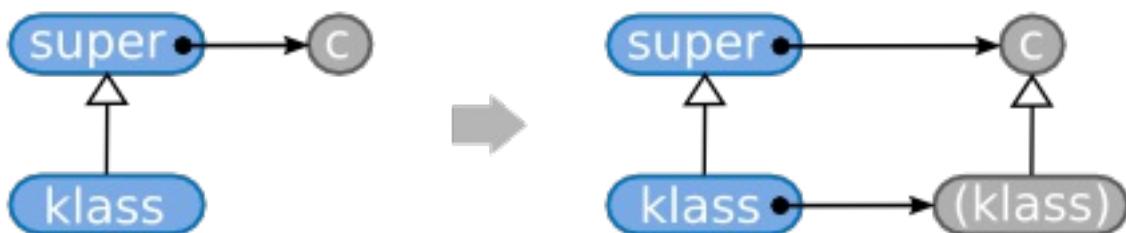


Figure 4: Introduction of a class's singleton class

By expanding our imagination further from this result, we can think that the superclass's class (the `C` in figure 4) must again be a singleton class. You'll understand with one more inheritance level (figure 5).

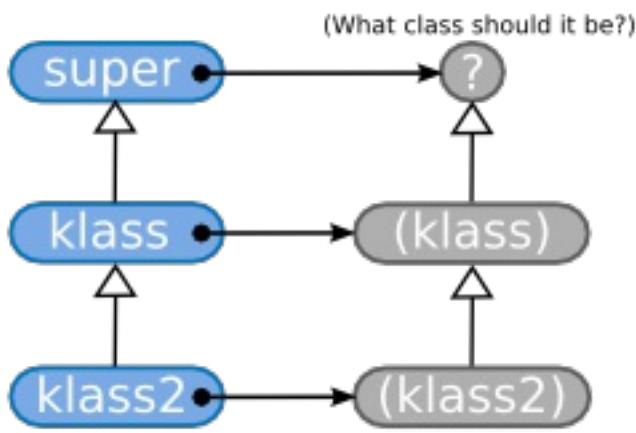


Figure 5: Hierarchy of multi-level inheritance

As the relationship between `super` and `klass` is the same as the one between `klass` and `klass2`, `c` must be the singleton class (`super`). If you continue like this, finally you'll arrive at the conclusion that `Object`'s class must be `(Object)`. And that's the case in practice. For example, by inheriting like in the following program :

```

class A < Object
end
class B < A
end
  
```

internally, a structure like figure 6 is created.

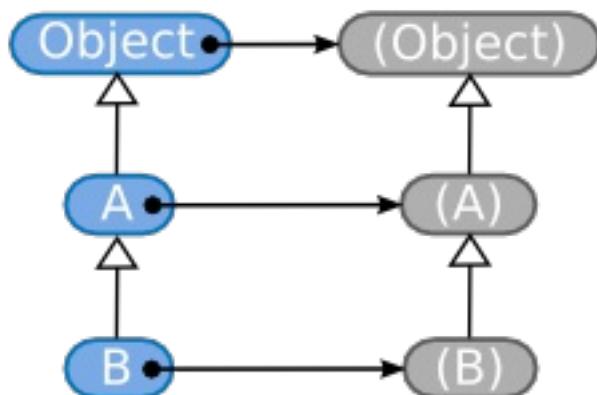


Figure 6: Class hierarchy and metaclasses

As classes and their metaclasses are linked and inherit like this, class methods are inherited.

# Class of a class of a class

You've understood the working of class methods inheritance, but by doing that, in the opposite some questions have appeared. What is the class of a class's singleton class? For this, we can check it by using debuggers. I've made figure 7 from the results of this investigation.

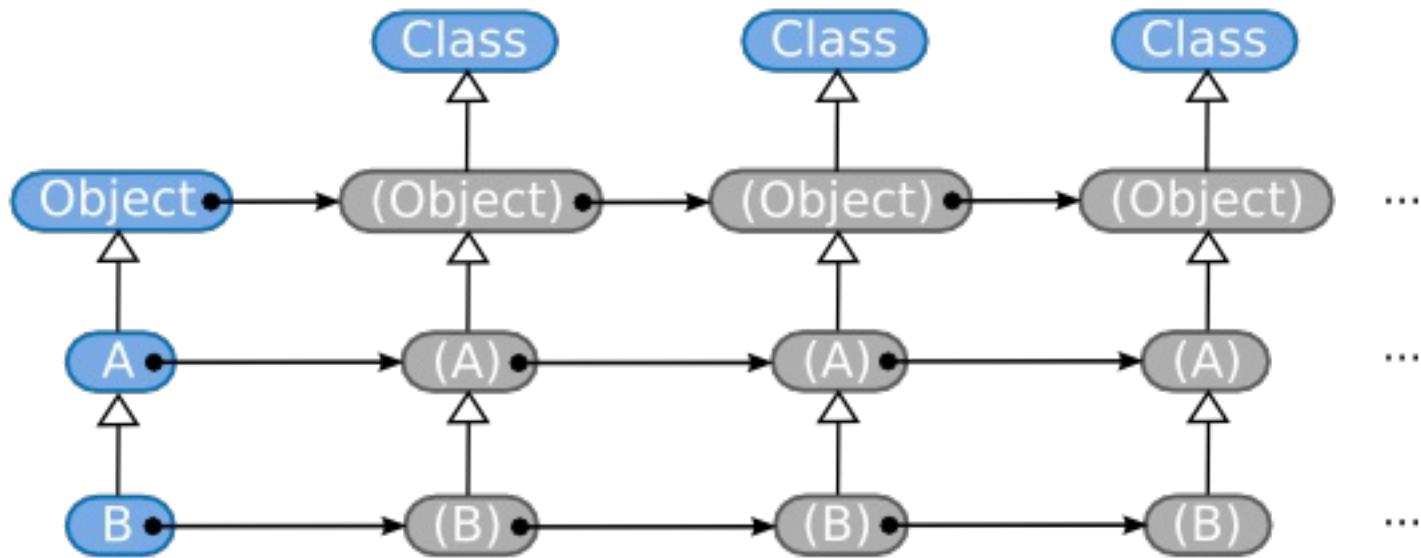


Figure 7: Class of a class's singleton class

A class's singleton class puts itself as its own class. Quite complicated.

The second question: the class of `Object` must be `Class`. Didn't I properly confirm this in chapter 1: Ruby language minimum by using `class()` method?

```
p(0bject.class()) # Class
```

Certainly, that's the case "at the Ruby level". But "at the C level", it's the singleton class (0bject). If (0bject) does not appear at the Ruby level, it's because 0bject#class skips the singleton classes. Let's look at the body of the method, rb\_obj\_class() to confirm that.

## ▼ rb\_obj\_class()

```
86  VALUE
87  rb_obj_class(obj)
88      VALUE obj;
89  {
90      return rb_class_real(CLASS_OF(obj));
91  }

76  VALUE
77  rb_class_real(cl)
78      VALUE cl;
79  {
80      while (FL_TEST(cl, FL_SINGLETON) || TYPE(cl) == T_ICLASS
81          cl = RCLASS(cl)->super;
82      }
83      return cl;
84  }
```

(object.c)

CLASS\_OF(obj) returns the basic.klass of obj. While in rb\_class\_real(), all singleton classes are skipped (advancing towards the superclass). In the first place, singleton class are caught between a class and its superclass, like a proxy. That's why when a "real" class is necessary, we have to follow the superclass

chain (figure 8).

`I_CLASS` will appear later when we will talk about include.

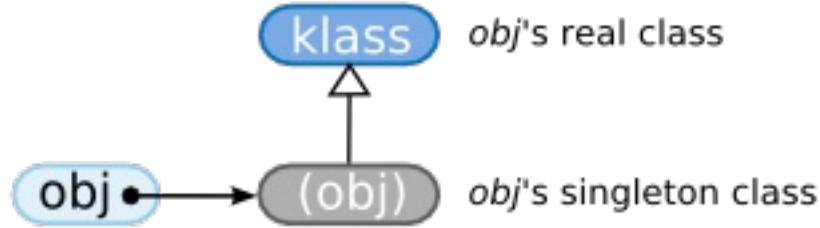


Figure 8: Singleton class and real class

## Singleton class and metaclass

Well, the singleton classes that were introduced in classes is also one type of class, it's a class's class. So it can be called metaclass.

However, you should be wary of the fact that being a singleton class does not mean being a metaclass. The singleton classes introduced in classes are metaclasses. The important fact is not that they are singleton classes, but that they are the classes of classes. I was stuck on this point when I started learning Ruby. As I may not be the only one, I would like to make this clear.

Thinking about this, the `rb_make_metaclass()` function name is not very good. When used for a class, it does indeed create a metaclass, but when used for other objects, the created class is not a metaclass.

Then finally, even if you understood that some classes are

metaclasses, it's not as if there was any concrete gain. I'd like you not to care too much about it.

## Bootstrap

We have nearly finished our talk about classes and metaclasses. But there is still one problem left. It's about the 3 metaobjects `Object`, `Module` and `Class`. These 3 cannot be created with the common use API. To make a class, its metaclass must be built, but like we saw some time ago, the metaclass's superclass is `Class`. However, as `Class` has not been created yet, the metaclass cannot be build. So in ruby, only these 3 classes's creation is handled specially.

Then let's look at the code:

### ▼ Object, Module and Class creation

```
1243 rb_cObject = boot_defclass("Object", 0);
1244 rb_cModule = boot_defclass("Module", rb_cObject);
1245 rb_cClass = boot_defclass("Class", rb_cModule);
1246
1247 metaclass = rb_make_metaclass(rb_cObject, rb_cClass);
1248 metaclass = rb_make_metaclass(rb_cModule, metaclass);
1249 metaclass = rb_make_metaclass(rb_cClass, metaclass);

(object.c)
```

First, in the first half, `boot_defclass()` is similar to `rb_class_boot()`, it just creates a class with its given superclass set. These links give us something like the left part of figure 9.

And in the three lines of the second half, (Object), (Module) and (Class) are created and set (right figure 9). (Object) and (Module)‘s classes... that is themselves... is already set in `rb_make_metaclass()` so there is no problem. With this, the metaobjects’ bootstrap is finished.

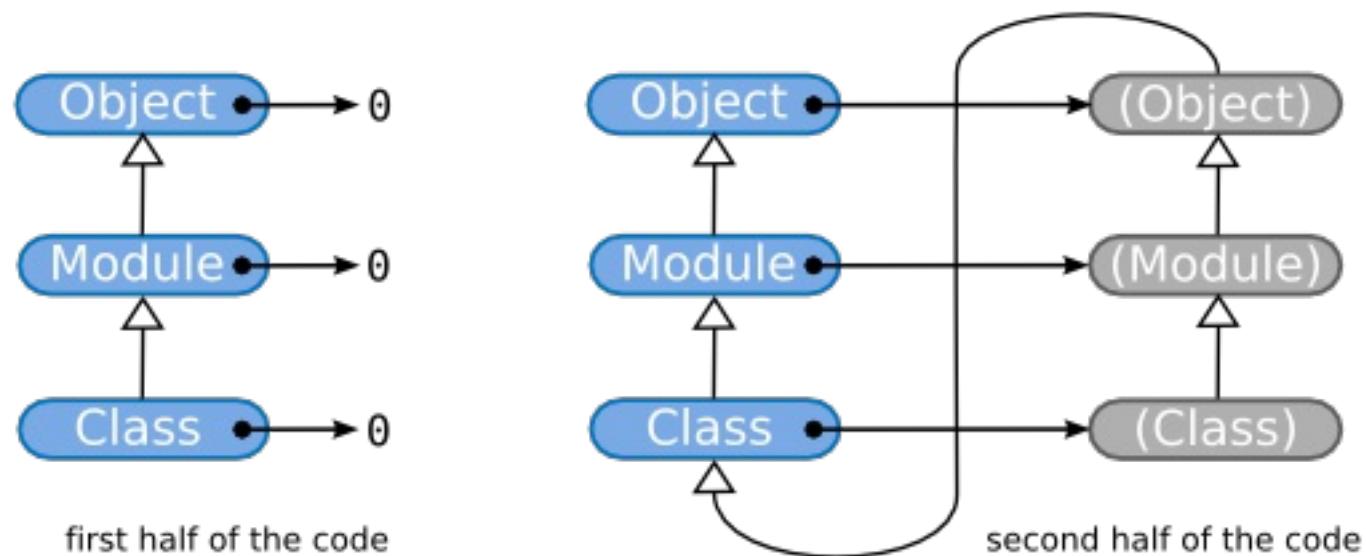


Figure 9: Metaobjects creation

After taking everything into account, it gives us the final shape like figure 10.

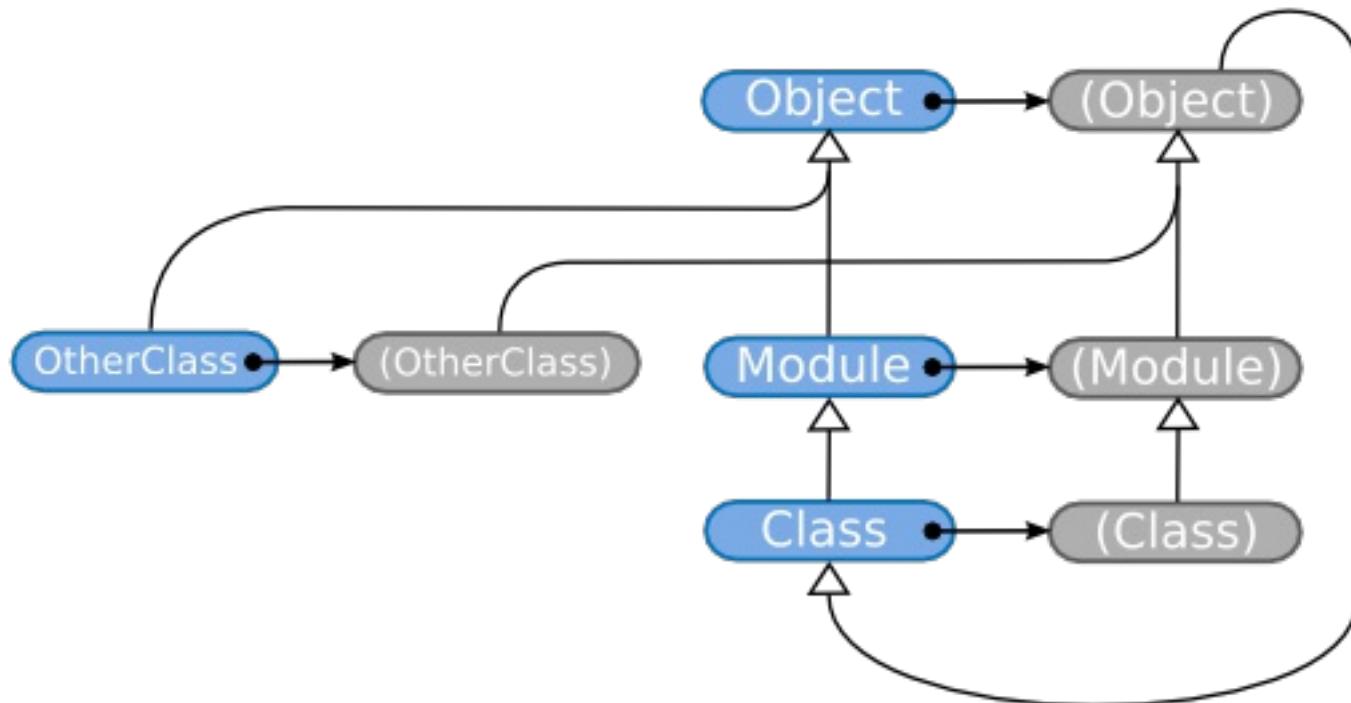


Figure 10: Ruby metaobjects

## Class names

In this section, we will analyse how's formed the reciprocal conversion between class and class names, in other words constants. Concretely, we will target `rb_define_class()` and `rb_define_class_under()`.

### ■ Name → class

First we'll read `rb_defined_class()`. After the end of this function, the class can be found from the constant.

▼ `rb_define_class()`

```

183 VALUE
184 rb_define_class(name, super)
185     const char *name;
186     VALUE super;
187 {
188     VALUE klass;
189     ID id;
190
191     id = rb_intern(name);
192     if (rb_autoload_defined(id)) { /* (A) autoload */
193         rb_autoload_load(id);
194     }
195     if (rb_const_defined(rb_cObject, id)) { /* (B) rb_const */
196         klass = rb_const_get(rb_cObject, id); /* (C) rb_const */
197         if (TYPE(klass) != T_CLASS) {
198             rb_raise(rb_eTypeError, "%s is not a class", name);
199         } /* (D) rb_class */
200         if (rb_class_real(RCLASS(klass)->super) != super) {
201             rb_name_error(id, "%s is already defined", name);
202         }
203         return klass;
204     }
205     if (!super) {
206         rb_warn("no super class for '%s', Object assumed", name);
207     }
208     klass = rb_define_class_id(id, super);
209     rb_class_inherited(super, klass);
210     st_add_direct(rb_class_tbl, id, klass);
211
212     return klass;
213 }

```

(class.c)

This can be clearly divided into the two parts: before and after `rb_define_class_id()`. The former is to acquire or create the class. The latter is to assign it to the constant. We will look at it in more detail below.

- (A) In Ruby, there is a feature named autoload that automatically loads libraries when certain constants are accessed. These functions named `rb_autoload_xxxx()` are for its checks. You can ignore it without any problem.
- (B) We determine whether the `name` constant has been defined or not in `Object`.
- (C) Get the value of the `name` constant. This will be explained in detail in chapter 6.
- (D) We've seen `rb_class_real()` some time ago. If the class `c` is a singleton class or an `ICLASS`, it climbs the `super` hierarchy up to a class that is not and returns it. In short, this function skips the virtual classes that should not appear at the Ruby level.

That's what we can read nearby.

As constants are involved around this, it is very troublesome. But I feel like the chapter about constants is probably not so right place to talk about class definition, that's the reason of such halfway description around here.

Moreover, about this coming after `rb_define_class_id()`,

```
st_add_direct(rb_class_tbl, id, klass);
```

This part assigns the class to the constant. However, whichever way you look at it you do not see that. In fact, top-level classes and

modules that are defined in C are separated from the other constants and regrouped in `rb_class_tbl()`. The split is slightly related to the GC. It's not essential.

## Class → name

We understood how the class can be obtained from the class name, but how to do the opposite? By doing things like calling `p` or `Class#name`, we can get the name of the class, but how is it implemented?

In fact this is done by `rb_name_class()` which already appeared a long time ago. The call is around the following:

```
rb_define_class
  rb_define_class_id
    rb_name_class
```

Let's look at its content:

### ▼ `rb_name_class()`

```
269 void
270 rb_name_class(klass, id)
271     VALUE klass;
272     ID id;
273 {
274     rb_iv_set(klass, "__classid__", ID2SYM(id));
275 }
```

`(variable.c)`

`_classid_` is another instance variable that can't be seen from Ruby. As only `VALUES` can be put in the instance variable table, the `ID` is converted to `Symbol` using `ID2SYM()`.

That's how we are able to find the constant name from the class.

## Nested classes

So, in the case of classes defined at the top-level, we know how works the reciprocal link between name and class. What's left is the case of classes defined in modules or other classes, and for that it's a little more complicated. The function to define these nested classes is `rb_define_class_under()`.

### ▼ `rb_define_class_under()`

```
215  VALUE
216  rb_define_class_under(outer, name, super)
217      VALUE outer;
218      const char *name;
219      VALUE super;
220  {
221      VALUE klass;
222      ID id;
223
224      id = rb_intern(name);
225      if (rb_const_defined_at(outer, id)) {
226          klass = rb_const_get(outer, id);
227          if (TYPE(klass) != T_CLASS) {
228              rb_raise(rb_eTypeError, "%s is not a class", nam
229          }
230          if (rb_class_real(RCLASS(klass)->super) != super) {
231              rb_name_error(id, "%s is already defined", name)
232          }
233      return klass;
```

```
234     }
235     if (!super) {
236         rb_warn("no super class for '%s::%s', Object assumed
237                 rb_class2name(outer), name);
238     }
239     klass = rb_define_class_id(id, super);
240     rb_set_class_path(klass, outer, name);
241     rb_class_inherited(super, klass);
242     rb_const_set(outer, id, klass);
243
244     return klass;
245 }
```

(class.c)

The structure is like the one of `rb_define_class()`: before the call to `rb_define_class_id()` is the redefinition check, after is the creation of the reciprocal link between constant and class. The first half is pretty boringly similar to `rb_define_class()` so we'll skip it. In the second half, `rb_set_class_path()` is new. We're going to look at it.

## rb\_set\_class\_path()

This function gives the name `name` to the class `klass` nested in the class `under`. “class path” means a constant name including all the nesting information starting from top-level, for example “`Net::NetPrivate::Socket`”.

### ▼ rb\_set\_class\_path()

```
210 void
211 rb_set_class_path(klass, under, name)
212     VALUE klass, under;
213     const char *name;
```

```
214  {
215      VALUE str;
216
217      if (under == rb_cObject) {
218          /* defined at top-level */
219          str = rb_str_new2(name);      /* create a Ruby string
220      }
221      else {
222          /* nested constant */
223          str = rb_str_dup(rb_class_path(under)); /* copy the
224          rb_str_cat2(str, "::");      /* concatenate "::" */
225          rb_str_cat2(str, name);      /* concatenate name */
226      }
227      rb_iv_set(klass, "__classpath__", str);
228  }
```

(variable.c)

Everything except the last line is the construction of the class path, and the last line makes the class remember its own name.

`__classpath__` is of course another instance variable that can't be seen from a Ruby program. In `rb_name_class()` there was `__classid__`, but `id` is different because it does not include nesting information (look at the table below).

<code>__classpath__</code>	<code>Net::NetPrivate::Socket</code>
<code>__classid__</code>	<code>Socket</code>

It means classes defined for example in `rb_defined_class()` all have `__classid__` or `__classpath__` defined. So to find `under`'s classpath we can look up in these instance variables. This is done by `rb_class_path()`. We'll omit its content.

## ■ Nameless classes

Contrary to what I have just said, there are in fact cases in which neither `_classpath_` nor `_classid_` are set. That is because in Ruby you can use a method like the following to create a class.

```
c = Class.new()
```

If a class is created like this, it won't go through `rb_define_class_id()` and the classpath won't be set. In this case, `c` does not have any name, which is to say we get an unnamed class.

However, if later it's assigned to a constant, a name will be attached to the class at that moment.

```
SomeClass = c # the class name is SomeClass
```

Strictly speaking, at the first time requesting the name after assigning it to a constant, the name will be attached to the class. For instance, when calling `p` on this `SomeClass` class or when calling the `Class#name` method. When doing this, a value equal to the class is searched in `rb_class_tbl`, and a name has to be chosen. The following case can also happen:

```
class A
  class B
    C = tmp = Class.new()
    p(tmp) # here we search for the name
  end
end
```

so in the worst case we have to search for the whole constant

space. However, generally, there aren't many constants so even searching all constants does not take too much time.

## Include

---

We only talked about classes so let's finish this chapter with something else and talk about module inclusion.

### rb\_include\_module (1)

Includes are done by the ordinary method `Module#include`. Its corresponding function in C is `rb_include_module()`. In fact, to be precise, its body is `rb_mod_include()`, and there `Module#append_feature` is called, and this function's default implementation finally calls `rb_include_module()`. Mixing what's happening in Ruby and C gives us the following call graph.

```
Module#include (rb_mod_include)
  Module#append_features (rb_mod_append_features)
    rb_include_module
```

Anyway, the manipulations that are usually regarded as inclusions are done by `rb_include_module()`. This function is a little long so we'll look at it a half at a time.

### ▼ rb\_include\_module (first half)

```

/* include module in class */
347 void
348 rb_include_module(klass, module)
349     VALUE klass, module;
350 {
351     VALUE p, c;
352     int changed = 0;
353
354     rb_frozen_class_p(klass);
355     if (!OBJ_TAINTED(klass)) {
356         rb_secure(4);
357     }
358
359     if (NIL_P(module)) return;
360     if (klass == module) return;
361
362     switch (TYPE(module)) {
363         case T_MODULE:
364         case T_CLASS:
365         case T_ICLASS:
366             break;
367         default:
368             Check_Type(module, T_MODULE);
369     }

```

(class.c)

For the moment it's only security and type checking, therefore we can ignore it. The process itself is below:

## ▼ rb\_include\_module (second half)

```

371     OBJ_INFECTION(klass, module);
372     c = klass;
373     while (module) {
374         int superclass_seen = Qfalse;
375
376         if (RCLASS(klass)->m_tbl == RCLASS(module)->m_tbl)
377             rb_raise(rb_eArgError, "cyclic include detected");
378         /* (A) skip if the superclass already includes module */

```

```

379         for (p = RCLASS(klass)->super; p; p = RCLASS(p)->sup
380             switch (BUILTIN_TYPE(p)) {
381                 case T_ICLASS:
382                     if (RCLASS(p)->m_tbl == RCLASS(module)->m_tb
383                         if (!superclass_seen) {
384                             c = p; /* move the insertion point
385                         }
386                         goto skip;
387                     }
388                     break;
389                 case T_CLASS:
390                     superclass_seen = Qtrue;
391                     break;
392                 }
393             }
394             c = RCLASS(c)->super =
395                 include_class_new(module, RCLASS(c)->sup
396                 changed = 1;
397             skip:
398                 module = RCLASS(module)->super;
399             }
400         if (changed) rb_clear_cache();
401     }

```

(class.c)

First, what the (A) block does is written in the comment. It seems to be a special condition so let's first skip reading it for now. By extracting the important parts from the rest we get the following:

```

c = klass;
while (module) {
    c = RCLASS(c)->super = include_class_new(module, RCLASS(c)->
    module = RCLASS(module)->super;
}

```

In other words, it's a repetition of `module`'s `super`. What is in `module`'s `super` must be a module included by `module` (because our intuition

tells us so). Then the superclass of the class where the inclusion occurs is replaced with something. We do not understand much what, but at the moment I saw that I felt “Ah, doesn’t this look the addition of elements to a list (like LISP’s cons)?” and it suddenly make the story faster. In other words it’s the following form:

```
list = new(item, list)
```

Thinking about this, it seems we can expect that module is inserted between `c` and `c->super`. If it’s like this, it fits module’s specification.

But to be sure of this we have to look at `include_class_new()`.

## ■ `include_class_new()`

### ▼ `include_class_new()`

```
319 static VALUE
320 include_class_new(module, super)
321     VALUE module, super;
322 {
323     NEWOBJ(klass, struct RClass);          /* (A) */
324     OBJSETUP(klass, rb_cClass, T_ICLASS);
325
326     if (BUILTIN_TYPE(module) == T_ICLASS) {
327         module = RBASIC(module)->klass;
328     }
329     if (!RCLASS(module)->iv_tbl) {
330         RCLASS(module)->iv_tbl = st_init_numtable();
331     }
332     klass->iv_tbl = RCLASS(module)->iv_tbl;    /* (B) */
333     klass->m_tbl = RCLASS(module)->m_tbl;
334     klass->super = super;                      /* (C) */
```

```
335     if (TYPE(module) == T_ICLASS) { /* (D) */
336         RBASIC(klass)->klass = RBASIC(module)->klass; /* (
337     }
338     else {
339         RBASIC(klass)->klass = module; /* (
340     }
341     OBJ_INFECT(klass, module);
342     OBJ_INFECT(klass, super);
343
344     return (VALUE)klass;
345 }
```

(class.c)

We're lucky there's nothing we do not know.

(A) First create a new class.

(B) Transplant `module`'s instance variable and method tables into this class.

(C) Make the including class's superclass (`super`) the super class of this new class.

In other words, it looks like this function creates an include class which we can regard it as something like an “avatar” of the `module`. The important point is that at (B) only the pointer is moved on, without duplicating the table. Later, if a method is added, the module's body and the include class will still have exactly the same methods (figure 11).

Include class

Module

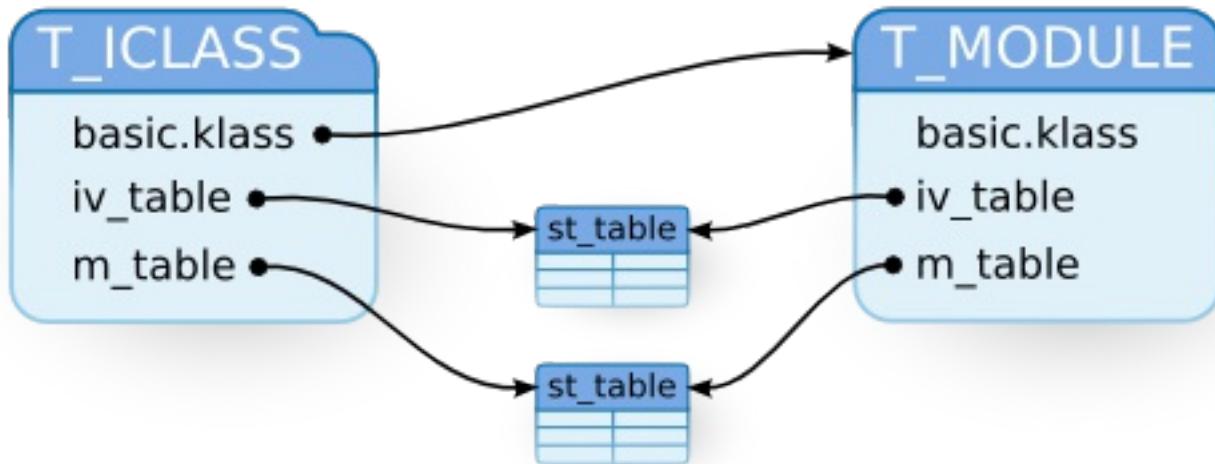
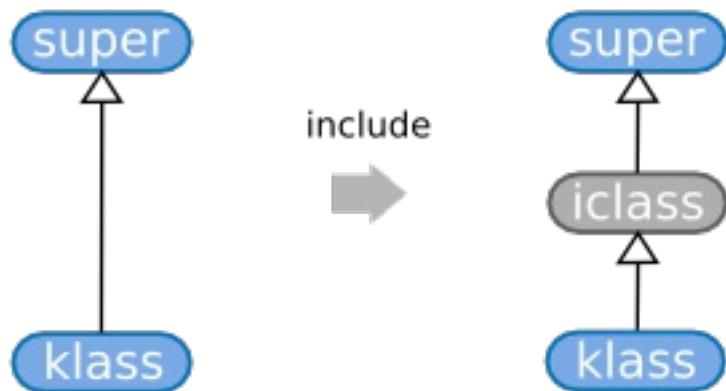


Figure 11: Include class

If you look closely at (A), the structure type flag is set to **T\_ICLASS**. This seems to be the mark of an include class. This function's name is `include_class_new()` so **ICLASS**'s **I** must be **include**.

And if you think about joining what this function and `rb_include_module()` do, we know that our previous expectations were not wrong. In brief, including is inserting the include class of a module between a class and its superclass (figure 12).



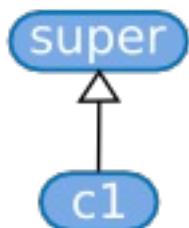
## Figure 12: Include

At (D-2) the module is stored in the include class's `klass`. At (D-1), the module's body is taken out... I'd like to say so if possible, but in fact this check does not have any use. The `T_ICLASS` check is already done at the beginning of this function, so when arriving here there can't still be a `T_ICLASS`. Modification to `ruby` piled up at piece by piece during quite a long period of time so there are quite a few small overlooks.

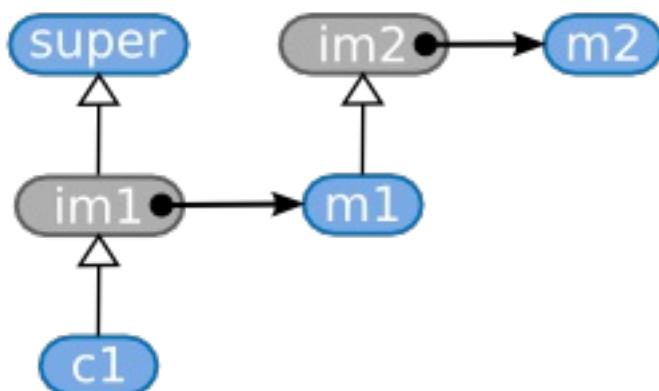
There is one more thing to consider. Somehow the include class's `basic(klass)` is only used to point to the module's body, so for example calling a method on the include class would be very bad. So include classes must not be seen from Ruby programs. And in practice all methods skip include classes, with no exception.

## ■ Simulation

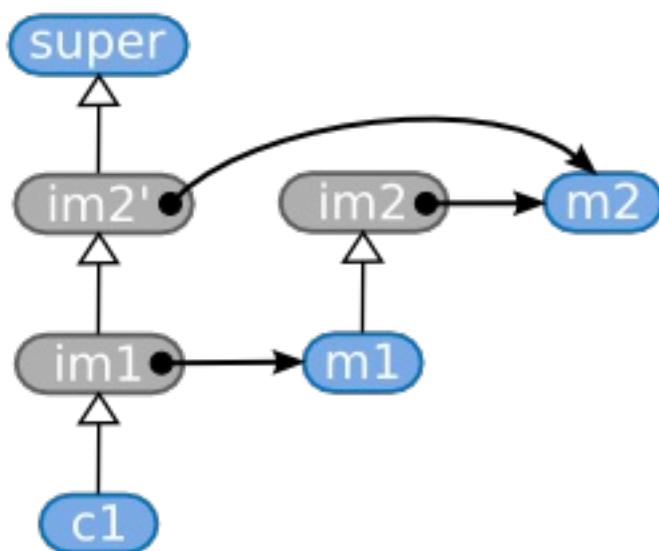
It was complicated so let's look at a concrete example. I'd like you to look at figure 13 (1). We have the `c1` class and the `m1` module that includes `m2`. From there, the changes made to include `m1` in `c1` are (2) and (3). `ims` are of course include classes.



(1) before include



(2) `rb_include_module()`'s first loop iteration  
module = `m1`



(3) `rb_include_module()`'s second loop iteration  
module = `im2`

Figure 13: Include

## rb\_include\_module (2)

Well, now we can explain the part of `rb_include_module()` we skipped.

## ▼ rb\_include\_module (avoiding double inclusion)

```
378 /* (A) skip if the superclass already includes module */
379 for (p = RCLASS(klass)->super; p; p = RCLASS(p)->super) {
380     switch (BUILTIN_TYPE(p)) {
381     case T_ICLASS:
382         if (RCLASS(p)->m_tbl == RCLASS(module)->m_tbl) {
383             if (!superclass_seen) {
384                 c = p; /* the inserting point is moved */
385             }
386             goto skip;
387         }
388         break;
389     case T_CLASS:
390         superclass_seen = Qtrue;
391         break;
392     }
393 }
```

(class.c)

Among the superclasses of the `klass` (`p`), if a `p` is `T_ICLASS` (an include class) and has the same method table as the one of the module we want to include (`module`), it means that the `p` is an include class of the `module`. Therefore, it would be skipped to not include the module twice. However, if this module includes another module (`module->super`), It would be checked once more.

But, because `p` is a module that has been included once, the modules included by it must also already be included... that's what I thought for a moment, but we can have the following context:

```
module M
end
module M2
```

```
end
class C
  include M    # M2 is not yet included in M
end          # therefore M2 is not in C's superclasses

module M
  include M2  # as there M2 is included in M,
end
class C
  include M    # I would like here to only add M2
end
```

To say this conversely, there are cases that a result of `include` is not propagated soon.

For class inheritance, the class's singleton methods were inherited but in the case of module there is no such thing. Therefore the singleton methods of the module are not inherited by the including class (or module). When you want to also inherit singleton methods, the usual way is to override `Module#append_features`.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

Translated by Sebastian Krause & ocha-

# Chapter 5: Garbage Collection

## A conception of an executing program

It's all of a sudden but at the beginning of this chapter, we'll learn about the memory space of an executing program. In this chapter we'll step inside the lower level parts of a computer quite a bit, so without preliminary knowledge it'll be hard to follow. And it'll be also necessary for the following chapters. Once we finish this here, the rest will be easier.

### Memory Segments

A general C program has the following parts in the memory space:

1. the text area
2. a place for static and global variables
3. the machine stack
4. the heap

The text area is where the code lies. Obviously the second area

holds static and global variables. Arguments and local variables of functions are piling up in the machine stack. The heap is the place where allocated by `malloc()`.

Let's talk a bit more about number three, the machine stack. Since it is called the machine "stack", obviously it has a stack structure. In other words, new stuff is piled on top of it one after another. When we actually pushes values on the stack, each value would be a tiny piece such as `int`. But logically, there are a little larger pieces. They are called stack frames.

One stack frame corresponds to one function call. Or in other words when there is a function call, one stack frame is pushed. When doing `return`, one stack frame will be popped. Figure 1 shows the really simplified appearance of the machine stack.

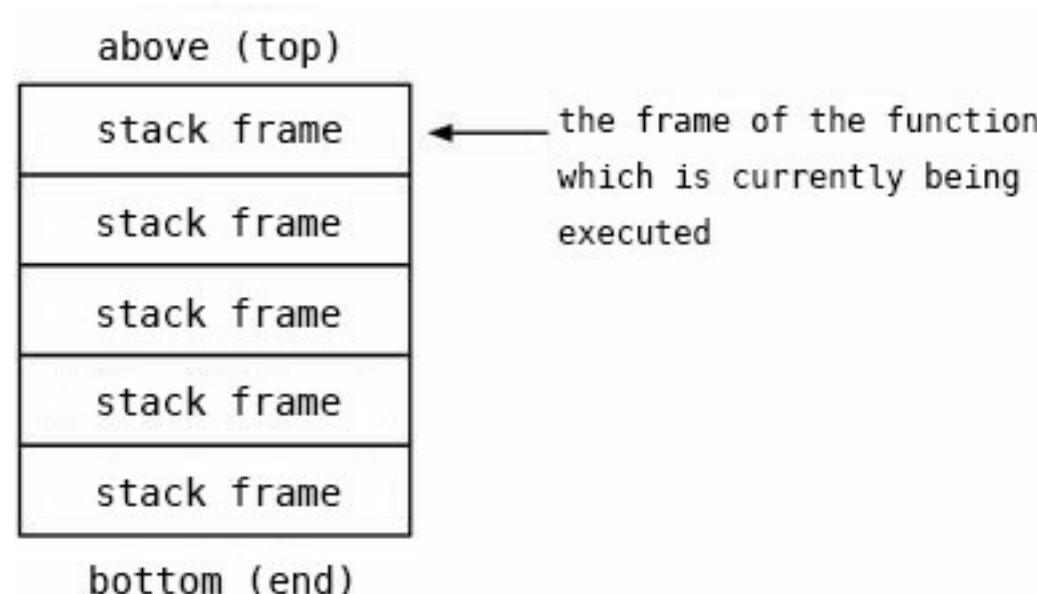


Figure 1: Machine Stack

In this picture, "above" is written above the top of the stack, but this it is not necessarily always the case that the machine stack

goes from low addresses to high addresses. For instance, on the x86 machine the stack goes from high to low addresses.

## ■ `alloca()`

By using `malloc()`, we can get an arbitrarily large memory area of the heap. `alloca()` is the machine stack version of it. But unlike `malloc()` it's not necessary to free the memory allocated with `alloca()`. Or one should say: it is freed automatically at the same moment of `return` of each function. That's why it's not possible to use an allocated value as the return value. It's the same as "You must not return the pointer to a local variable."

There's been not any difficulty. We can consider it something to locally allocate an array whose size can be changed at runtime.

However there exist environments where there is no native `alloca()`. There are still many who would like to use `alloca()` even if in such environment, sometimes a function to do the same thing is written in C. But in that case, only the feature that we don't have to free it by ourselves is implemented and it does not necessarily allocate the memory on the machine stack. In fact, it often does not. If it were possible, a native `alloca()` could have been implemented in the first place.

How can one implement `alloca()` in C? The simplest implementation is: first allocate memory normally with `malloc()`. Then remember the pair of the function which called `alloca()` and

the assigned addresses in a global list. After that, check this list whenever `alloca()` is called, if there are the memories allocated for the functions already finished, free them by using `free()`.

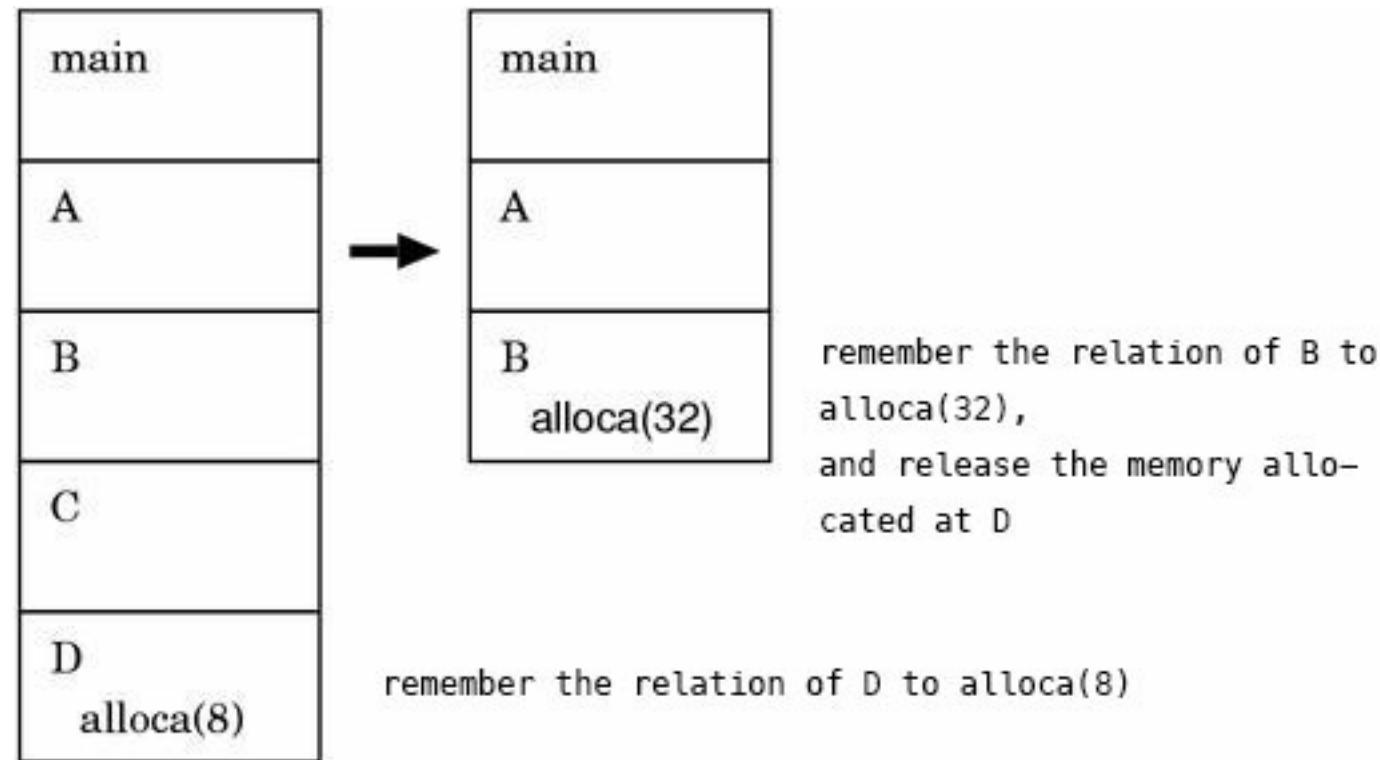


Figure 2: The behavior of an `alloca()` implemented in C

The `missing/alloca.c` of ruby is an example of an emulated `alloca()`.

## Overview

---

From here on we can at last talk about the main subject of this chapter: garbage collection.

## What is GC?

Objects are normally on top of the memory. Naturally, if a lot of objects are created, a lot of memory is used. If memory were infinite there would be no problem, but in reality there is always a memory limit. That's why the memory which is not used anymore must be collected and recycled. More concretely the memory received through `malloc()` must be returned with `free()`.

However, it would require a lot of efforts if the management of `malloc()` and `free()` were entirely left to programmers. Especially in object oriented programs, because objects are referring each other, it is difficult to tell when to release memory.

There garbage collection comes in. Garbage Collection (GC) is a feature to automatically detect and free the memory which has become unnecessary. With garbage collection, the worry “When should I have to `free()` ??” has become unnecessary. Between when it exists and when it does not exist, the ease of writing programs differs considerably.

By the way, in a book about something that I've read, there's a description “the thing to tidy up the fragmented usable memory is GC”. This task is called “compaction”. It is compaction because it makes a thing compact. Because compaction makes memory cache more often hit, it has effects for speed-up to some extent, but it is not the main purpose of GC. The purpose of GC is to collect memory. There are many GCs which collect memories but don't do compaction. The GC of ruby also does not do compaction.

Then, in what kind of system is GC available? In C and C++, there's Boehm GC\footnote{Boehm GC}

[http://www.hpl.hp.com/personal/Hans\\_Boehm/gc](http://www.hpl.hp.com/personal/Hans_Boehm/gc) which can be used as an add-on. And, for the recent languages such as Java and Perl, Python, C#, Eiffel, GC is a standard equipment. And of course, Ruby has its GC. Let's follow the details of ruby's GC in this chapter. The target file is `gc.c`.

## What does GC do?

Before explaining the GC algorithm, I should explain “what garbage collection is”. In other words, what kind of state of the memory is “the unnecessary memory”?

To make descriptions more concrete, let's simplify the structure by assuming that there are only objects and links. This would look as shown in Figure 3.

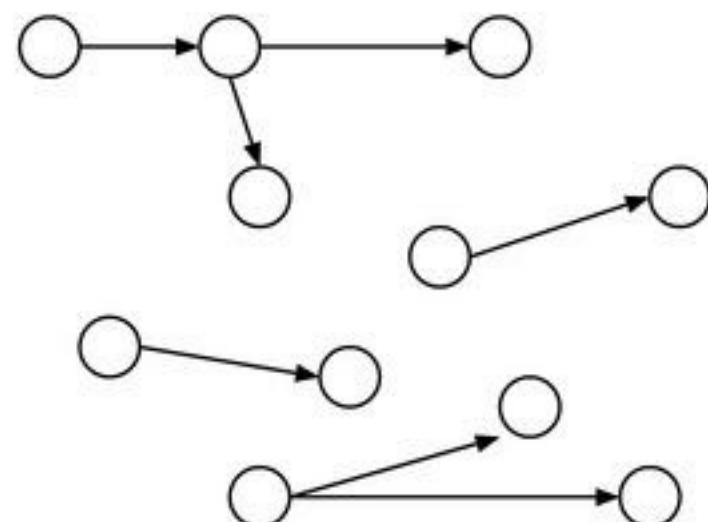


Figure 3: Objects

The objects pointed to by global variables and the objects on the

stack of a language are surely necessary. And objects pointed to by instance variables of these objects are also necessary. Furthermore, the objects that are reachable by following links from these objects are also necessary.

To put it more logically, the necessary objects are all objects which can be reached recursively via links from the “surely necessary objects” as the start points. This is depicted in figure 4. What are on the left of the line are all “surely necessary objects”, and the objects which can be reached from them are colored black. These objects colored black are the necessary objects. The rest of the objects can be released.

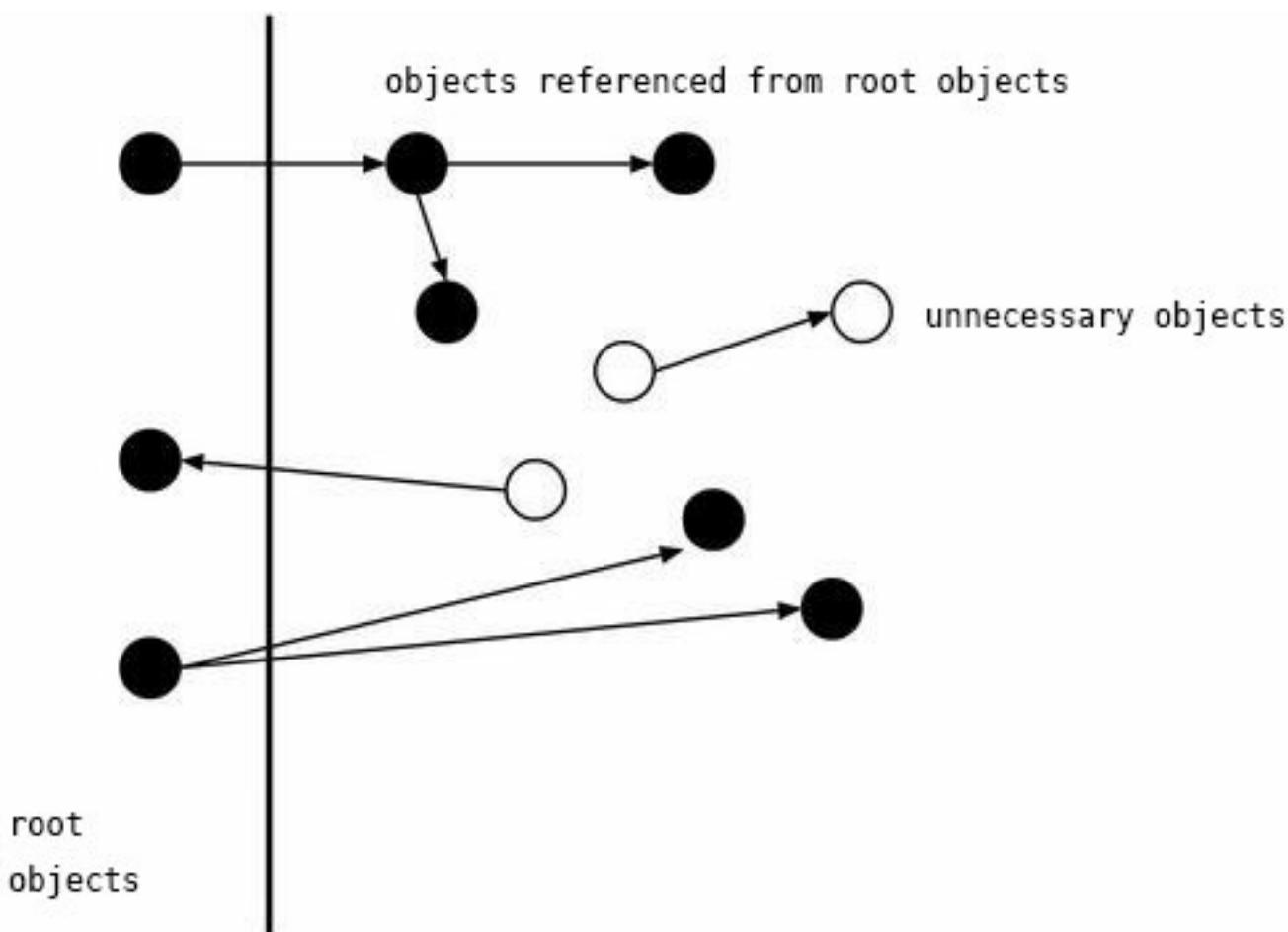


Figure 4: necessary objects and unnecessary objects

In technical terms, “the surely necessary objects” are called “the roots of GC”. That’s because they are the roots of tree structures that emerges as a consequence of tracing necessary objects.

## Mark and Sweep

GC was first implemented in Lisp. The GC implemented in Lisp at first, it means the world’s first GC, is called mark&sweep GC. The GC of ruby is one type of it.

The image of Mark-and-Sweep GC is pretty close to our definition of “necessary object”. First, put “marks” on the root objects. Setting them as the start points, put “marks” on all reachable objects. This is the mark phase.

At the moment when there’s not any reachable object left, check all objects in the object pool, release (sweep) all objects that have not marked. “Sweep” is the “sweep” of Minesweeper.

There are two advantages.

- There does not need to be any (or almost any) concern for garbage collection outside the implementation of GC.
- Cycles can also be released. (As for cycles, see also the section of “Reference Count”)

There are also two disadvantages.

- In order to sweep every object must be touched at least once.

- The load of the GC is concentrated at one point.

When using the emacs editor, there sometimes appears " Garbage collecting... " and it completely stops reacting. That is an example of the second disadvantage. But this point can be alleviated by modifying the algorithm (it is called incremental GC).

## ■ Stop and Copy

Stop and Copy is a variation of Mark and Sweep. First, prepare several object areas. To simplify this description, assume there are two areas A and B here. And put an “active” mark on the one of the areas. When creating an object, create it only in the “active” one. (Figure 5)

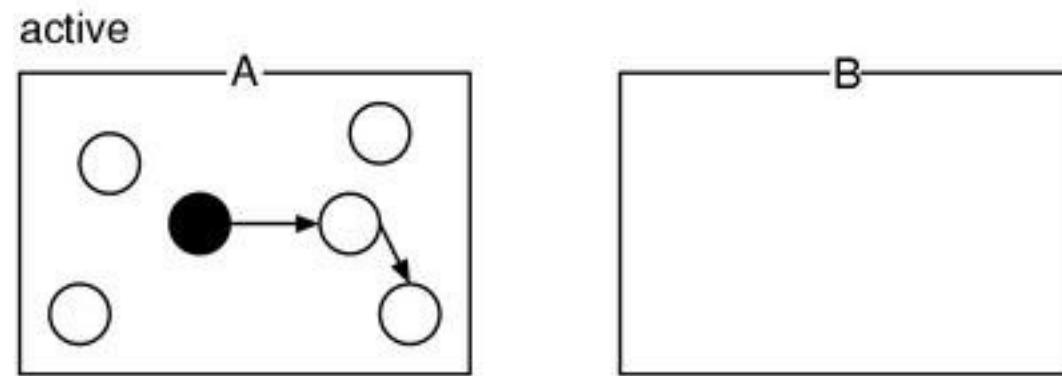


Figure 5: Stop and Copy (1)

When the GC starts, follow links from the roots in the same manner as mark-and-sweep. However, move objects to another area instead of marking them (Figure 6). When all the links have been followed, discard the all elements which remain in A, and make B active next.

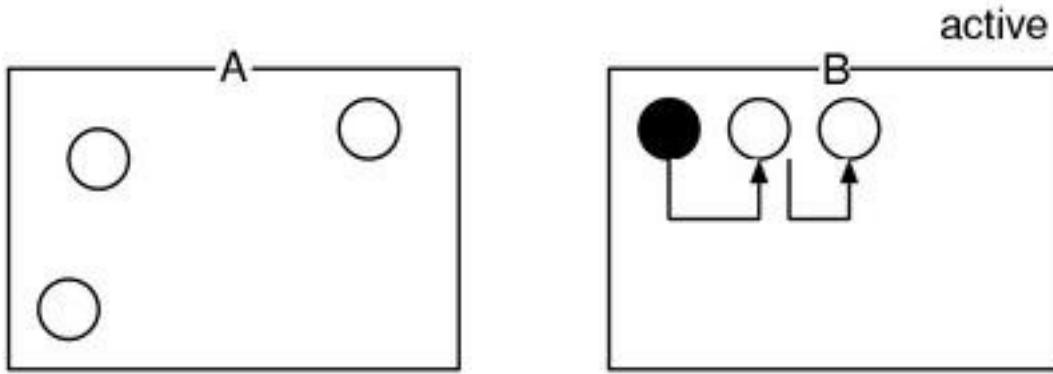


Figure 6: Stop and Copy (2)

Stop and Copy also has two advantages:

- Compaction happens at the same time as collecting the memory
- Since objects that reference each other move closer together, there's more possibility of hitting the cache.

And also two disadvantages:

- The object area needs to be more than twice as big
- The positions of objects will be changed

It seems what exist in this world are not only positive things.

## Reference counting

Reference counting differs a bit from the aforementioned GCs, the reach-check code is distributed in several places.

First, attach an integer count to each element. When referring via variables or arrays, the counter of the referenced object is increased. When quitting to refer, decrease the counter. When the counter of an object becomes zero, release the object. This is the

method called reference counting (Figure 7).

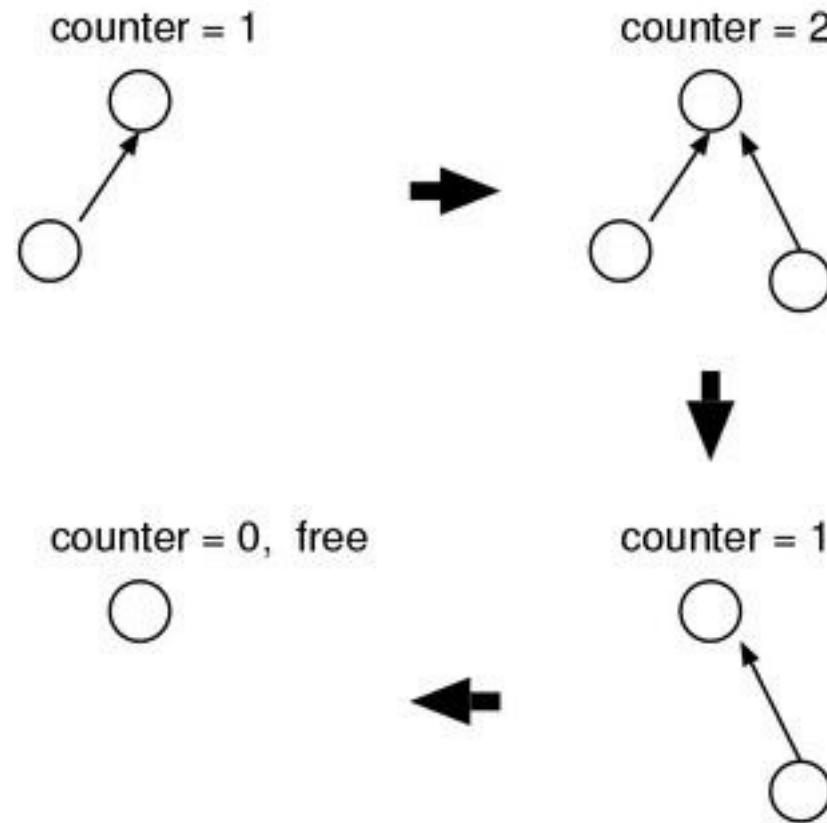


Figure 7: Reference counting

This method also has two advantages:

- The load of GC is distributed over the entire program.
- The object that becomes unnecessary is immediately freed.

And also two disadvantages.

- The counter handling tends to be forgotten.
- When doing it naively cycles are not released.

I'll explain about the second point just in case. A cycle is a cycle of references as shown in Figure 8. If this is the case the counters will never decrease and the objects will never be released.

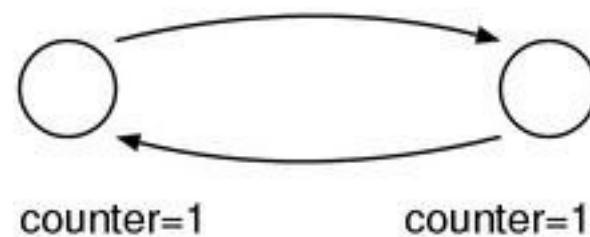


Figure 8: Cycle

By the way, latest Python(2.2) uses reference counting GC but it can free cycles. However, it is not because of the reference counting itself, but because it sometimes invokes mark and sweep GC to check.

## Object Management

---

Ruby's garbage collection is only concerned with ruby objects. Moreover, it only concerned with the objects created and managed by ruby. Conversely speaking, if the memory is allocated without following a certain procedure, it won't be taken care of. For instance, the following function will cause a memory leak even if ruby is running.

```
void not_ok()
{
    malloc(1024); /* receive memory and discard it */
}
```

However, the following function does not cause a memory leak.

```
void this_is_ok()
```

```
{  
    rb_ary_new(); /* create a ruby array and discard it */  
}
```

Since `rb_ary_new()` uses Ruby's proper interface to allocate memory, the created object is under the management of the GC of ruby, thus ruby will take care of it.

## ■ struct RVALUE

Since the substance of an object is a struct, managing objects means managing that structs. Of course the non-pointer objects like `Fixnum` `Symbol` `nil` `true` `false` are exceptions, but I won't always describe about it to prevent descriptions from being redundant.

Each struct type has its different size, but probably in order to keep management simpler, a union of all the structs of built-in classes is declared and the union is always used when dealing with memory. The declaration of that union is as follows.

### ▼ RVALUE

```
211  typedef struct RVALUE {  
212      union {  
213          struct {  
214              unsigned long flags; /* 0 if not used */  
215              struct RVALUE *next;  
216          } free;  
217          struct RBasic basic;  
218          struct RObject object;  
219          struct RClass klass;  
220          struct RFloat flonum;  
221          struct RString string;
```

```
222     struct RArray  array;
223     struct RRegexp regexp;
224     struct RHash   hash;
225     struct RData   data;
226     struct RStruct rstruct;
227     struct RBignum bignum;
228     struct RFile   file;
229     struct RNode   node;
230     struct RMatch  match;
231     struct RVarmap varmap;
232     struct SCOPE   scope;
233 } as;
234 } RVALUE;
```

(gc.c)

struct RVALUE is a struct that has only one element. I've heard that the reason why `union` is not directly used is to enable to easily increase its members when debugging or when extending in the future.

First, let's focus on the first element of the union `free.flags`. The comment says “`0` if not used”, but is it true? Is there not any possibility for `free.flags` to be `0` by chance?

As we've seen in Chapter 2: Objects, all object structs have `struct RBasic` as its first element. Therefore, by whichever element of the union we access, `obj->as.free.flags` means the same as it is written as `obj->as.basic.flags`. And objects always have the struct-type flag (such as `T_STRING`), and the flag is always not `0`. Therefore, the flag of an “alive” object will never coincidentally be `0`. Hence, we can confirm that setting their flags to `0` is necessity and sufficiency to represent “dead” objects.

# Object heap

The memory for all the object structs has been brought together in global variable `heaps`. Hereafter, let's call this an object heap.

## ▼ Object heap

```
239 #define HEAPS_INCREMENT 10
240 static RVALUE **heaps;
241 static int heaps_length = 0;
242 static int heaps_used = 0;
243
244 #define HEAP_MIN_SLOTS 10000
245 static int *heaps_limits;
246 static int heap_slots = HEAP_MIN_SLOTS;
```

(gc.c)

`heaps` is an array of arrays of `struct RVALUE`. Since it is `heaps`, the each contained array is probably each heap. Each element of `heap` is each slot (Figure 9).

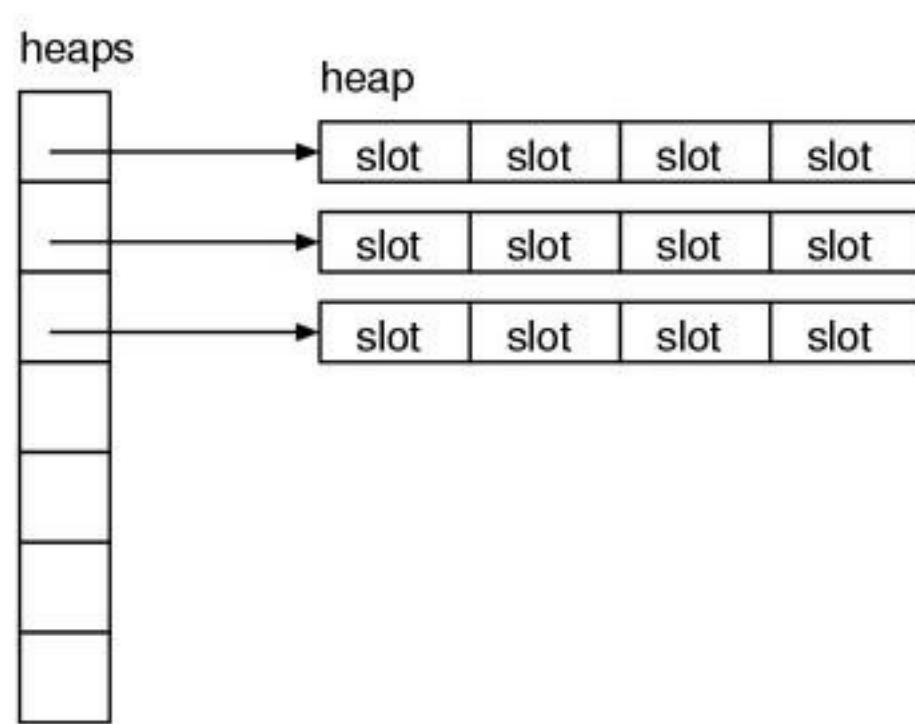


Figure 9: heaps, heap, slot

The length of `heaps` is `heap_length` and it can be changed. The number of the slots actually in use is `heaps_used`. The length of each heap is in the corresponding `heaps_limits[index]`. Figure 10 shows the structure of the object heap.

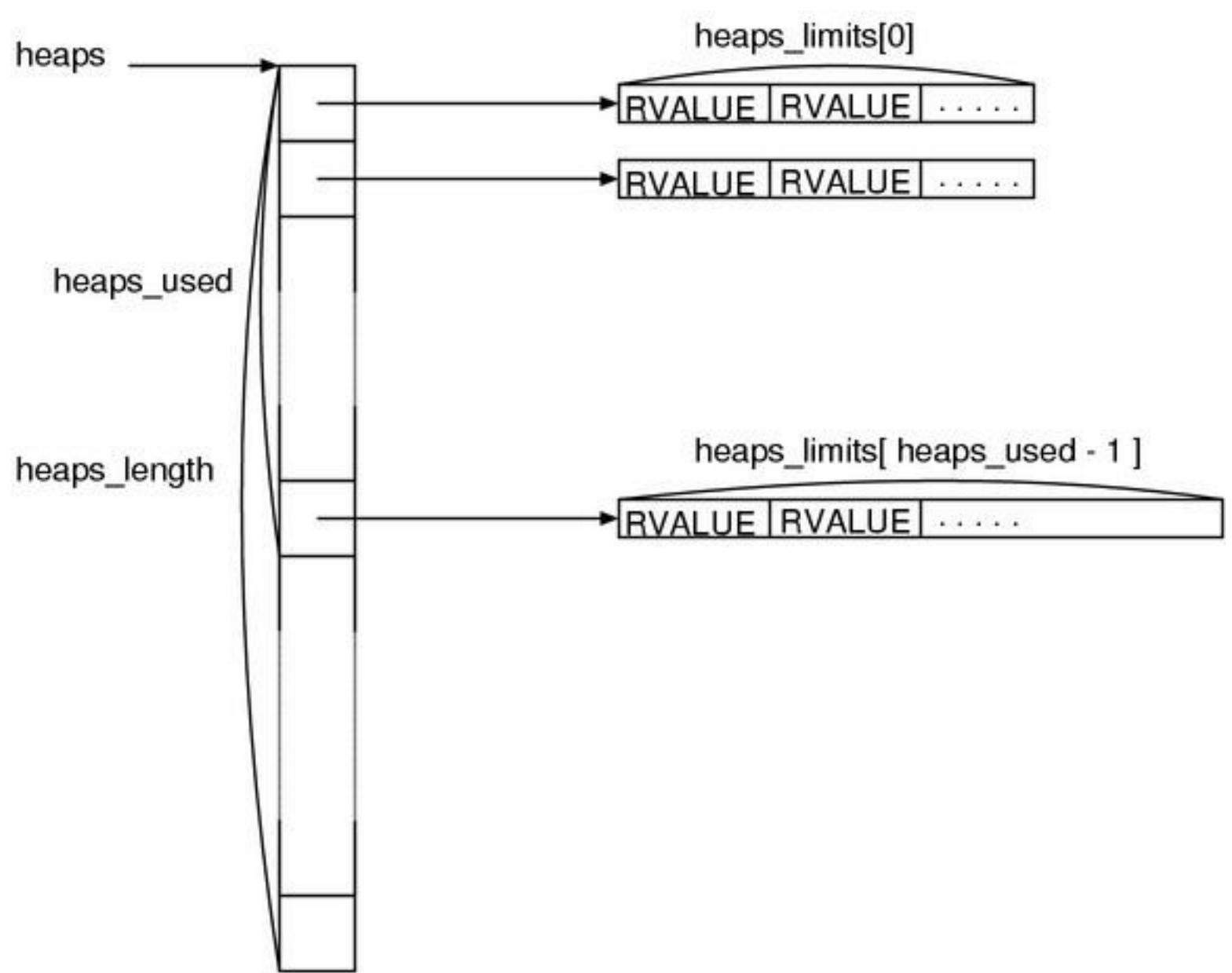


Figure 10: conceptual diagram of heaps in memory

This structure has a necessity to be this way. For instance, if all structs are stored in an array, the memory space would be the most compact, but we cannot do `realloc()` because it could change the addresses. This is because `VALUES` are mere pointers.

In the case of an implementation of Java, the counterpart of `VALUES` are not addresses but the indexes of objects. Since they are handled through a pointer table, objects are movable. However in this case, indexing of the array comes in every time an object access occurs

and it lowers the performance in some degree.

On the other hand, what happens if it is an one-dimensional array of pointers to RVALUES (it means VALUES)? This seems to be able to go well at the first glance, but it does not when GC. That is, as I'll describe in detail, the GC of ruby needs to know the integers "which seems VALUE (the pointers to RVALUE). If all RVALUE are allocated in addresses which are far from each other, it needs to compare all address of RVALUE with all integers "which could be pointers". This means the time for GC becomes the order more than  $O(n^2)$ , and not acceptable.

According to these requirements, it is good that the object heap form a structure that the addresses are cohesive to some extent and whose position and total amount are not restricted at the same time.

## freelist

Unused RVALUES are managed by being linked as a single line which is a linked list that starts with `freelist`. The `as.free.next` of RVALUE is the link used for this purpose.

### freelist

```
236 static RVALUE *freelist = 0;  
(gc.c)
```

## ■ add\_heap( )

As we understood the data structure, let's read the function `add_heap()` to add a heap. Because this function contains a lot of lines not part of the main line, I'll show the one simplified by omitting error handlings and castings.

### ▼ `add_heap()` (simplified)

```
static void
add_heap()
{
    RVALUE *p, *pend;

    /* extend heaps if necessary */
    if (heaps_used == heaps_length) {
        heaps_length += HEAPS_INCREMENT;
        heaps        = realloc(heaps,           heaps_length * sizeof(RVALUE));
        heaps_limits = realloc(heaps_limits, heaps_length * sizeof(RVALUE));
    }

    /* increase heaps by 1 */
    p = heaps[heaps_used] = malloc(sizeof(RVALUE) * heap_slots);
    heaps_limits[heaps_used] = heap_slots;
    pend = p + heap_slots;
    if (lomem == 0 || lomem > p) lomem = p;
    if (himem < pend) himem = pend;
    heaps_used++;
    heap_slots *= 1.8;

    /* link the allocated RVALUE to freelist */
    while (p < pend) {
        p->as.free.flags = 0;
        p->as.free.next = freelist;
        freelist = p;
        p++;
    }
}
```

Please check the following points.

- the length of `heap` is `heap_slots`
- the `heap_slots` becomes 1.8 times larger every time when a `heap` is added
- the length of `heaps[i]` (the value of `heap_slots` when creating a `heap`) is stored in `heaps_limits[i]`.

Plus, since `lomem` and `himem` are modified only by this function, only by this function you can understand the mechanism. These variables hold the lowest and the highest addresses of the object heap. These values are used later when determining the integers “which seems `VALUE`”.

## ■ `rb_newobj()`

Considering all of the above points, we can tell the way to create an object in a second. If there is at least a `RVALUE` linked from `freelist`, we can use it. Otherwise, do `GC` or increase the `heaps`. Let's confirm this by reading the `rb_newobj()` function to create an object.

## ▼ `rb_newobj()`

```
297  VALUE
298  rb_newobj()
299  {
300      VALUE obj;
301
302      if (!freelist) rb_gc();
303
304      obj = (VALUE)freelist;
```

```
305     freelist = freelist->as.free.next;
306     MEMZERO((void*)obj, RVALUE, 1);
307     return obj;
308 }
```

(gc.c)

If `freelist` is 0, in other words, if there's not any unused structs, invoke GC and create spaces. Even if we could not collect not any object, there's no problem because in this case a new space is allocated in `rb_gc()`. And take a struct from `freelist`, zerofill it by `MEMZERO()`, and return it.

## Mark

---

As described, ruby's GC is Mark & Sweep. Its “mark” is, concretely speaking, to set a `FL_MARK` flag: look for unused `VALUE`, set `FL_MARK` flags to found ones, then look at the object heap after investigating all and free objects that `FL_MARK` has not been set.

### rb\_gc\_mark()

`rb_gc_mark()` is the function to mark objects recursively.

#### ▼ rb\_gc\_mark()

```
573 void
574 rb_gc_mark(ptr)
```

```
575     VALUE ptr;
576 {
577     int ret;
578     register RVALUE *obj = RANY(ptr);
579
580     if (rb_special_const_p(ptr)) return; /* special const no
581     if (obj->as.basic.flags == 0) return;      /* free cell
582     if (obj->as.basic.flags & FL_MARK) return; /* already m
583
584     obj->as.basic.flags |= FL_MARK;
585
586     CHECK_STACK(ret);
587     if (ret) {
588         if (!mark_stack_overflow) {
589             if (mark_stack_ptr - mark_stack < MARK_STACK_MAX
590                 *mark_stack_ptr = ptr;
591                 mark_stack_ptr++;
592         }
593         else {
594             mark_stack_overflow = 1;
595         }
596     }
597 }
598 else {
599     rb_gc_mark_children(ptr);
600 }
601 }
```

(gc.c)

The definition of RANY() is as follows. It is not particularly important.

▼ RANY()

```
295 #define RANY(o) ((RVALUE*)(o))
(gc.c)
```

There are the checks for non-pointers or already freed objects and the recursive checks for marked objects at the beginning,

```
obj->as.basic.flags |= FL_MARK;
```

and `obj` (this is the `ptr` parameter of this function) is marked. Then next, it's the turn to follow the references from `obj` and mark. `rb_gc_mark_children()` does it.

The others, what starts with `CHECK_STACK()` and is written a lot is a device to prevent the machine stack overflow. Since `rb_gc_mark()` uses recursive calls to mark objects, if there is a big object cluster, it is possible to run short of the length of the machine stack. To counter that, if the machine stack is nearly overflow, it stops the recursive calls, piles up the objects on a global list, and later it marks them once again. This code is omitted because it is not part of the main line.

## ■ `rb_gc_mark_children()`

Now, as for `rb_gc_mark_children()`, it just lists up the internal types and marks one by one, thus it is not just long but also not interesting. Here, it is shown but the simple enumerations are omitted:

### ▼ `rb_gc_mark_children()`

```
603 void
604 rb_gc_mark_children(ptr)
```

```
605     VALUE ptr;
606 {
607     register RVALUE *obj = RANY(ptr);
608
609     if (FL_TEST(obj, FL_EXIVAR)) {
610         rb_mark_generic_ivar((VALUE)obj);
611     }
612
613     switch (obj->as.basic.flags & T_MASK) {
614         case T_NIL:
615         case T_FIXNUM:
616             rb_bug("rb_gc_mark() called for broken object");
617             break;
618
619         case T_NODE:
620             mark_source_filename(obj->as.node.nd_file);
621             switch (nd_type(obj)) {
622                 case NODE_IF:           /* 1,2,3 */
623                 case NODE_FOR:
624                 case NODE_ITER:
625                     /* ..... omitted ..... */
626             }
627             return; /* not need to mark basic.klass */
628         }
629
630         rb_gc_mark(obj->as.basic.klass);
631         switch (obj->as.basic.flags & T_MASK) {
632             case T_ICLASS:
633             case T_CLASS:
634             case T_MODULE:
635                 rb_gc_mark(obj->as.klass.super);
636                 rb_mark_tbl(obj->as.klass.m_tbl);
637                 rb_mark_tbl(obj->as.klass.iv_tbl);
638                 break;
639
640             case T_ARRAY:
641                 if (FL_TEST(obj, ELTS_SHARED)) {
642                     rb_gc_mark(obj->as.array.aux.shared);
643                 }
644                 else {
645                     long i, len = obj->as.array.len;
646                     VALUE *ptr = obj->as.array.ptr;
647
648                     for (i = 0; i < len; i++) {
649                         rb_gc_mark(ptr[i]);
650                     }
651
652                     rb_gc_mark(obj->as.array.aux.shared);
653                 }
654             }
655         }
656     }
657 }
```

```
771                 for (i=0; i < len; i++) {
772                     rb_gc_mark(*ptr++);
773                 }
774             }
775             break;

    /* ..... omitted ..... */

837         default:
838             rb_bug("rb_gc_mark(): unknown data type 0x%x(0x%x) %
839                     obj->as.basic.flags & T_MASK, obj,
840                     is_pointer_to_heap(obj) ? "corrupted object"
841                     : "non object");
842     }
843 }
```

(gc.c)

It calls `rb_gc_mark()` recursively, is only what I'd like you to confirm. In the omitted part, `NODE` and `T_xxxx` are enumerated respectively. `NODE` will be introduced in Part 2.

Additionally, let's see the part to mark `T_DATA` (the struct used for extension libraries) because there's something we'd like to check. This code is extracted from the second `switch` statement.

## ▼ `rb_gc_mark_children() - T_DATA`

```
789         case T_DATA:
790             if (obj->as.data.dmark) (*obj->as.data.dmark)(DATA_P
791             break;

(gc.c)
```

Here, it does not use `rb_gc_mark()` or similar functions, but the

dmark which is given from users. Inside it, of course, it might use `rb_gc_mark()` or something, but not using is also possible. For example, in an extreme situation, if a user defined object does not contain `VALUE`, there's no need to mark.

## ► rb\_gc()

By now, we've finished to talk about each object. From now on, let's see the function `rb_gc()` that presides the whole. The objects marked here are “objects which are obviously necessary”. In other words, “the roots of GC”.

## ▼ rb\_gc()

```
1110 void
1111 rb_gc()
1112 {
1113     struct gc_list *list;
1114     struct FRAME *volatile frame; /* gcc 2.7.2.3 -O2 bug??
1115     jmp_buf save_regs_gc_mark;
1116     SET_STACK_END;
1117
1118     if (dont_gc || during_gc) {
1119         if (!freelist) {
1120             add_heap();
1121         }
1122         return;
1123     }
1124
1125     /* ..... mark from the all roots ..... */
1126
1127     gc_sweep();
1128 }
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184 }
```

(gc.c)

The roots which should be marked will be shown one by one after this, but I'd like to mention just one point here.

In `ruby` the CPU registers and the machine stack are also the roots. It means that the local variables and arguments of C are automatically marked. For example,

```
static int
f(void)
{
    VALUE arr = rb_ary_new();
    /* ..... do various things ..... */
}
```

like this way, we can protect an object just by putting it into a variable. This is a very significant trait of the GC of `ruby`. Because of this feature, `ruby`'s extension libraries are insanely easy to write.

However, what is on the stack is not only `VALUE`. There are a lot of totally unrelated values. How to resolve this is the key when reading the implementation of GC.

## ▀ The Ruby Stack

First, it marks the (`ruby`'s) stack frames used by the interpreter. Since you will be able to find out who it is after reaching Part 3, you don't have to think so much about it for now.

### ▼ Marking the Ruby Stack

```
1130     /* mark frame stack */
1131     for (frame = ruby_frame; frame; frame = frame->prev) {
1132         rb_gc_mark_frame(frame);
1133         if (frame->tmp) {
1134             struct FRAME *tmp = frame->tmp;
1135             while (tmp) {
1136                 rb_gc_mark_frame(tmp);
1137                 tmp = tmp->prev;
1138             }
1139         }
1140     }
1141     rb_gc_mark((VALUE) ruby_class);
1142     rb_gc_mark((VALUE) ruby_scope);
1143     rb_gc_mark((VALUE) ruby_dyna_vars);

(gc.c)
```

`ruby_frame` `ruby_class` `ruby_scope` `ruby_dyna_vars` are the variables to point to each top of the stacks of the evaluator. These hold the frame, the class scope, the local variable scope, and the block local variables at that time respectively.

## Register

Next, it marks the CPU registers.

### ▼ marking the registers

```
1148     FLUSH_REGISTER_WINDOWS;
1149     /* Here, all registers must be saved into jmp_buf. */
1150     setjmp(save_regs_gc_mark);
1151     mark_locations_array((VALUE*)save_regs_gc_mark,
1152                           sizeof(save_regs_gc_mark) / sizeof(
```

(gc.c)

FLUSH\_REGISTER\_WINDOWS is special. We will see it later.

setjmp() is essentially a function to remotely jump, but the content of the registers are saved into the argument (which is a variable of type jmp\_buf) as its side effect. Making use of this, it attempts to mark the content of the registers. Things around here really look like secret techniques.

However only djgpp and Human68k are specially treated. djgpp is a gcc environment for DOS. Human68k is an OS of SHARP X680X0 Series. In these two environments, the whole registers seem to be not saved only by the ordinary setjmp(), setjmp() is redefined as follows as an inline-assembler to explicitly write out the registers.

## ▼ the original version of setjmp

```
1072 #ifdef __GNUC__  
1073 #if defined(__human68k__) || defined(DJGPP)  
1074 #if defined(__human68k__)  
1075     typedef unsigned long rb_jmp_buf[8];  
1076     __asm__ (".even\n\"  
1077     _rb_setjmp:\n\"  
1078             move.l 4(sp),a0\n\"  
1079             movem.l d3-d7/a3-a5,(a0)\n\"  
1080             moveq.l #0,d0\n\"  
1081             rts");  
1082 #ifdef setjmp  
1083 #undef setjmp  
1084 #endif  
1085 #else  
1086 #if defined(DJGPP)  
1087     typedef unsigned long rb_jmp_buf[6];  
1088     __asm__ (".align 4\n\"  
1089     _rb_setjmp:\n\"  
1090             pushl  %ebp\n\"  
1091             movl  %ebp,d0\n\"  
1092             movem.l d1-d5,d0\n\"  
1093             moveq.l #0,d1\n\"  
1094             rts");  
1095 #endif  
1096 #endif  
1097 #endif  
1098 #endif  
1099 #endif
```

2-byte alignment  
the label of rb\_setjmp()  
load the first argument  
copy the registers to w  
set 0 to d0 (as the ret  
return  
order 4-byte alignment  
the label for rb\_setjmp  
push ebp to the stack

```

1091     movl    %esp,%ebp\n\
1092     movl    8(%ebp),%ebp\n\
1093     movl    %eax,(%ebp)\n\
1094     movl    %ebx,4(%ebp)\n\
1095     movl    %ecx,8(%ebp)\n\
1096     movl    %edx,12(%ebp)\n\
1097     movl    %esi,16(%ebp)\n\
1098     movl    %edi,20(%ebp)\n\
1099     popl    %ebp\n\
1100     xorl    %eax,%eax\n\
1101     ret");
1102 #endif
1103 #endif
1104 int rb_setjmp (rb_jmp_buf);
1105 #define jmp_buf rb_jmp_buf
1106 #define setjmp rb_setjmp
1107 #endif /* __human68k__ or DJGPP */
1108 #endif /* __GNUC__ */

```

(gc.c)

set the stack pointer to where ebp points to  
pick up the first argument in the followings, store to where ebp points to  
restore ebp from the stack  
set 0 to eax (as the return value)

Alignment is the constraint when putting variables on memories. For example, in 32-bit machine `int` is usually 32 bits, but we cannot always take 32 bits from anywhere of memories. Particularly, RISC machine has strict constraints, it is decided like “from a multiple of 4 byte” or “from even byte”. When there are such constraints, memory access unit can be more simplified (thus, it can be faster). When there’s the constraint of “from a multiple of 4 byte”, it is called “4-byte alignment”.

Plus, in `cc` of djgpp or Human68k, there’s a rule that the compiler put the underline to the head of each function name. Therefore, when writing a C function in Assembler, we need to put the underline (`_`) to its head by ourselves. This type of constraints are techniques in order to avoid the conflicts in names with library

functions. Also in UNIX, it is said that the underline had been attached by some time ago, but it almost disappears now.

Now, the content of the registers has been able to be written out into `jmp_buf`, it will be marked in the next code:

▼ mark the registers (shown again)

```
1151     mark_locations_array((VALUE*)save_regs_gc_mark,  
                           sizeof(save_regs_gc_mark) / sizeof(  
(gc.c))
```

This is the first time that `mark_locations_array()` appears. I'll describe it in the next section.

## mark\_locations\_array()

▼ `mark_locations_array()`

```
500 static void  
501 mark_locations_array(x, n)  
502     register VALUE **x;  
503     register long n;  
504 {  
505     while (n--) {  
506         if (is_pointer_to_heap((void *)*x)) {  
507             rb_gc_mark(*x);  
508         }  
509         x++;  
510     }  
511 }
```

(gc.c)

This function is to mark the all elements of an array, but it slightly differs from the previous mark functions. Until now, each place to be marked is where we know it surely holds a **VALUE** (a pointer to an object). However this time, where it attempts to mark is the register space, it is enough to expect that there're also what are not **VALUE**. To counter that, it tries to detect whether or not the value is a **VALUE** (a pointer), then if it seems, the value will be handled as a pointer. This kind of methods are called “conservative GC”. It seems that it is conservative because it “tentatively inclines things to the safe side”

Next, we'll look at the function to check if “it looks like a **VALUE**”, it is `is_pointer_to_heap()`.

## is\_pointer\_to\_heap()

### ▼ `is_pointer_to_heap()`

```
480 static inline int
481 is_pointer_to_heap(ptr)
482     void *ptr;
483 {
484     register RVALUE *p = RANY(ptr);
485     register RVALUE *heap_org;
486     register long i;
487
488     if (p < lomem || p > himem) return Qfalse;
489
490     /* check if there's the possibility that p is a pointer
491      for (i=0; i < heaps_used; i++) {
492          heap_org = heaps[i];
493          if (heap_org <= p && p < heap_org + heaps_limits[i]
494              (((char*)p)-((char*)heap_org))%sizeof(RVALUE))
```

```
495             return Qtrue;
496         }
497         return Qfalse;
498     }
```

(gc.c)

If I briefly explain it, it would look like the followings:

- check if it is in between the top and the bottom of the addresses where RVALUES reside.
- check if it is in the range of a heap
- make sure the value points to the head of a RVALUE.

Since the mechanism is like this, it's obviously possible that a non-VALUE value is mistakenly handled as a VALUE. But at least, it will never fail to find out the used VALUES. And, with this amount of tests, it may rarely pick up a non-VALUE value unless it intentionally does. Therefore, considering about the benefits we can obtain by GC, it's sufficient to compromise.

## Register Window

This section is about `FLUSH_REGISTER_WINDOWS()` which has been deferred.

Register windows are the mechanism to enable to put a part of the machine stack into inside the CPU. In short, it is a cache whose purpose of use is narrowed down. Recently, it exists only in Sparc architecture. It's possible that there are also VALUES in register

windows, and it's also necessary to get down them into memory.

The content of the macro is like this:

### ▼ FLUSH\_REGISTER\_WINDOWS

```
125 #if defined(sparc) || defined(__sparc__)
126 # if defined(linux) || defined(__linux__)
127 #define FLUSH_REGISTER_WINDOWS asm("ta 0x83")
128 # else /* Solaris, not sparc linux */
129 #define FLUSH_REGISTER_WINDOWS asm("ta 0x03")
130 #endif
131 #else /* Not a sparc */
132 #define FLUSH_REGISTER_WINDOWS
133 #endif
```

(defines.h)

asm(...) is a built-in assembler. However, even though I call it assembler, this instruction named ta is the call of a privileged instruction. In other words, the call is not of the CPU but of the OS. That's why the instruction is different for each OS. The comments describe only about Linux and Solaris, but actually FreeBSD and NetBSD are also works on Sparc, so this comment is wrong.

Plus, if it is not Sparc, it is unnecessary to flush, thus FLUSH\_REGISTER\_WINDOWS is defined as nothing. Like this, the method to get a macro back to nothing is very famous technique that is also convenient when debugging.

## Machine Stack

Then, let's go back to the rest of `rb_gc()`. This time, it marks `VALUES` in the machine stack.

## ▼ mark the machine stack

```
1152     rb_gc_mark_locations(rb_gc_stack_start, (VALUE*)STACK_EN
1153 #if defined(__human68k__)
1154     rb_gc_mark_locations((VALUE*)((char*)rb_gc_stack_start +
1155                               (VALUE*)((char*)STACK_END + 2));
1156 #endif
(gc.c)
```

`rb_gc_stack_start` seems the start address (the end of the stack) and `STACK_END` seems the end address (the top). And, `rb_gc_mark_locations()` practically marks the stack space.

There are `rb_gc_mark_locations()` two times in order to deal with the architectures which are not 4-byte alignment.

`rb_gc_mark_locations()` tries to mark for each portion of `sizeof(VALUE)`, so if it is in 2-byte alignment environment, sometimes not be able to properly mark. In this case, it moves the range 2 bytes then marks again.

Now, `rb_gc_stack_start`, `STACK_END`, `rb_gc_mark_locations()`, let's examine these three in this order.

## Init\_stack()

The first thing is `rb_gc_starck_start`. This variable is set only during `Init_stack()`. As the name `Init_` might suggest, this function is

called at the time when initializing the ruby interpreter.

## ▼ Init\_stack()

```
1193 void
1194 Init_stack(addr)
1195     VALUE *addr;
1196 {
1197 #if defined(__human68k__)
1198     extern void *_SEND;
1199     rb_gc_stack_start = _SEND;
1200 #else
1201     VALUE start;
1202
1203     if (!addr) addr = &start;
1204     rb_gc_stack_start = addr;
1205 #endif
1206 #ifdef HAVE_GETRLIMIT
1207 {
1208     struct rlimit rlim;
1209
1210     if (getrlimit(RLIMIT_STACK, &rlim) == 0) {
1211         double space = (double)rlim.rlim_cur*0.2;
1212
1213         if (space > 1024*1024) space = 1024*1024;
1214         STACK_LEVEL_MAX = (rlim.rlim_cur - space) / size
1215     }
1216 }
1217 #endif
1218 }
```

(gc.c)

What is important is only the part in the middle. It defines an arbitrary local variable (it is allocated on the stack) and it sets its address to `rb_gc_stack_start`. The `_SEND` inside the code for `__human68k__` is probably the variable defined by a library of

compiler or system. Naturally, you can presume that it is the contraction of Stack END.

Meanwhile, the code after that bundled by HAVE\_GETRLIMIT appears to check the length of the stack and do mysterious things. This is also in the same context of what is done at rb\_gc\_mark\_children() to prevent the stack overflow. We can ignore this.

## STACK\_END

Next, we'll look at the STACK\_END which is the macro to detect the end of the stack.

### ▼ STACK\_END

```
345 #ifdef C_ALLOCA
346 # define SET_STACK_END VALUE stack_end; alloca(0);
347 # define STACK_END (&stack_end)
348 #else
349 # if defined(__GNUC__) && defined(USE_BUILTIN_FRAME_ADDRESS)
350 #  define SET_STACK_END VALUE *stack_end = __builtin_frame_
351 # else
352 #  define SET_STACK_END VALUE *stack_end = alloca(1)
353 # endif
354 # define STACK_END (stack_end)
355 #endif
```

(gc.c)

As there are three variations of SET\_STACK\_END, let's start with the bottom one. alloca() allocates a space at the end of the stack and returns it, so the return value and the end address of the stack should be very close. Hence, it considers the return value of

`alloca()` as an approximate value of the end of the stack.

Let's go back and look at the one at the top. When the macro `C_ALLOCA` is defined, `alloca()` is not natively defined, ... in other words, it indicates a compatible function is defined in C. I mentioned that in this case `alloca()` internally allocates memory by using `malloc()`. However, it does not help to get the position of the stack at all. To deal with this situation, it determines that the local variable `stack_end` of the currently executing function is close to the end of the stack and uses its address (`&stack_end`).

Plus, this code contains `alloca(0)` whose purpose is not easy to see. This has been a feature of the `alloca()` defined in C since early times, and it means “please check and free the unused space”. Since this is used when doing GC, it attempts to free the memory allocated with `alloca()` at the same time. But I think it's better to put it in another macro instead of mixing into such place ...

And at last, in the middle case, it is about `_builtin_frame_address()`. `_GNUC_` is a symbol defined in `gcc` (the compiler of GNU C). Since this is used to limit, it is a built-in instruction of `gcc`. You can get the address of the n-times previous stack frame with `_builtin_frame_address(n)`. As for `_builtin_frame_address(0)`, it provides the address of the current frame.

## `rb_gc_mark_locations()`

The last one is the `rb_gc_mark_locations()` function that actually

marks the stack.

## ▼ rb\_gc\_mark\_locations()

```
513 void
514 rb_gc_mark_locations(start, end)
515     VALUE *start, *end;
516 {
517     VALUE *tmp;
518     long n;
519
520     if (start > end) {
521         tmp = start;
522         start = end;
523         end = tmp;
524     }
525     n = end - start + 1;
526     mark_locations_array(start,n);
527 }
```

(gc.c)

Basically, delegating to the function `mark_locations_array()` which marks a space is sufficient. What this function does is properly adjusting the arguments. Such adjustment is required because in which direction the machine stack extends is undecided. If the machine stack extends to lower addresses, `end` is smaller, if it extends to higher addresses, `start` is smaller. Therefore, so that the smaller one becomes `start`, they are adjusted here.

## ▀ The other root objects

Finally, it marks the built-in `VALUE` containers of the interpreter.

## ▼ The other roots

```
1159     /* mark the registered global variables */
1160     for (list = global_List; list; list = list->next) {
1161         rb_gc_mark(*list->varptr);
1162     }
1163     rb_mark_end_proc();
1164     rb_gc_mark_global_tbl();
1165
1166     rb_mark_tbl(rb_class_tbl);
1167     rb_gc_mark_trap_list();
1168
1169     /* mark the instance variables of true, false, etc if ex
1170     rb_mark_generic_ivar_tbl();
1171
1172     /* mark the variables used in the ruby parser (only whil
1173     rb_gc_mark_parser();

(gc.c)
```

When putting a `VALUE` into a global variable of C, it is required to register its address by user via `rb_gc_register_address()`. As these objects are saved in `global_List`, all of them are marked.

`rb_mark_end_proc()` is to mark the procedural objects which are registered via kind of `END` statement of Ruby and executed when a program finishes. (`END` statements will not be described in this book).

`rb_gc_mark_global_tbl()` is to mark the global variable table `rb_global_tbl`. (See also the next chapter “Variables and Constants”)

`rb_mark_tbl(rb_class_tbl)` is to mark `rb_class_tbl` which was discussed in the previous chapter.

`rb_gc_mark_trap_list()` is to mark the procedural objects which are registered via the Ruby's function-like method `trap`. (This is related to signals and will also not be described in this book.)

`rb_mark_generic_ivar_tbl()` is to mark the instance variable table prepared for non-pointer `VALUE` such as `true`.

`rb_gc_mark_parser()` is to mark the semantic stack of the parser. (The semantic stack will be described in Part 2.)

Until here, the mark phase has been finished.

## Sweep

---

### ■ The special treatment for NODE

The sweep phase is the procedures to find out and free the not-marked objects. But, for some reason, the objects of type `T_NODE` are specially treated. Take a look at the next part:

▼ at the beginning of `gc_sweep()`

```
846 static void
847 gc_sweep()
848 {
849     RVALUE *p, *pend, *final_list;
850     int freed = 0;
851     int i, used = heaps_used;
```

```
852
853     if (ruby_in_compile && ruby_parser_stack_on_heap()) {
854         /* If the yacc stack is not on the machine stack,
855            do not collect NODE while parsing */
856         for (i = 0; i < used; i++) {
857             p = heaps[i]; pend = p + heaps_limits[i];
858             while (p < pend) {
859                 if (!(p->as.basic.flags & FL_MARK) &&
860                     BUILTIN_TYPE(p) == T_NODE)
861                     rb_gc_mark((VALUE)p);
862                 p++;
863             }
864         }
865     }
866 }
```

(gc.c)

NODE is a object to express a program in the parser. NODE is put on the stack prepared by a tool named yacc while compiling, but that stack is not always on the machine stack. Concretely speaking, when `ruby_parser_stack_on_heap()` is false, it indicates it is not on the machine stack. In this case, a NODE could be accidentally collected in the middle of its creation, thus the objects of type `T_NODE` are unconditionally marked and protected from being collected while compiling (`ruby_in_compile`).

## Finalizer

After it has reached here, all not-marked objects can be freed. However, there's one thing to do before freeing. In Ruby the freeing of objects can be hooked, and it is necessary to call them. This hook is called “finalizer”.

## ▼ gc\_sweep() Middle

```
869     freelist = 0;
870     final_list = deferred_final_list;
871     deferred_final_list = 0;
872     for (i = 0; i < used; i++) {
873         int n = 0;
874
875         p = heaps[i]; pend = p + heaps_limits[i];
876         while (p < pend) {
877             if (!(p->as.basic.flags & FL_MARK)) {
878 (A)                 if (p->as.basic.flags) {
879                     obj_free((VALUE)p);
880                 }
881 (B)                 if (need_call_final && FL_TEST(p, FL_FINALIZ
882                     p->as.free.flags = FL_MARK; /* remains m
883                     p->as.free.next = final_list;
884                     final_list = p;
885                 }
886                 else {
887                     p->as.free.flags = 0;
888                     p->as.free.next = freelist;
889                     freelist = p;
890                 }
891                 n++;
892             }
893 (C)             else if (RBASIC(p)->flags == FL_MARK) {
894                 /* the objects that need to finalize */
895                 /* are left untouched */
896             }
897             else {
898                 RBASIC(p)->flags &= ~FL_MARK;
899             }
900             p++;
901         }
902         freed += n;
903     }
904     if (freed < FREE_MIN) {
905         add_heap();
906     }
907     during_gc = 0;
```

This checks all over the object heap from the edge, and frees the object on which `FL_MARK` flag is not set by using `obj_free()` (A). `obj_free()` frees, for instance, only `char[]` used by `String` objects or `VALUE[]` used by `Array` objects, but it does not free the `RVALUE` struct and does not touch `basic.flags` at all. Therefore, if a struct is manipulated after `obj_free()` is called, there's no worry about going down.

After it frees the objects, it branches based on `FL_FINALIZE` flag (B). If `FL_FINALIZE` is set on an object, since it means at least a finalizer is defined on the object, the object is added to `final_list`. Otherwise, the object is immediately added to `freelist`. When finalizing, `basic.flags` becomes `FL_MARK`. The struct-type flag (such as `T_STRING`) is cleared because of this, and the object can be distinguished from alive objects.

Then, this phase completes by executing the all finalizers. Notice that the hooked objects have already died when calling the finalizers. It means that while executing the finalizers, one cannot use the hooked objects.

## ▼ `gc_sweep()` the rest

```

910      if (final_list) {
911          RVALUE *tmp;
912
913          if (rb_prohibit_interrupt || ruby_in_compile) {
914              deferred_final_list = final_list;

```

```
915             return;
916         }
917
918         for (p = final_list; p; p = tmp) {
919             tmp = p->as.free.next;
920             run_final((VALUE)p);
921             p->as.free.flags = 0;
922             p->as.free.next = freelist;
923             freelist = p;
924         }
925     }
926 }
```

(gc.c)

The `for` in the last half is the main finalizing procedure. The `if` in the first half is the case when the execution could not be moved to the Ruby program for various reasons. The objects whose finalization is deferred will be appear in the route (C) of the previous list.

## ■ rb\_gc\_force\_recycle()

I'll talk about a little different thing at the end. Until now, the ruby's garbage collector decides whether or not it collects each object, but there's also a way that users explicitly let it collect a particular object. It's `rb_gc_force_recycle()`.

## ▼ rb\_gc\_force\_recycle()

```
928 void
929 rb_gc_force_recycle(p)
930     VALUE p;
931 {
932     RANY(p)->as.free.flags = 0;
```

```
933     RANY(p)->as.free.next = freelist;
934     freelist = RANY(p);
935 }
```

(gc.c)

Its mechanism is not so special, but I introduced this because you'll see it several times in Part 2 and Part 3.

## Discussions

---

### ■ To free spaces

The space allocated by an individual object, say, `char[]` of `String`, is freed during the sweep phase, but the code to free the `RVALUE` struct itself has not appeared yet. And, the object heap also does not manage the number of structs in use and such. This means that if the ruby's object space is once allocated it would never be freed.

For example, the mailer what I'm creating now temporarily uses the space almost 40M bytes when constructing the threads for 500 mails, but if most of the space becomes unused as the consequence of GC it will keep occupying the 40M bytes. Because my machine is also kind of modern, it does not matter if just the 40M bytes are used. But, if this occurs in a server which keeps running, there's the possibility of becoming a problem.

However, one also need to consider that `free()` does not always

mean the decrease of the amount of memory in use. If it does not return memory to OS, the amount of memory in use of the process never decrease. And, depending on the implementation of `malloc()`, although doing `free()` it often does not cause returning memory to OS.

... I had written so, but just before the deadline of this book, RVALUE became to be freed. The attached CD-ROM also contains the edge ruby, so please check by `diff`. ... what a sad ending.

## ■ Generational GC

Mark & Sweep has an weak point, it is “it needs to touch the entire object space at least once”. There’s the possibility that using the idea of Generational GC can make up for the weak point.

The fundamental of Generational GC is the experiential rule that “Most objects are lasting for either very long or very short time”. You may be convinced about this point by thinking for seconds about the programs you write.

Then, thinking based on this rule, one may come up with the idea that “long-lived objects do not need to be marked or swept each and every time”. Once an object is thought that it will be long-lived, it is treated specially and excluded from the GC target. Then, for both marking and sweeping, it can significantly decrease the number of target objects. For example, if half of the objects are long-lived at a particular GC time, the number of the target objects

is half.

There's a problem, though. Generational GC is very difficult to do if objects can't be moved. It is because the long-lived objects are, as I just wrote, needed to "be treated specially". Since generational GC decreases the number of the objects dealt with and reduces the cost, if which generation a object belongs to is not clearly categorized, as a consequence it is equivalent to dealing with both generations. Furthermore, the `ruby`'s GC is also a conservative GC, so it also has to be created so that `is_pointer_to_heap()` work. This is particularly difficult.

How to solve this problem is ... By the hand of Mr. Kiyama Masato, the implementation of Generational GC for `ruby` has been published. I'll briefly describe how this patch deals with each problem. And this time, by courtesy of Mr. Kiyama, this Generational GC patch and its paper are contained in attached CD-ROM. (See also `doc/generational-gc.html`)

Then, I shall start the explanation. In order to ease explaining, from now on, the long-lived objects are called as "old-generation objects", the short-lived objects are called as "new-generation objects",

First, about the biggest problem which is the special treatment for the old-generation objects. This point is resolved by linking only the new-generation objects into a list named `newlist`. This list is substantialized by increasing `RVALUE`'s elements.

Second, about the way to detect the old-generation objects. It is very simply done by just removing the `newlist` objects which were not garbage collected from the `newlist`. In other words, once an object survives through GC, it will be treated as an old-generation object.

Third, about the way to detect the references from old-generation objects to new-generation objects. In Generational GC, it's sort of, the old-generation objects keep being in the marked state. However, when there are links from old-generation to new-generation, the new-generation objects will not be marked. (Figure 11)

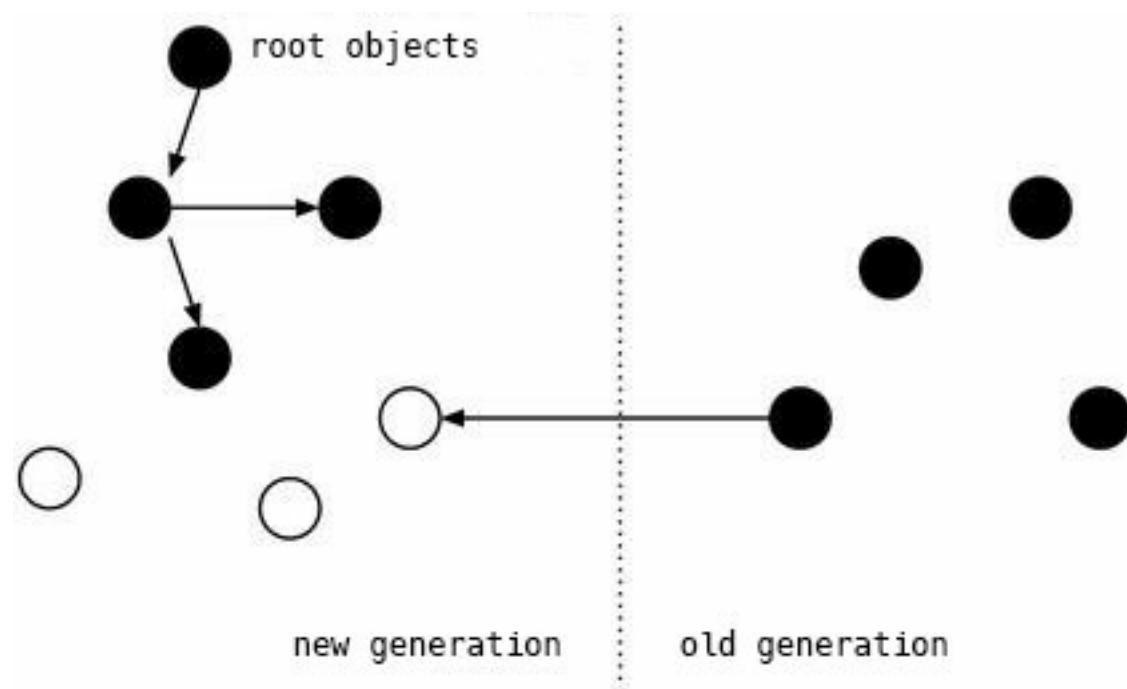


Figure 11: reference over generations

This is not good, so at the moment when an old-generational object refers to a new-generational object, the new-generational object must be turned into old-generational. The patch modifies the

libraries and adds checks to where there's possibility that this kind of references happens.

This is the outline of its mechanism. It was scheduled that this patch is included `ruby 1.7`, but it has not been included yet. It is said that the reason is its speed, There's an inference that the cost of the third point "check all references" matters, but the precise cause has not figured out.

## Compaction

Could the `ruby`'s GC do compaction? Since `VALUE` of `ruby` is a direct pointer to a struct, if the address of the struct are changed because of compaction, it is necessary to change the all `VALUES` that point to the moved structs.

However, since the `ruby`'s GC is a conservative GC, "the case when it is impossible to determine whether or not it is really a `VALUE`" is possible. Changing the value even though in this situation, if it was not `VALUE` something awful will happen. Compaction and conservative GC are really incompatible.

But, let's contrive countermeasures in one way or another. The first way is to let `VALUE` be an object ID instead of a pointer. (Figure 12) It means sandwiching a indirect layer between `VALUE` and a struct. In this way, as it's not necessary to rewrite `VALUE`, structs can be safely moved. But as trade-offs, accessing speed slows down and the compatibility of extension libraries is lost.

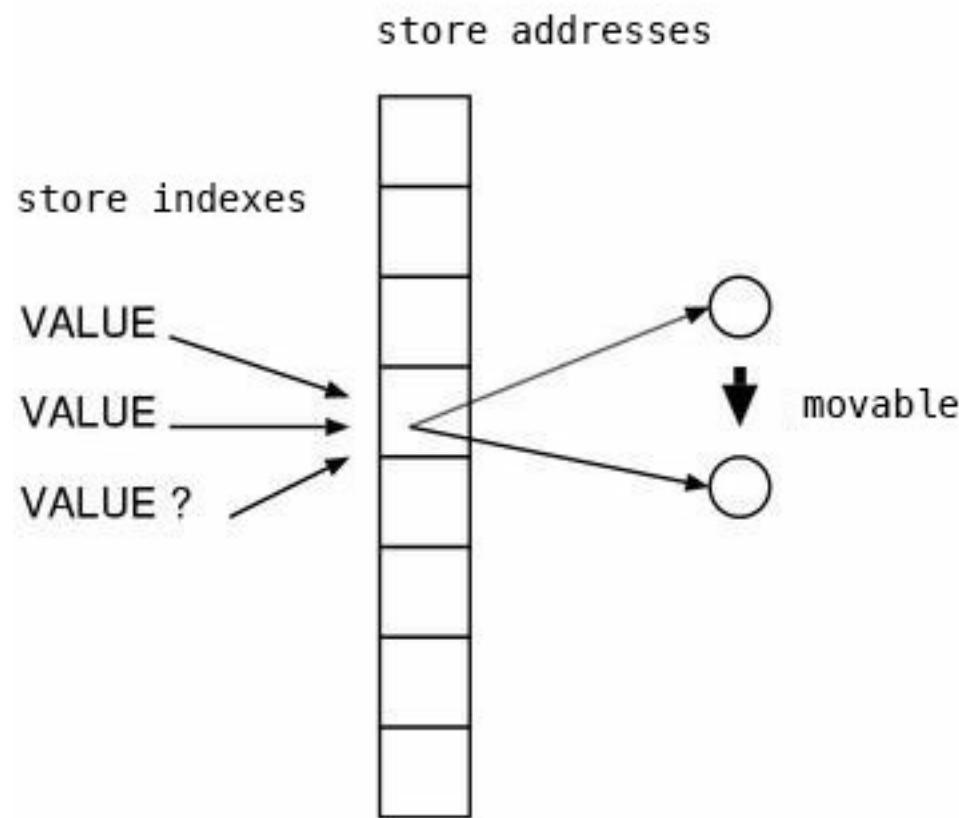


Figure 12: reference through the object ID

Then, the next way is to allow moving the struct only when they are pointed from only the pointers that “is surely VALUE” (Figure 13). This method is called Mostly-copying garbage collection. In the ordinary programs, there are not so many objects that `is_pointer_to_heap()` is true, so the probability of being able to move the object structs is quite high.

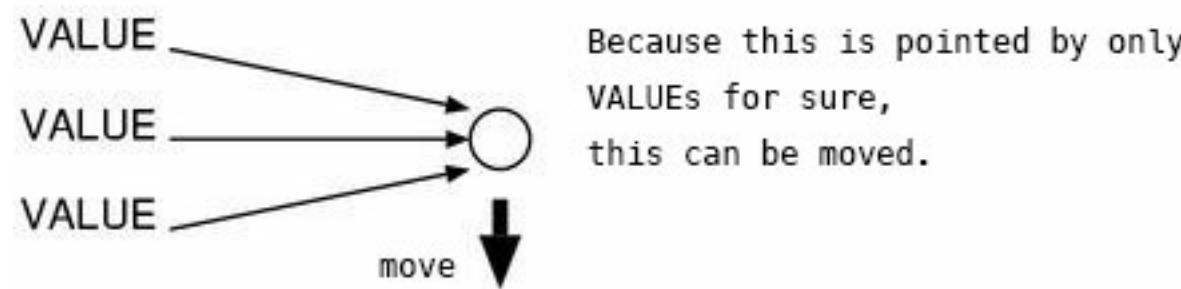
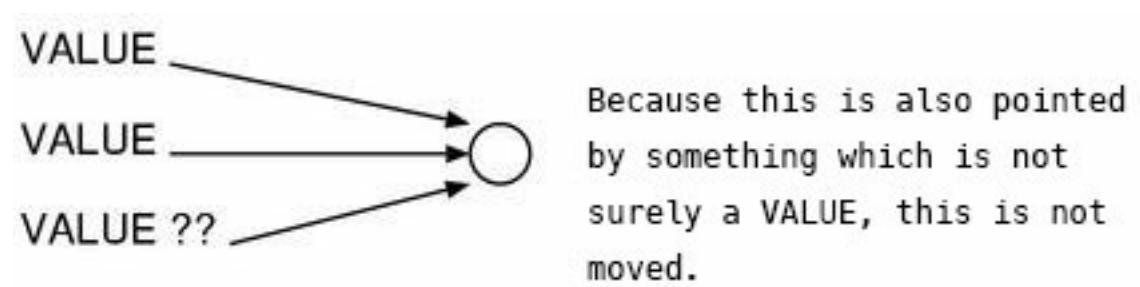


Figure 13: Mostly-copying garbage collection

Moreover and moreover, by enabling to move the struct, the implementation of Generational GC becomes simple at the same time. It seems to be worth to challenge.

## volatile to protect from GC

I wrote that GC takes care of VALUE on the stack, therefore if a VALUE is located as a local variable the VALUE should certainly be marked. But in reality due to the effects of optimization, it's possible that the variables disappear. For example, there's a possibility of disappearing in the following case:

```
VALUE str;
str = rb_str_new2("...");
printf("%s\n", RSTRING(str)->ptr);
```

Because this code does not access the str itself, some compilers

only keeps `str->ptr` in memory and deletes the `str`. If this happened, the `str` would be collected and the process would be down. There's no choice in this case

```
volatile VALUE str;
```

we need to write this way. `volatile` is a reserved word of C, and it has an effect of forbidding optimizations that have to do with this variable. If `volatile` was attached in the code relates to Ruby, you could assume almost certainly that its exists for GC. When I read K & R, I thought “what is the use of this?”, and totally didn't expect to see the plenty of them in `ruby`.

Considering these aspects, the promise of the conservative GC “users don't have to care about GC” seems not always true. There was once a discussion that “the Scheme's GC named KSM does not need `volatile`”, but it seems it could not be applied to `ruby` because its algorithm has a hole.

## When to invoke

---

### Inside `gc.c`

When to invoke GC? Inside `gc.c`, there are three places calling `rb_gc()` inside of `gc.c`,

- `ruby_xmalloc()`
- `ruby_xrealloc()`
- `rb_newobj()`

As for `ruby_xmalloc()` and `ruby_xrealloc()`, it is when failing to allocate memory. Doing GC may free memories and it's possible that a space becomes available again. `rb_newobj()` has a similar situation, it invokes when `freelist` becomes empty.

## Inside the interpreter

There's several places except for `gc.c` where calling `rb_gc()` in the interpreter.

First, in `io.c` and `dir.c`, when it runs out of file descriptors and could not open, it invokes GC. If `IO` objects are garbage collected, it's possible that the files are closed and file descriptors become available.

In `ruby.c`, `rb_gc()` is sometimes done after loading a file. As I mentioned in the previous Sweep section, it is to compensate for the fact that `NODE` cannot be garbage collected while compiling.

## Object Creation

---

We've finished about GC and come to be able to deal with the Ruby objects from its creation to its freeing. So I'd like to describe about object creations here. This is not so related to GC, rather, it is related a little to the discussion about classes in the previous chapter.

## Allocation Framework

We've created objects many times. For example, in this way:

```
class C
end
C.new()
```

At this time, how does `C.new` create a object?

First, `C.new` is actually `Class#new`. Its actual body is this:

### ▼ `rb_class_new_instance()`

```
725  VALUE
726  rb_class_new_instance(argc, argv, klass)
727      int argc;
728      VALUE *argv;
729      VALUE klass;
730  {
731      VALUE obj;
732
733      obj = rb_obj_alloc(klass);
734      rb_obj_call_init(obj, argc, argv);
735
736      return obj;
737  }
```

`rb_obj_alloc()` calls the `allocate` method against the `klass`. In other words, it calls `C.allocate` in this example currently explained. It is `Class#allocate` by default and its actual body is `rb_class_allocate_instance()`.

### ▼ `rb_class_allocate_instance()`

```

708 static VALUE
709 rb_class_allocate_instance(klass)
710     VALUE klass;
711 {
712     if (FL_TEST(klass, FL_SINGLETON)) {
713         rb_raise(rb_eTypeError,
714                 "can't create instance of virtual class");
715     if (rb_frame_last_func() != alloc) {
716         return rb_obj_alloc(klass);
717     }
718     else {
719         NEWOBJ(obj, struct RObject);
720         OBJSETUP(obj, klass, T_OBJECT);
721         return (VALUE)obj;
722     }
723 }
```

`rb_newobj()` is a function that returns a `RVALUE` by taking from the freelist. `NEWHOBJ()` is just a `rb_newobj()` with type-casting. The `OBJSETUP()` is a macro to initialize the `struct RBasic` part, you can think that this exists only in order not to forget to set the `FL_TAINT` flag.

The rest is going back to `rb_class_new_instance()`, then it calls `rb_obj_call_init()`. This function calls `initialize` on the just created object, and the initialization completes.

This is summarized as follows:

```
SomeClass.new          = Class#new (rb_class_new_instance)
SomeClass.allocate     = Class#allocate (rb_class_allocate)
SomeClass#initialize   = Object#initialize (rb_obj_dummy)
```

I could say that the `allocate` class method is to physically initialize, the `initialize` is to logically initialize. The mechanism like this, in other words the mechanism that an object creation is divided into `allocate` / `initialize` and `new` presides them, is called the “allocation framework”.

## Creating User Defined Objects

Next, we'll examine about the instance creations of the classes defined in extension libraries. As it is called user-defined, its struct is not decided, without telling how to allocate it, ruby don't understand how to create its object. Let's look at how to tell it.

### Data\_Wrap\_Struct()

Whichever it is user-defined or not, its creation mechanism itself can follow the allocation framework. It means that when defining a new `SomeClass` class in C, we overwrite both `SomeClass.allocate` and `SomeClass#initialize`.

Let's look at the `allocate` side first. Here, it does the physical initialization. What is necessary to allocate? I mentioned that the instance of the user-defined class is a pair of `struct RData` and a user-prepared struct. We'll assume that the struct is of type `struct my`. In order to create a `VALUE` based on the `struct my`, you can use `Data_Wrap_Struct()`. This is how to use:

```
struct my *ptr = malloc(sizeof(struct my)); /* arbitrarily alloc
VALUE val = Data_Wrap_Struct(data_class, mark_f, free_f, ptr);
```

`data_class` is the class that `val` belongs to, `ptr` is the pointer to be wrapped. `mark_f` is (the pointer to) the function to mark this struct. However, this does not mark the `ptr` itself and is used when the struct pointed by `ptr` contains `VALUE`. On the other hand, `free_f` is the function to free the `ptr` itself. The argument of the both functions is `ptr`. Going back a little and reading the code to mark may help you to understand things around here in one shot.

Let's also look at the content of `Data_Wrap_Struct()`.

### ▼ `Data_Wrap_Struct()`

```
369 #define Data_Wrap_Struct(klass, mark, free, sval) \
370     rb_data_object_alloc(klass, sval, \
                           (RUBY_DATA_FUNC)mark, \
                           (RUBY_DATA_FUNC)free)

365 typedef void (*RUBY_DATA_FUNC) _((void*));
(ruby.h)
```

Most of it is delegated to `rb_object_alloc()`.

## ▼ `rb_data_object_alloc()`

```
310 VALUE
311 rb_data_object_alloc(klass, datap, dmark, dfree)
312     VALUE klass;
313     void *datap;
314     RUBY_DATA_FUNC dmark;
315     RUBY_DATA_FUNC dfree;
316 {
317     NEWOBJ(data, struct RData);
318     OBJSETUP(data, klass, T_DATA);
319     data->data = datap;
320     data->dfree = dfree;
321     data->dmark = dmark;
322
323     return (VALUE)data;
324 }
```

(gc.c)

This is not complicated. As the same as the ordinary objects, it prepares a `RVALUE` by using `NEWWOBJ()` `OBJSETUP()`, and sets the members.

Here, let's go back to `allocate`. We've succeeded to create a `VALUE` by now, so the rest is putting it in an arbitrary function and defining the function on a class by `rb_define_singleton_method()`.

## `Data_Get_Struct()`

The next thing is `initialize`. Not only for `initialize`, the methods need a way to pull out the `struct my*` from the previously created

VALUE. In order to do it, you can use the `Data_Get_Struct()` macro.

## ▼ `Data_Get_Struct()`

```
378 #define Data_Get_Struct(obj,type,sval) do {\  
379     Check_Type(obj, T_DATA); \  
380     sval = (type*)DATA_PTR(obj);\  
381 } while (0)  
  
360 #define DATA_PTR(dta) (RDATA(dta)->data)  
(ruby.h)
```

As you see, it just takes the pointer (to `struct my`) from a member of `RData`. This is simple. `Check_Type()` just checks the struct type.

## ■ The Issues of the Allocation Framework

So, I've explained innocently until now, but actually the current allocation framework has a fatal issue. I just described that the object created with `allocate` appears to the `initialize` or the other methods, but if the passed object that was created with `allocate` is not of the same class, it must be a very serious problem. For example, if the object created with the default `Objct.allocate` (`Class#allocate`) is passed to the method of `String`, this cause a serious problem. That is because even though the methods of `String` are written based on the assumption that a struct of type `struct RString` is given, the given object is actually a `struct RObject`. In order to avoid such situation, the object created with `C.allocate` must be passed only to the methods of `C` or its subclasses.

Of course, this is always true when things are ordinarily done. As `c.allocate` creates the instance of the class `c`, it is not passed to the methods of the other classes. As an exception, it is possible that it is passed to the method of `Object`, but the methods of `Object` does not depend on the struct type.

However, what if it is not ordinarily done? Since `c.allocate` is exposed at the Ruby level, though I've not described about them yet, by making use of `alias` or `super` or something, the definition of `allocate` can be moved to another class. In this way, you can create an object whose class is `String` but whose actual struct type is `struct RObject`. It means that you can freely let `ruby` down from the Ruby level. This is a problem.

The source of the issue is that `allocate` is exposed to the Ruby level as a method. Conversely speaking, a solution is to define the content of `allocate` on the class by using a way that is anything but a method. So,

```
rb_define_allocator(rb_cMy, my_allocate);
```

an alternative like this is currently in discussion.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

# License

# Ruby Hacking Guide

Translated by Vincent ISAMBART

# Chapter 6: Variables and constants

## Outline of this chapter

---

### ■ Ruby variables

In Ruby there are quite a lot of different types of variables and constants. Let's line them up, starting from the largest scope.

- Global variables
- Constants
- Class variables
- Instance variables
- Local variables

Instance variables were already explained in chapter 2 “Objects”. In this chapter we'll talk about:

- Global variables
- Class variables
- Constants

We will talk about local variables in the third part of the book.

## API for variables

The object of this chapter's analysis is `variable.c`. Let me first introduce the APIs which would be the entry points.

```
VALUE rb_iv_get(VALUE obj, char *name)
VALUE rb_ivar_get(VALUE obj, ID name)
VALUE rb_iv_set(VALUE obj, char *name, VALUE val)
VALUE rb_ivar_set(VALUE obj, ID name, VALUE val)
```

These are the APIs to access instance variables which have already been described. They are shown here again because their definitions are in `variable.c`.

```
VALUE rb_cv_get(VALUE klass, char *name)
VALUE rb_cvar_get(VALUE klass, ID name)
VALUE rb_cv_set(VALUE klass, char *name, VALUE val)
VALUE rb_cvar_set(VALUE klass, ID name, VALUE val)
```

These functions are the API for accessing class variables. Class variables belong directly to classes so the functions take a class as parameter. There are in two groups, depending if their name starts with `rb_Xv` or `rb_Xvar`. The difference lies in the type of the variable “name”. The ones with a shorter name are generally easier to use because they take a `char*`. The ones with a longer name are more for internal use as they take a `ID`.

```
VALUE rb_const_get(VALUE klass, ID name)
VALUE rb_const_get_at(VALUE klass, ID name)
```

```
VALUE rb_const_set(VALUE klass, ID name, VALUE val)
```

These functions are for accessing constants. Constants also belong to classes so they take classes as parameter. `rb_const_get()` follows the superclass chain, whereas `rb_const_get_at()` does not (it just looks in `klass`).

```
struct global_entry *rb_global_entry(ID name)
VALUE rb_gv_get(char *name)
VALUE rb_gvar_get(struct global_entry *ent)
VALUE rb_gv_set(char *name, VALUE val)
VALUE rb_gvar_set(struct global_entry *ent, VALUE val)
```

These last functions are for accessing global variables. They are a little different from the others due to the use of `struct global_entry`. We'll explain this while describing the implementation.

## Points of this chapter

The most important point when talking about variables is “Where and how are variables stored?”, in other words: data structures.

The second most important matter is how we search for the values. The scopes of Ruby variables and constants are quite complicated because variables and constants are sometimes inherited, sometimes looked for outside of the local scope... To have a better understanding, you should think by comparing the implementation with the specification, like “It behaves like this in this situation so its implementation couldn't be other then this!”

# Class variables

Class variables are variables that belong to classes. In Java or C++ they are called static variables. They can be accessed from both the class or its instances. But “from an instance” or “from the class” is information only available in the evaluator, and we do not have one for the moment. So from the C level it’s like having no access range. We’ll just focus on the way these variables are stored.

## Reading

The functions to get a class variable are `rb_cvar_get()` and `rb_cv_get()`. The function with the longer name takes `ID` as parameter and the one with the shorter one takes `char*`. Because the one taking an `ID` seems closer to the internals, we’ll look at it.

### ▼ `rb_cvar_get()`

```
1508 VALUE
1509 rb_cvar_get(klass, id)
1510     VALUE klass;
1511     ID id;
1512 {
1513     VALUE value;
1514     VALUE tmp;
1515
1516     tmp = klass;
1517     while (tmp) {
1518         if (RCLASS(tmp)->iv_tbl) {
1519             if (st_lookup(RCLASS(tmp)->iv_tbl, id, &value)) {
1520                 if (RTEST(ruby_verbose)) {
1521                     cvar_override_check(id, tmp);
1522                 }
1523             }
1524         }
1525     }
1526 }
```

```
1523             return value;
1524         }
1525     }
1526     tmp = RCLASS(tmp)->super;
1527 }
1528
1529     rb_name_error(id,"uninitialized class variable %s in %s"
1530                 rb_id2name(id), rb_class2name(klass));
1531     return Qnil;           /* not reached */
1532 }
```

(variable.c)

This function reads a class variable in `klass`.

Error management functions like `rb_raise()` can be simply ignored like I said before. The `rb_name_error()` that appears this time is a function for raising an exception, so it can be ignored for the same reasons. In `ruby`, you can assume that all functions ending with `_error` raise an exception.

After removing all this, we can see that it is just following the `klass`'s superclass chain one by one and searching in each `iv_tbl`.... At this point, I'd like you to say “What? `iv_tbl` is the instance variables table, isn't it?” As a matter of fact, class variables are stored in the instance variable table.

We can do this because when creating `IDS`, the whole name of the variables is taken into account, including the prefix: `rb_intern()` will return different `IDS` for “`@var`” and “`@@var`”. At the Ruby level, the variable type is determined only by the prefix so there's no way to access a class variable called `@var` from Ruby.

# Constants

---

It's a little abrupt but I'd like you to remember the members of `struct RClass`. If we exclude the `basic` member, `struct RClass` contains:

- `VALUE super`
- `struct st_table *iv_tbl`
- `struct st_table *m_tbl`

Then, considering that:

1. constants belong to a class
2. we can't see any table dedicated to constants in `struct RClass`
3. class variables and instance variables are both in `iv_tbl`

Could it mean that the constants are also...

## Assignment

`rb_const_set()` is a function to set the value of constants: it sets the constant `id` in the class `klass` to the value `val`.

### ▼ `rb_const_set()`

```
1377 void
1378 rb_const_set(klass, id, val)
1379     VALUE klass;
1380     ID id;
1381     VALUE val;
```

```
1382 {  
1383     mod_av_set(klass, id, val, Qtrue);  
1384 }
```

(variable.c)

mod\_av\_set() does all the hard work:

▼ mod\_av\_set()

```
1352 static void  
1353 mod_av_set(klass, id, val, isconst)  
1354     VALUE klass;  
1355     ID id;  
1356     VALUE val;  
1357     int isconst;  
1358 {  
1359     char *dest = isconst ? "constant" : "class variable";  
1360  
1361     if (!OBJ_TAINTED(klass) && rb_safe_level() >= 4)  
1362         rb_raise(rb_eSecurityError, "Insecure: can't set %s"  
1363         if (OBJ_FROZEN(klass)) rb_error_frozen("class/module");  
1364         if (!RCLASS(klass)->iv_tbl) {  
1365             RCLASS(klass)->iv_tbl = st_init_numtable();  
1366         }  
1367         else if (isconst) {  
1368             if (st_lookup(RCLASS(klass)->iv_tbl, id, 0) ||  
1369                 (klass == rb_cObject && st_lookup(rb_class_tbl,  
1370                     rb_warn("already initialized %s %s", dest, rb_id  
1371                     )  
1372                 )  
1373             st_insert(RCLASS(klass)->iv_tbl, id, val);  
1375 }
```

(variable.c)

You can this time again ignore the warning checks (rb\_raise(), rb\_error\_frozen() and rb\_warn()). Here's what's left:

## ▼ mod\_av\_set() (only the important part)

```
if (!RCLASS(klass)->iv_tbl) {  
    RCLASS(klass)->iv_tbl = st_init_numtable();  
}  
st_insert(RCLASS(klass)->iv_tbl, id, val);
```

We're now sure constants also reside in the instance table. It means in the `iv_tbl` of `struct RClass`, the following are mixed together:

1. the class's own instance variables
2. class variables
3. constants

## ■ Reading

We now know how the constants are stored. We'll now check how they really work.

## rb\_const\_get()

We'll now look at `rb_const_get()`, the function to read a constant. This function returns the constant referred to by `id` from the class `klass`.

## ▼ rb\_const\_get()

```
1156 VALUE  
1157 rb_const_get(klass, id)  
1158     VALUE klass;
```

```

1159     ID id;
1160 {
1161     VALUE value, tmp;
1162     int mod_retry = 0;
1163
1164     tmp = klass;
1165     retry:
1166     while (tmp) {
1167         if (RCLASS(tmp)->iv_tbl &&
1168             st_lookup(RCLASS(tmp)->iv_tbl, id, &value)) {
1169             return value;
1170         }
1171         if (tmp == rb_cObject && top_const_get(id, &value))
1172             return value;
1173         tmp = RCLASS(tmp)->super;
1174     }
1175     if (!mod_retry && BUILTIN_TYPE(klass) == T_MODULE) {
1176         mod_retry = 1;
1177         tmp = rb_cObject;
1178         goto retry;
1179     }
1180
1181     /* Uninitialized constant */
1182     if (klass && klass != rb_cObject) {
1183         rb_name_error(id, "uninitialized constant %s at %s",
1184                         rb_id2name(id),
1185                         RSTRING(rb_class_path(klass))->ptr);
1186     }
1187     else { /* global_uninitialized */
1188         rb_name_error(id, "uninitialized constant %s", rb_id2
1189     }
1190     return Qnil; /* not reached */
1191 }

```

(variable.c)

There's a lot of code in the way. First, we should at least remove the `rb_name_error()` in the second half. In the middle, what's around `mod_entry` seems to be a special handling for modules. Let's also remove that for the time being. The function gets reduced to this:

## ▼ rb\_const\_get (simplified)

```
VALUE
rb_const_get(klass, id)
    VALUE klass;
    ID id;
{
    VALUE value, tmp;

    tmp = klass;
    while (tmp) {
        if (RCLASS(tmp)->iv_tbl && st_lookup(RCLASS(tmp)->iv_tbl, i
            return value;
        }
        if (tmp == rb_cObject && top_const_get(id, &value)) return
        tmp = RCLASS(tmp)->super;
    }
}
```

Now it should be pretty easy to understand. The function searches for the constant in `iv_tbl` while climbing `klass`'s superclass chain. That means:

```
class A
  Const = "ok"
end
class B < A
  p(Const)    # can be accessed
end
```

The only problem remaining is `top_const_get()`. This function is only called for `rb_cObject` so `top` must mean “top-level”. If you don't remember, at the top-level, the class is `Object`. This means the same as “in the class statement defining `c`, the class becomes `c`”, meaning that “the top-level's class is `Object`”.

```
# the class of the top-level is Object
class A
  # the class is A
  class B
    # the class is B
  end
end
```

So `top_const_get()` probably does something specific to the top level.

## top\_const\_get()

Let's look at this `top_const_get` function. It looks up the `id` constant writes the value in `klassp` and returns.

### ▼ top\_const\_get()

```
1102 static int
1103 top_const_get(id, klassp)
1104     ID id;
1105     VALUE *klassp;
1106 {
1107     /* pre-defined class */
1108     if (st_lookup(rb_class_tbl, id, klassp)) return Qtrue;
1109
1110     /* autoload */
1111     if (autoload_tbl && st_lookup(autoload_tbl, id, 0)) {
1112         rb_autoload_load(id);
1113         *klassp = rb_const_get(rb_cObject, id);
1114         return Qtrue;
1115     }
1116     return Qfalse;
1117 }
```

(variable.c)

`rb_class_tbl` was already mentioned in chapter 4 “Classes and modules”. It’s the table for storing the classes defined at the top-level. Built-in classes like `String` or `Array` have for example an entry in it. That’s why we should not forget to search in this table when looking for top-level constants.

The next block is related to autoloading. It is designed to be able to register a library that is loaded automatically when accessing a particular top-level constant for the first time. This can be used like this:

```
autoload(:VeryBigClass, "verybigclass")  # VeryBigClass is defined in
```

After this, when `VeryBigClass` is accessed for the first time, the `verybigclass` library is loaded (with `require`). As long as `VeryBigClass` is defined in the library, execution can continue smoothly. It’s an efficient approach, when a library is too big and a lot of time is spent on loading.

This autoload is processed by `rb_autoload_xxxx()`. We won’t discuss autoload further in this chapter because there will probably be a big change in how it works soon.

(translator’s note: The way autoload works *did* change in 1.8: autoloaded constants do not need to be defined at top-level anymore).

## Other classes?

But where did the code for looking up constants in other classes end up? After all, constants are first looked up in the outside classes, then in the superclasses.

In fact, we do not yet have enough knowledge to look at that. The outside classes change depending on the location in the program. In other words it depends of the program context. So we need first to understand how the internal state of the evaluator is handled. Specifically, this search in other classes is done in the `ev_const_get()` function of `eval.c`. We'll look at it and finish with the constants in the third part of the book.

## Global variables

---

### General remarks

Global variables can be accessed from anywhere. Or put the other way around, there is no need to restrict access to them. Because they are not attached to any context, the table only has to be at one place, and there's no need to do any check. Therefore implementation is very simple.

But there is still quite a lot of code. The reason for this is that global variables of Ruby are equipped with some gimmicks which make it hard to regard them as mere variables. Functions like the following are only available for global variables:

- you can “hook” access of global variables
- you can alias them with alias

Let’s explain this simply.

## Aliases of variables

```
alias $newname $oldname
```

After this, you can use `$newname` instead of `$oldname`. alias for variables is mainly a counter-measure for “symbol variables”. “symbol variables” are variables inherited from Perl like `$=` or `$0`. `$=` decides if during string comparison upper and lower case letters should be differentiated. `$0` shows the name of the main Ruby program. There are some other symbol variables but anyway as their name is only one character long, they are difficult to remember for people who don’t know Perl. So, aliases were created to make them a little easier to understand.

That said, currently symbol variables are not recommended, and are moved one by one in singleton methods of suitable modules. The current school of thought is that `$=` and others will be abolished in 2.0.

## Hooks

You can “hook” read and write of global variables.

Although hooks can be also be set at the Ruby level, I think the

purpose of it seems rather to prepare the special variables for system use like `$KCODE` at C level. `$KCODE` is the variable containing the encoding the interpreter currently uses to handle strings. Essentially only special strings like "EUC" or "UTF8" can be assigned to it, but this is too bothersome so it is designed so that "e" or "u" can also be used.

```
p($KCODE)      # "NONE" (default)
$KCODE = "e"
p($KCODE)      # "EUC"
$KCODE = "u"
p($KCODE)      # "UTF8"
```

Knowing that you can hook assignment of global variables, you should understand easily how this can be done. By the way, `$KCODE`'s K comes from "kanji" (the name of Chinese characters in Japanese).

You might say that even with `alias` or `hooks`, global variables just aren't used much, so it's functionality that doesn't really matter. It's adequate not to talk much about unused functions, and I'd like to use more pages for the analysis of the parser and evaluator. That's why I'll proceed with the explanation below whose degree of half-hearted is 85%.

## ■ Data structure

I said that the point when looking at how variables work is the way they are stored. First, I'd like you to firmly grasp the structure used

by global variables.

## ▼ Data structure for global variables

```
21 static st_table *rb_global_tbl;  
  
334 struct global_entry {  
335     struct global_variable *var;  
336     ID id;  
337 };  
  
324 struct global_variable {  
325     int counter;          /* reference counter */  
326     void *data;            /* value of the variable */  
327     VALUE (*getter)();    /* function to get the variable */  
328     void (*setter)();    /* function to set the variable */  
329     void (*marker)();    /* function to mark the variable */  
330     int block_trace;  
331     struct trace_var *trace;  
332 };
```

(variable.c)

`rb_global_tbl` is the main table. All global variables are stored in this table. The keys of this table are of course variable names (`ID`). A value is expressed by a `struct global_entry` and a `struct global_variable` (figure 1).

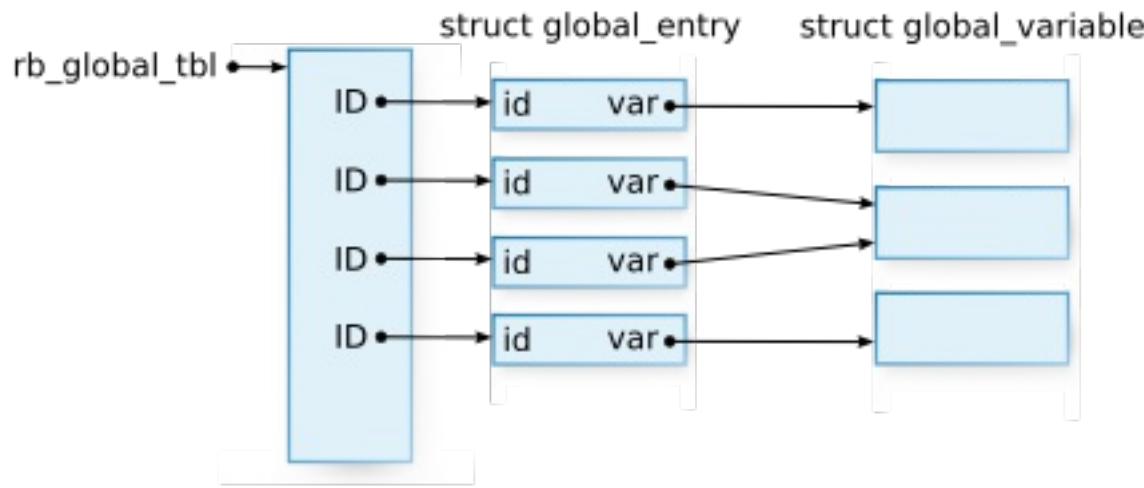


Figure 1: Global variables table at execution time

The structure representing the variables is split in two to be able to create aliases. When an alias is established, two `global_entry`s point to the same `global_variable`.

It's at this time that the reference counter (the `counter` member of `struct global_variable`) is necessary. I explained the general idea of a reference counter in the previous section “Garbage collection”. Reviewing it briefly, when a new reference to the structure is made, the counter is incremented by 1. When the reference is not used anymore, the counter is decreased by 1. When the counter reaches 0, the structure is no longer useful so `free()` can be called.

When hooks are set at the Ruby level, a list of `struct trace_vars` is stored in the `trace` member of `struct global_variable`, but I won't talk about it, and omit `struct trace_var`.

## Reading

You can have a general understanding of global variables just by looking at how they are read. The functions for reading them are `rb_gv_get()` and `rb_gvar_get()`.

## ▼ `rb_gv_get()` `rb_gvar_get()`

```
716 VALUE
717 rb_gv_get(name)
718     const char *name;
719 {
720     struct global_entry *entry;
721
722     entry = rb_global_entry(global_id(name));
723     return rb_gvar_get(entry);
724 }

649 VALUE
650 rb_gvar_get(entry)
651     struct global_entry *entry;
652 {
653     struct global_variable *var = entry->var;
654     return (*var->getter)(entry->id, var->data, var);
655 }
```

(variable.c)

A substantial part of the content seems to turn around the `rb_global_entry()` function, but that does not prevent us understanding what's going on. `global_id` is a function that converts a `char*` to `ID` and checks if it's the `ID` of a global variable. `(*var->getter)(...)` is of course a function call using the function pointer `var->getter`. If `p` is a function pointer, `(*p)(arg)` calls the function.

But the main part is still `rb_global_entry()`.

## ▼ `rb_global_entry()`

```
351 struct global_entry*
352 rb_global_entry(id)
353     ID id;
354 {
355     struct global_entry *entry;
356
357     if (!st_lookup(rb_global_tbl, id, &entry)) {
358         struct global_variable *var;
359         entry = ALLOC(struct global_entry);
360         st_add_direct(rb_global_tbl, id, entry);
361         var = ALLOC(struct global_variable);
362         entry->id = id;
363         entry->var = var;
364         var->counter = 1;
365         var->data = 0;
366         var->getter = undef_getter;
367         var->setter = undef_setter;
368         var->marker = undef_marker;
369
370         var->block_trace = 0;
371         var->trace = 0;
372     }
373     return entry;
374 }
```

`(variable.c)`

The main treatment is only done by the `st_lookup()` at the beginning. What's done afterwards is just creating (and storing) a new entry. As, when accessing a non existing global variable, an entry is automatically created, `rb_global_entry()` will never return NULL.

This was mainly done for speed. When the parser finds a global variable, it gets the corresponding `struct global_entry`. When reading the value of the variable, the value is just obtained from the entry (using `rb_gv_get()`).

Let's now continue a little with the code that follows. `var->getter` and others are set to `undef_xxxx`. `undef` probably means that they are the setter/getter/marker for a global variable whose state is undefined.

`undef_getter()` just shows a warning and returns `nil`, as even undefined global variables can be read. `undef_setter()` is a little bit interesting so let's look at it.

### ▼ `undef_setter()`

```
385 static void
386 undef_setter(val, id, data, var)
387     VALUE val;
388     ID id;
389     void *data;
390     struct global_variable *var;
391 {
392     var->getter = val_getter;
393     var->setter = val_setter;
394     var->marker = val_marker;
395
396     var->data = (void*)val;
397 }
```

`(variable.c)`

`val_getter()` takes the value from `entry->data` and returns it.

`val_getter()` just puts a value in `entry->data`. Setting handlers this way allows us not to need special handling for undefined variables (figure 2). Skillfully done, isn't it?

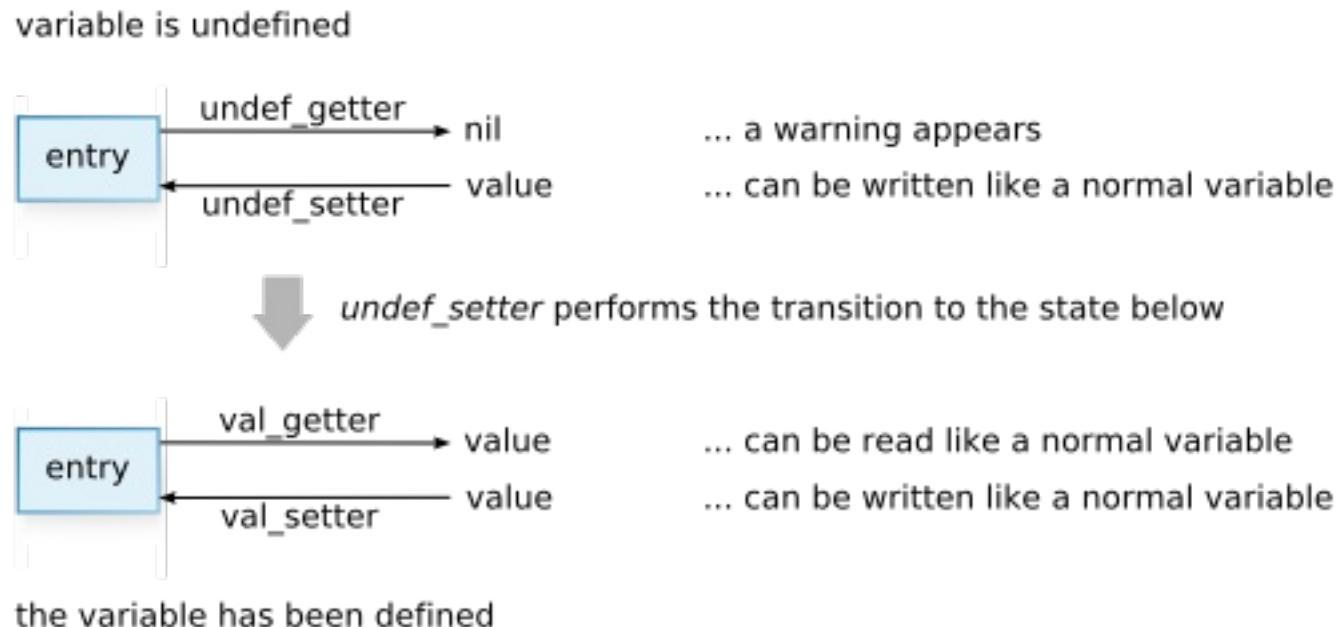


Figure 2: Setting and consultation of global variables

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

Translated by Clifford Escobar CAOILE & ocha-

# Chapter 7: Security

## Fundamentals

I say security but I don't mean passwords or encryption. The Ruby security feature is used for handling untrusted objects in a environment like CGI programming.

For example, when you want to convert a string representing a number into a integer, you can use the `eval` method. However, `eval` is a method that “runs a string as a Ruby program.” If you `eval` a string from a unknown person from the network, it is very dangerous. However for the programmer to fully differentiate between safe and unsafe things is very tiresome and cumbersome. Therefore, it is for certain that a mistake will be made. So, let us make it part of the language, was reasoning for this feature.

So then, how Ruby protect us from that sort of danger? Causes of dangerous operations, for example, opening unintended files, are roughly divided into two groups:

- Dangerous data
- Dangerous code

For the former, the code that handles these values is created by the programmers themselves, so therefore it is (relatively) safe. For

the latter, the program code absolutely cannot be trusted.

Because the solution is vastly different between the two causes, it is important to differentiate them by level. This are called security levels. The Ruby security level is represented by the `$SAFE` global variable. The value ranges from minimum value 0 to maximum value 4. When the variable is assigned, the level increases. Once the level is raised it can never be lowered. And for each level, the operations are limited.

I will not explain level 1 or 3. Level 0 is the normal program environment and the security system is not running. Level 2 handles dangerous values. Level 4 handles dangerous code. We can skip 0 and move on to explain in detail levels 2 and 4.

((errata: Level 1 handles dangerous values. “Level 2 has no use currently” is right.))

## Level 1

This level is for dangerous data, for example, in normal CGI applications, etc.

A per-object “tainted mark” serves as the basis for the Level 1 implementation. All objects read in externally are marked tainted, and any attempt to `eval` or `File.open` with a tainted object will cause an exception to be raised and the attempt will be stopped.

This tainted mark is “infectious”. For example, when taking a part

of a tainted string, that part is also tainted.

## Level 4

This level is for dangerous programs, for example, running external (unknown) programs, etc.

At level 1, operations and the data it uses are checked, but at level 4, operations themselves are restricted. For example, exit, file I/O, thread manipulation, redefining methods, etc. Of course, the tainted mark information is used, but basically the operations are the criteria.

## Unit of Security

`$SAFE` looks like a global variable but is in actuality a thread local variable. In other words, Ruby's security system works on units of thread. In Java and .NET, rights can be set per component (object), but Ruby does not implement that. The assumed main target was probably CGI.

Therefore, if one wants to raise the security level of one part of the program, then it should be made into a different thread and have its security level raised. I haven't yet explained how to create a thread, but I will show an example here:

```
# Raise the security level in a different thread
p($SAFE)  # 0 is the default
Thread.fork {  # Start a different thread
  $SAFE = 4  # Raise the level
```

```
eval(str)      # Run the dangerous program
}
p($SAFE)      # Outside of the block, the level is still 0
```

## Reliability of \$SAFE

Even with implementing the spreading of tainted marks, or restricting operations, ultimately it is still handled manually. In other words, internal libraries and external libraries must be completely compatible and if they don't, then the partway the "tainted" operations will not spread and the security will be lost. And actually this kind of hole is often reported. For this reason, this writer does not wholly trust it.

That is not to say, of course, that all Ruby programs are dangerous. Even at `$SAFE=0` it is possible to write a secure program, and even at `$SAFE=4` it is possible to write a program that fits your whim. However, one cannot put too much confidence on `$SAFE` (yet).

In the first place, functionality and security do not go together. It is common sense that adding new features can make holes easier to open. Therefore it is prudent to think that `ruby` can probably be dangerous.

## Implementation

From now on, we'll start to look into its implementation. In order to wholly grasp the security system of `ruby`, we have to look at "where is being checked" rather than its mechanism. However, this

time we don't have enough pages to do it, and just listing them up is not interesting. Therefore, in this chapter, I'll only describe about the mechanism used for security checks. The APIs to check are mainly these below two:

- `rb_secure(n)` : If more than or equal to level n, it would raise `SecurityError`.
- `SafeStringValue()` : If more than or equal to level 1 and a string is tainted, then it would raise an exception.

We won't read `SafeStringValue()` here.

## Tainted Mark

The taint mark is, to be concrete, the `FL_TAINT` flag, which is set to `basic->flags`, and what is used to infect it is the `OBJ_INFECT()` macro. Here is its usage.

<code>OBJ_TAINT(obj)</code>	<code>/* set FL_TAINT to obj */</code>
<code>OBJ_TAINTED(obj)</code>	<code>/* check if FL_TAINT is set to obj */</code>
<code>OBJ_INFECT(dest, src)</code>	<code>/* infect FL_TAINT from src to dest */</code>

Since `OBJ_TAINT()` and `OBJ_TAINTED()` can be assumed not important, let's briefly look over only `OBJ_INFECT()`.

### ▼ `OBJ_INFECT`

```
441 #define OBJ_INFECT(x,s) do {  
    if (FL_ABLE(x) && FL_ABLE(s))  
        RBASIC(x)->flags |= RBASIC(s)->flags & FL_TAINT; \\\n
```

```
} while (0)
```

```
(ruby.h)
```

FL\_ABLE() checks if the argument VALUE is a pointer or not. If the both objects are pointers (it means each of them has its flags member), it would propagate the flag.

## \$SAFE

### ▼ ruby\_safe\_level

```
124 int ruby_safe_level = 0;  
  
7401 static void  
7402 safe_setter(val)  
7403     VALUE val;  
7404 {  
7405     int level = NUM2INT(val);  
7406  
7407     if (level < ruby_safe_level) {  
7408         rb_raise(rb_eSecurityError, "tried to downgrade safe  
7409                     ruby_safe_level, level);  
7410     }  
7411     ruby_safe_level = level;  
7412     curr_thread->safe = level;  
7413 }
```

(eval.c)

The substance of \$SAFE is ruby\_safe\_level in eval.c. As I previously wrote, \$SAFE is local to each thread, It needs to be written in eval.c where the implementation of threads is located. In other words, it is in eval.c only because of the restrictions of C, but it can

essentially be located in another place.

`safe_setter()` is the setter of the `$SAFE` global variable. It means, because this function is the only way to access it from Ruby level, the security level cannot be lowered.

However, as you can see, from C level, because `static` is not attached to `ruby_safe_level`, you can ignore the interface and modify the security level.

## rb\_secure()

### ▼ rb\_secure()

```
136 void
137 rb_secure(level)
138     int level;
139 {
140     if (level <= ruby_safe_level) {
141         rb_raise(rb_eSecurityError, "Insecure operation `%%s'
142                         rb_id2name(ruby_frame->last_func), ruby_saf
143     }
144 }
```

(eval.c)

If the current safe level is more than or equal to `level`, this would raise `SecurityError`. It's simple.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

# Chapter 8 : Ruby Language Details

I'll talk about the details of Ruby's syntax and evaluation, which haven't been covered yet. I didn't intend a complete exposition, so I left out everything which doesn't come up in this book. That's why you won't be able to write Ruby programs just by reading this. A complete exposition can be found in the \footnote{Ruby reference manual: `archives/ruby-refm.tar.gz` in the attached CD-ROM}

Readers who know Ruby can skip over this chapter.

## Literals

---

The expressiveness of Ruby's literals is extremely high. In my opinion, what makes Ruby a script language is firstly the existence of the toplevel, secondly it's the expressiveness of its literals. Thirdly it might be the richness of its standard library.

A single literal already has enormous power, but even more when multiple literals are combined. Especially the ability of creating complex literals that hash and array literals are combined is the biggest advantage of Ruby's literal. One can write, for instance, a hash of arrays of regular expressions by constructing straightforwardly.

What kind of expressions are valid? Let's look at them one by one.

## ■ Strings

Strings and regular expressions can't be missing in a scripting language. The expressiveness of Ruby's string is very various even more than the other Ruby's literals.

### Single Quoted Strings

```
'string'          # 「string」
'\"begin{document}' # 「\begin{document}」
'\n'               # 「\n」 backslash and an n, not a newline
'\1'               # 「\1」 backslash and 1
'\''              # 「'」
```

This is the simplest form. In C, what enclosed in single quotes becomes a character, but in Ruby, it becomes a string. Let's call this a '-string'. The backslash escape is in effect only for \ itself and '. If one puts a backslash in front of another character the backslash remains as in the fourth example.

And Ruby's strings aren't divided by newline characters. If we write a string over several lines the newlines are contained in the string.

```
'multi
line
string'
```

And if the -K option is given to the ruby command, multibyte strings will be accepted. At present the three encodings EUC-JP (-Ke), Shift

JIS (-Ks), and UTF8 (-Ku) can be specified.

```
'「漢字が通る」と「マルチバイト文字が通る」はちょっと違う'  
# 'There's a little difference between "Kanji are accepted" and
```

## Double Quoted Strings

```
"string"           # 「string」  
"\n"               # newline  
"\x0f"             # a byte given in hexadecimal form  
"page#{n}.html"   # embedding a command
```

With double quotes we can use command expansion and backslash notation. The backslash notation is something classical that is also supported in C, for instance, \n is a newline, \b is a backspace. In Ruby, `ctrl-c` and `ESC` can also be expressed, that's convenient. However, merely listing the whole notation is not fun, regarding its implementation, it just means a large number of cases to be handled and there's nothing especially interesting. Therefore, they are entirely left out here.

On the other hand, expression expansion is even more fantastic. We can write an arbitrary Ruby expression inside `#{ }` and it will be evaluated at runtime and embedded into the string. There are no limitations like only one variable or only one method. Getting this far, it is not a mere literal anymore but the entire thing can be considered as an expression to express a string.

```
"embedded #{lvar} expression"  
"embedded #{@ivar} expression"
```

```
"embedded #{1 + 1} expression"
"embedded #{method_call(arg)} expression"
"embedded #{'"string' in string"} expression"
```

## Strings with %

%q(string)	# same as 'string'
%Q(string)	# same as "string"
%(string)	# same as %Q(string) or "string"

If a lot of separator characters appear in a string, escaping all of them becomes a burden. In that case the separator characters can be changed by using %. In the following example, the same string is written as a "-string and %-string.

```
"<a href=\"http://i.loveruby.net#{path}\">"
%Q(<a href="http://i.loveruby.net#{path}">)
```

The both expressions has the same length, but the %-one is a lot nicer to look at. When we have more characters to escape in it, %-string would also have advantage in length.

Here we have used parentheses as delimiters, but something else is fine, too. Like brackets or braces or #. Almost every symbol is fine, even %.

```
%q#this is string#
%q[this is string]
%q%this is string%
```

## Here Documents

Here document is a syntax which can express strings spanning multiple lines. A normal string starts right after the delimiter " and everything until the ending " would be the content. When using here document, the lines between the line which contains the starting <<EOS and the line which contains the ending EOS would be the content.

"the characters between the starting symbol and the ending symbol will become a string."

```
<<EOS
All lines between the starting and
the ending line are in this
here document
EOS
```

Here we used EOS as identifier but any word is fine. Precisely speaking, all the character matching [a-zA-Z\_0-9] and multi-byte characters can be used.

The characteristic of here document is that the delimiters are “the lines containing the starting identifier or the ending identifier”. The line which contains the start symbol is the starting delimiter. Therefore, the position of the start identifier in the line is not important. Taking advantage of this, it doesn’t matter that, for instance, it is written in the middle of an expression:

```
printf(<<EOS, count_n(str))
count=%d
EOS
```

In this case the string "count=%d\n" goes in the place of <<EOS. So it's the same as the following.

```
printf("count=%d\n", count_n(str))
```

The position of the starting identifier is really not restricted, but on the contrary, there are strict rules for the ending symbol: It must be at the beginning of the line and there must not be another letter in that line. However if we write the start symbol with a minus like this <<-EOS we can indent the line with the end symbol.

<<-EOS

It would be convenient if one could indent the content of a here document. But that's not possible.

If you want that, writing a method to delete indents is usually a way to go. But beware of tabs.

EOS

Furthermore, the start symbol can be enclosed in single or double quotes. Then the properties of the whole here document change. When we change <<EOS to <<"EOS" we can use embedded expressions and backslash notation.

<<"EOS"

One day is #{24 \* 60 \* 60} seconds.

Incredible.

EOS

But <<'EOS' is not the same as a single quoted string. It starts the complete literal mode. Everything even backslashes go into the string as they are typed. This is useful for a string which contains

many backslashes.

In Part 2, I'll explain how to parse a here document. But I'd like you to try to guess it before.

## Characters

Ruby strings are byte sequences, there are no character objects. Instead there are the following expressions which return the integers which correspond a certain character in ASCII code.

```
?a          # the integer which corresponds to "a"  
?.          # the integer which corresponds to "."  
?\n         # LF  
?\C-a      # Ctrl-a
```

## Regular Expressions

```
/regexp/  
/^Content-Length:/i  
/正規表現/  
/\/\/.*.*?\*\/\//m      # An expression which matches C comments  
/reg#{1 + 1}exp/        # the same as /reg2exp/
```

What is contained between slashes is a regular expression. Regular expressions are a language to designate string patterns. For example

```
/abc/
```

This regular expression matches a string where there's an a

followed by a `b` followed by a `c`. It matches “abc” or “fffffffabc” or “abcxxxxx”.

One can designate more special patterns.

```
/^From:/
```

This matches a string where there’s a `From` followed by a `:` at the beginning of a line. There are several more expressions of this kind, such that one can create quite complex patterns.

The uses are infinite: Changing the matched part to another string, deleting the matched part, determining if there’s one match and so on...

A more concrete use case would be, for instance, extracting the `From:` header from a mail, or changing the `\n` to an `\r`, or checking if a string looks like a mail address.

Since the regular expression itself is an independent language, it has its own parser and evaluator which are different from `ruby`. They can be found in `regex.c`. Hence, it’s enough for `ruby` to be able to cut out the regular expression part from a Ruby program and feed it. As a consequence, they are treated almost the same as strings from the grammatical point of view. Almost all of the features which strings have like escapes, backslash notations and embedded expressions can be used in the same way in regular expressions.

However, we can say they are treated as the same as strings only when we are in the viewpoint of “Ruby’s syntax”. As mentioned before, since regular expression itself is a language, naturally we have to follow its language constraints. To describe regular expression in detail, it’s so large that one more can be written, so I’d like you to read another book for this subject. I recommend “Mastering Regular Expression” by Jeffrey E.F. Friedl.

## Regular Expressions with %

Also as with strings, regular expressions also have a syntax for changing delimiters. In this case it is %r. To understand this, looking at some examples are enough to understand.

```
%r(regexp)
%r[/\*.*?\*/]          # matches a C comment
%r("(?:[^"\\]+|\\.)*") # matches a string in C
%r{reg#{1 + 1}exp}      # embedding a Ruby expression
```

## Arrays

A comma-separated list enclosed in brackets [] is an array literal.

```
[1, 2, 3]
['This', 'is', 'an', 'array', 'of', 'string']

[/regexp/, {'hash'=>3}, 4, 'string', ?\C-a]

lvar = $gvar = @ivar = @@cvar = nil
[lvar, $gvar, @ivar, @@cvar]
[Object.new(), Object.new(), Object.new()]
```

Ruby's array (Array) is a list of arbitrary objects. From a syntactical standpoint, it's characteristic is that arbitrary expressions can be elements. As mentioned earlier, an array of hashes of regular expressions can easily be made. Not just literals but also expressions which variables or method calls combined together can also be written straightforwardly.

Note that this is “an expression which generates an array object” as with the other literals.

```
i = 0
while i < 5
  p([1,2,3].id)    # Each time another object id is shown.
  i += 1
end
```

## Word Arrays

When writing scripts one uses arrays of strings a lot, hence there is a special notation only for arrays of strings. That is %w. With an example it's immediately obvious.

```
%w( alpha beta gamma delta )    # ['alpha','beta','gamma','delta']
%w( 月 火 水 木 金 土 日 )
%w( Jan Feb Mar Apr May Jun
    Jul Aug Sep Oct Nov Dec )
```

There's also %w where expressions can be embedded. It's a feature implemented fairly recently.

```
n = 5
```

```
%w( list0 list#{n} )  # ['list0', 'list#{n}']  
%w( list0 list#{n} )  # ['list0', 'list5']
```

The author hasn't come up with a good use of `%w` yet.

## Hashes

Hash tables are data structure which store a one-to-one relation between arbitrary objects. By writing as follows, they will be expressions to generate tables.

```
{ 'key' => 'value', 'key2' => 'value2' }  
{ 3 => 0, 'string' => 5, ['array'] => 9 }  
{ Object.new() => 3, Object.new() => 'string' }  
  
# Of course we can put it in several lines.  
{ 0 => 0,  
 1 => 3,  
 2 => 6 }
```

We explained hashes in detail in the third chapter “Names and Nametables”. They are fast lookup tables which allocate memory slots depending on the hash values. In Ruby grammar, both keys and values can be arbitrary expressions.

Furthermore, when used as an argument of a method call, the `{...}` can be omitted under a certain condition.

```
some_method(arg, key => value, key2 => value2)  
# some_method(arg, {key => value, key2 => value2}) # same as abc
```

With this we can imitate named (keyword) arguments.

```
button.set_geometry('x' => 80, 'y' => '240')
```

Of course in this case `set_geometry` must accept a hash as input. Though real keyword arguments will be transformed into parameter variables, it's not the case for this because this is just a "imitation".

## Ranges

Range literals are oddballs which don't appear in most other languages. Here are some expressions which generate Range objects.

```
0..5          # from 0 to 5 containing 5
0...5         # from 0 to 5 not containing 5
1+2 .. 9+0    # from 3 to 9 containing 9
'a'...'z'     # strings from 'a' to 'z' containing 'z'
```

If there are two dots the last element is included. If there are three dots it is not included. Not only integers but also floats and strings can be made into ranges, even a range between arbitrary objects can be created if you'd attempt. However, this is a specification of Range class, which is the class of range objects, (it means a library), this is not a matter of grammar. From the parser's standpoint, it just enables to concatenate arbitrary expressions with ... If a range cannot be generated with the objects as the evaluated results, it would be a runtime error.

By the way, because the precedence of .. and ... is quite low,

sometimes it is interpreted in a surprising way.

```
1..5.to_a()    # 1..(5.to_a())
```

I think my personality is relatively bent for Ruby grammar, but somehow I don't like only this specification.

## ■ Symbols

In Part 1, we talked about symbols at length. It's something corresponds one-to-one to an arbitrary string. In Ruby symbols are expressed with a : in front.

```
:identifier  
:abcde
```

These examples are pretty normal. Actually, besides them, all variable names and method names can become symbols with a : in front. Like this:

```
:$gvar  
:@ivar  
:@@cvar  
:CONST
```

Moreover, though we haven't talked this yet, [] or attr= can be used as method names, so naturally they can also be used as symbols.

```
:[ ]  
:attr=
```

When one uses these symbols as values in an array, it'll look quite complicated.

## ▀ Numerical Values

This is the least interesting. One possible thing I can introduce here is that, when writing a million,

1\_000\_000

as written above, we can use underscore delimiters in the middle. But even this isn't particularly interesting. From here on in this book, we'll completely forget about numerical values.

## Methods

---

Let's talk about the definition and calling of methods.

## ▀ Definition and Calls

```
def some_method( arg )
  ...
end

class C
  def some_method( arg )
  ...
end
```

```
end
```

Methods are defined with `def`. If they are defined at toplevel they become function style methods, inside a class they become methods of this class. To call a method which was defined in a class, one usually has to create an instance with `new` as shown below.

```
C.new().some_method(0)
```

## The Return Value of Methods

The return value of a method is, if a `return` is executed in the middle, its value. Otherwise, it's the value of the statement which was executed last.

```
def one()      # 1 is returned
  return 1
  999
end
```

```
def two()      # 2 is returned
  999
  2
end
```

```
def three()    # 3 is returned
  if true then
    3
  else
    999
  end
end
```

If the method body is empty, it would automatically be `nil`, and an expression without a value cannot put at the end. Hence every method has a return value.

## Optional Arguments

Optional arguments can also be defined. If the number of arguments doesn't suffice, the parameters are automatically assigned to default values.

```
def some_method( arg = 9 )  # default value is 9
  p arg
end

some_method(0)      # 0 is shown.
some_method()       # The default value 9 is shown.
```

There can also be several optional arguments. But in that case they must all come at the end of the argument list. If elements in the middle of the list were optional, how the correspondences of the arguments would be very unclear.

```
def right_decl( arg1, arg2, darg1 = nil, darg2 = nil )
  ...
end

# This is not possible
def wrong_decl( arg, default = nil, arg2 )  # A middle argument
  ...
end
```

## Omitting argument parentheses

In fact, the parentheses of a method call can be omitted.

```
puts 'Hello, World!'    # puts("Hello, World")
obj = Object.new        # obj = Object.new()
```

In Python we can get the method object by leaving out parentheses, but there is no such thing in Ruby.

If you'd like to, you can omit more parentheses.

```
puts(File.basename fname)
# puts(File.basename(fname)) same as the above
```

If we like we can even leave out more

```
puts File.basename fname
# puts(File.basename(fname)) same as the above
```

However, recently this kind of “nested omissions” became a cause of warnings. It's likely that this will not pass anymore in Ruby 2.0.

Actually even the parentheses of the parameters definition can also be omitted.

```
def some_method param1, param2, param3
end
```

```
def other_method    # without arguments ... we see this a lot
end
```

Parentheses are often left out in method calls, but leaving out parentheses in the definition is not very popular. However if there

are no arguments, the parentheses are frequently omitted.

## Arguments and Lists

Because Arguments form a list of objects, there's nothing odd if we can do something converse: extracting a list (an array) as arguments, as the following example.

```
def delegate(a, b, c)
  p(a, b, c)
end

list = [1, 2, 3]
delegate(*list)  # identical to delegate(1, 2, 3)
```

In this way we can distribute an array into arguments. Let's call this device a `*argument` now. Here we used a local variable for demonstration, but of course there is no limitation. We can also directly put a literal or a method call instead.

```
m(*[1,2,3])      # We could have written the expanded form in the
m(*mcall())
```

The `*` argument can be used together with ordinary arguments, but the `*` argument must come last. Otherwise, the correspondences to parameter variables cannot be determined in a single way.

In the definition on the other hand we can handle the arguments in bulk when we put a `*` in front of the parameter variable.

```
def some_method( *args )
```

```
p args
end

some_method()          # prints []
some_method(0)         # prints [0]
some_method(0, 1)       # prints [0,1]
```

The surplus arguments are gathered in an array. Only one \*parameter can be declared. It must also come after the default arguments.

```
def some_method0( arg, *rest )
end
def some_method1( arg, darg = nil, *rest )
end
```

If we combine list expansion and bulk reception together, the arguments of one method can be passed as a whole to another method. This might be the most practical use of the \*parameter.

```
# a method which passes its arguments to other_method
def delegate(*args)
  other_method(*args)
end

def other_method(a, b, c)
  return a + b + c
end

delegate(0, 1, 2)      # same as other_method(0, 1, 2)
delegate(10, 20, 30)   # same as other_method(10, 20, 30)
```

## ■ Various Method Call Expressions

Being just a single feature as ‘method call’ does not mean its

representation is also single. Here is about so-called syntactic sugar. In Ruby there is a ton of it, and they are really attractive for a person who has a fetish for parsers. For instance the examples below are all method calls.

```
1 + 2                      # 1.+(2)
a == b                      # a.==(b)
~/regexp/                   # /regexp/.~
obj.attr = val              # obj.attr=(val)
obj[i]                      # obj.[](i)
obj[k] = v                  # obj.[]=(k,v)
<code>cvs diff abstract.rd</code> # Kernel.``('cvs diff abstract')
```

It's hard to believe until you get used to it, but `attr=`, `[]=`, `\`` are (indeed) all method names. They can appear as names in a method definition and can also be used as symbols.

```
class C
  def []( index )
  end
  def +( another )
  end
end
p(:attr=)
p(:[]=)
p(:`)
```

As there are people who don't like sweets, there are also many people who dislike syntactic sugar. Maybe they feel unfair when the things which are essentially the same appear in faked looks. (Why's everyone so serious?)

Let's see some more details.

# Symbol Appendices

obj.name?  
obj.name!

First a small thing. It's just appending a ? or a !. Call and Definition do not differ, so it's not too painful. There are convention for what to use these method names, but there is no enforcement on language level. It's just a convention at human level. This is probably influenced from Lisp in which a great variety of characters can be used in procedure names.

## Binary Operators

1 + 2    # 1.+ (2)

Binary Operators will be converted to a method call to the object on the left hand side. Here the method + from the object 1 is called. As listed below there are many of them. There are the general operators + and -, also the equivalence operator == and the spaceship operator `<=>' as in Perl, all sorts. They are listed in order of their precedence.

```
**  
* / %  
+ -  
<< >>  
&  
| ^  
> >= < <=  
<=> == === =~
```

The symbols & and | are methods, but the double symbols && and || are built-in operators. Remember how it is in C.

## Unary Operators

```
+2
-1.0
~/regexp/
```

These are the unary operators. There are only three of them: + - ~. + and - work as they look like (by default). The operator ~ matches a string or a regular expression with the variable `$_`. With an integer it stands for bit conversion.

To distinguish the unary + from the binary + the method names for the unary operators are `+@` and `-@` respectively. Of course they can be called by just writing `+n` or `-n`.

((errata: + or - as the prefix of a numeric literal is actually scanned as a part of the literal. This is a kind of optimizations.))

## Attribute Assignment

```
obj.attr = val  # obj.attr=(val)
```

This is an attribute assignment fashion. The above will be translated into the method call `attr=`. When using this together with method calls whose parentheses are omitted, we can write code which looks like attribute access.

```

class C
  def i() @i end          # We can write the definition in one line
  def i=(n) @i = n end
end

c = C.new
c.i = 99
p c.i    # prints 99

```

However it will turn out both are method calls. They are similar to get/set property in Delphi or slot accessors in CLOS.

Besides, we cannot define a method such as `obj.attr(arg)=`, which can take another argument in the attribute assignment fashion.

## Index Notation

```
obj[i]    # obj.[](i)
```

The above will be translated into a method call for `[]`. Array and hash access are also implemented with this device.

```
obj[i] = val    # obj.[](i, val)
```

Index assignment fashion. This is translated into a call for a method named `[]=`.

## super

We relatively often have a situation where we want add a little bit to the behaviour of an already existing method rather than

replacing it. Here a mechanism to call a method of the superclass when overwriting a method is required. In Ruby, that's `super`.

```
class A
  def test
    puts 'in A'
  end
end
class B < A
  def test
    super    # invokes A#test
  end
end
```

Ruby's `super` differs from the one in Java. This single word means "call the method with the same name in the superclass". `super` is a reserved word.

When using `super`, be careful about the difference between `super` with no arguments and `super` whose arguments are omitted. The `super` whose arguments are omitted passes all the given parameter variables.

```
class A
  def test( *args )
    p args
  end
end

class B < A
  def test( a, b, c )
    # super with no arguments
    super()    # shows []
    # super with omitted arguments. Same result as super(a, b, c)
    super      # shows [1, 2, 3]
  end
end
```

```
end  
end
```

```
B.new.test(1,2,3)
```

# Visibility

In Ruby, even when calling the same method, it can be or cannot be called depending on the location (meaning the object). This functionality is usually called “visibility” (whether it is visible). In Ruby, the below three types of methods can be defined.

- `public`
- `private`
- `protected`

`public` methods can be called from anywhere in any form. `private` methods can only be called in a form “syntactically” without a receiver. In effect they can only be called by instances of the class in which they were defined and in instances of its subclass.

`protected` methods can only be called by instances of the defining class and its subclasses. It differs from `private` that methods can still be called from other instances of the same class.

The terms are the same as in C++ but the meaning is slightly different. Be careful.

Usually we control visibility as shown below.

```
class C
```

```
public
def a1() end    # becomes public
def a2() end    # becomes public

private
def b1() end    # becomes private
def b2() end    # becomes private

protected
def c1() end    # becomes protected
def c2() end    # becomes protected
end
```

Here `public`, `private` and `protected` are method calls without parentheses. These aren't even reserved words.

`public` and `private` can also be used with an argument to set the visibility of a particular method. But its mechanism is not interesting. We'll leave this out.

## Module functions

Given a module 'M'. If there are two methods with the exact same content

- `M.method_name`
- `M#method_name(Visibility is private)`

then we call this a module function.

It is not apparent why this should be useful. But let's look at the next example which is happily used.

```
Math.sin(5)          # If used for a few times this is more convenient
include Math
sin(5)              # If used more often this is more practical
```

It's important that both functions have the same content. With a different `self` but with the same code the behavior should still be the same. Instance variables become extremely difficult to use. Hence such method is very likely a method in which only procedures are written (like `sin`). That's why they are called module "functions".

## Iterators

---

Ruby's iterators differ a bit from Java's or C++'s iterator classes or 'Iterator' design pattern. Precisely speaking, those iterators are called exterior iterators, Ruby's iterators are interior iterators. Regarding this, it's difficult to understand from the definition so let's explain it with a concrete example.

```
arr = [0,2,4,6.8]
```

This array is given and we want to access the elements in order. In C style we would write the following.

```
i = 0
while i < arr.length
  print arr[i]
```

```
i += 1
end
```

Using an iterator we can write:

```
arr.each do |item|
  print item
end
```

Everything from `each` to `end` is the call to an iterator method. More precisely `each` is the iterator method and between `do` and `end` is the iterator block. The part between the vertical bars are called block parameters, which become variables to receive the parameters passed from the iterator method to the block.

Saying it a little abstractly, an iterator is something like a piece of code which has been cut out and passed. In our example the piece `print item` has been cut out and is passed to the `each` method. Then `each` takes all the elements of the array in order and passes them to the cut out piece of code.

We can also think the other way round. The other parts except `print item` are being cut out and enclosed into the `each` method.

```
i = 0
while i < arr.length
  print arr[i]
  i += 1
end

arr.each do |item|
  print item
end
```

# Comparison with higher order functions

What comes closest in C to iterators are functions which receive function pointers, it means higher order functions. But there are two points in which iterators in Ruby and higher order functions in C differ.

Firstly, Ruby iterators can only take one block. For instance we can't do the following.

```
# Mistake. Several blocks cannot be passed.
array_of_array.each do |i|
  ...
end do |j|
  ...
end
```

Secondly, Ruby's blocks can share local variables with the code outside.

```
lvar = 'ok'
[0,1,2].each do |i|
  p lvar    # Can acces local variable outside the block.
end
```

That's where iterators are convenient.

But variables can only be shared with the outside. They cannot be shared with the inside of the iterator method ( e.g. each). Putting it

intuitively, only the variables in the place which looks of the source code continued are visible.

## Block Local Variables

Local variables which are assigned inside a block stay local to that block, it means they become block local variables. Let's check it out.

```
[0].each do
  i = 0
  p i    # 0
end
```

For now, to create a block, we apply each on an array of length 1 (We can fully leave out the block parameter). In that block, the `i` variable is first assigned .. meaning declared. This makes `i` block local.

It is said block local, so it should not be able to access from the outside. Let's test it.

```
% ruby -e '
[0].each do
  i = 0
end
p i    # Here occurs an error.
'
-e:5: undefined local variable or method `i'
for #<Object:0x40163a9c> (NameError)
```

When we referenced a block local variable from outside the block,

surely an error occurred. Without a doubt it stayed local to the block.

Iterators can also be nested repeatedly. Each time the new block creates another scope.

```
lvar = 0
[1].each do
  var1 = 1
  [2].each do
    var2 = 2
    [3].each do
      var3 = 3
      # Here lvar, var1, var2, var3 can be seen
    end
    # Here lvar, var1, var2 can be seen
  end
  # Here lvar, var1 can be seen
end
# Here only lvar can be seen
```

There's one point which you have to keep in mind. Differing from nowadays' major languages Ruby's block local variables don't do shadowing. Shadowing means for instance in C that in the code below the two declared variables `i` are different.

```
{
  int i = 3;
  printf("%d\n", i);          /* 3 */
  {
    int i = 99;
    printf("%d\n", i);          /* 99 */
  }
  printf("%d\n", i);          /* 3 (元に戻った) */
```

Inside the block the `i` inside overshadows the `i` outside. That's why it's called shadowing.

But what happens with block local variables of Ruby where there's no shadowing. Let's look at this example.

```
i = 0
p i          # 0
[0].each do
  i = 1
  p i          # 1
end
p i          # 1 the change is preserved
```

Even when we assign `i` inside the block, if there is the same name outside, it would be used. Therefore when we assign to inside `i`, the value of outside `i` would be changed. On this point there came many complains: "This is error prone. Please do shadowing." Each time there's nearly flaming but till now no conclusion was reached.

## ■ The syntax of iterators

There are some smaller topics left.

First, there are two ways to write an iterator. One is the `do ~ end` as used above, the other one is the enclosing in braces. The two expressions below have exactly the same meaning.

```
arr.each do |i|
  puts i
end
```

```
arr.each { |i|      # The author likes a four space indentation for
          puts i      # an iterator with braces.
}
```

But grammatically the precedence is different. The braces bind much stronger than `do~end`.

```
 m m do .... end      # m(m) do....end
 m m { .... }          # m(m()) {....})
```

And iterators are definitely methods, so there are also iterators that take arguments.

```
re = /\d/                  # regular expression to match a digit
$stdin.grep(re) do |line|  # look repeatedly for this regular e>
  ...
end
```

## yield

Of course users can write their own iterators. Methods which have a `yield` in their definition text are iterators. Let's try to write an iterator with the same effect as `Array#each`:

```
# adding the definition to the Array class
class Array
  def my_each
    i = 0
    while i < self.length
      yield self[i]
      i += 1
    end
  end
end
```

```
# this is the original each
[0,1,2,3,4].each do |i|
  p i
end
```

```
# my_each works the same
[0,1,2,3,4].my_each do |i|
  p i
end
```

`yield` calls the block. At this point control is passed to the block, when the execution of the block finishes it returns back to the same location. Think about it like a characteristic function call. When the present method does not have a block a runtime error will occur.

```
% ruby -e '[0,1,2].each'
-e:1:in `each': no block given (LocalJumpError)
  from -e:1
```

## Proc

I said, that iterators are like cut out code which is passed as an argument. But we can even more directly make code to an object and carry it around.

```
twice = Proc.new {|n| n * 2 }
p twice.call(9)  # 18 will be printed
```

In short, it is like a function. As might be expected from the fact it is created with `new`, the return value of `Proc.new` is an instance of the `Proc` class.

`Proc.new` looks surely like an iterator and it is indeed so. It is an ordinary iterator. There's only some mystic mechanism inside `Proc.new` which turns an iterator block into an object.

Besides there is a function style method `lambda` provided which has the same effect as `Proc.new`. Choose whatever suits you.

```
twice = lambda {|n| n * 2 }
```

## Iterators and Proc

Why did we start talking all of a sudden about `Proc`? Because there is a deep relationship between iterators and `Proc`. In fact, iterator blocks and `Proc` objects are quite the same thing. That's why one can be transformed into the other.

First, to turn an iterator block into a `Proc` object one has to put an `&` in front of the parameter name.

```
def print_block( &block )
  p block
end

print_block() do end    # Shows something like <Proc:0x40155884>
print_block()           # Without a block nil is printed
```

With an `&` in front of the argument name, the block is transformed to a `Proc` object and assigned to the variable. If the method is not an iterator (there's no block attached) `nil` is assigned.

And in the other direction, if we want to pass a Proc to an iterator we also use &.

```
block = Proc.new {|i| p i }
[0,1,2].each(&block)
```

This code means exactly the same as the code below.

```
[0,1,2].each {|i| p i }
```

If we combine these two, we can delegate an iterator block to a method somewhere else.

```
def each_item( &block )
  [0,1,2].each(&block)
end

each_item do |i|      # same as [0,1,2].each do |i|
  p i
end
```

## Expressions

---

“Expressions” in Ruby are things with which we can create other expressions or statements by combining with the others. For instance a method call can be another method call’s argument, so it is an expression. The same goes for literals. But literals and method calls are not always combinations of elements. On the

contrary, “expressions”, which I’m going to introduce, always consists of some elements.

## if

We probably do not need to explain the `if` expression. If the conditional expression is true, the body is executed. As explained in Part 1, every object except `nil` and `false` is true in Ruby.

```
if cond0 then
  ....
  elsif cond1 then
  ....
  elsif cond2 then
  ....
else
  ....
end
```

`elsif/else`-clauses can be omitted. Each `then` as well. But there are some finer requirements concerning `then`. For this kind of thing, looking at some examples is the best way to understand. Here only thing I’d say is that the below codes are valid.

# 1	# 4
if cond then ..... end	if cond
	then ..... end
# 2	# 5
if cond; ..... end	if cond
	then
# 3	.....
if cond then; ..... end	end

And in Ruby, `if` is an expression, so there is the value of the entire `if` expression. It is the value of the body where a condition expression is met. For example, if the condition of the first `if` is true, the value would be the one of its body.

```
p(if true  then 1 else 2 end)    #=> 1
p(if false then 1 else 2 end)    #=> 2
p(if false then 1 elsif true then 2 else 3 end)  #=> 2
```

If there's no match, or the matched clause is empty, the value would be `nil`.

```
p(if false then 1 end)    #=> nil
p(if true  then  end)    #=> nil
```

## ■ unless

An `if` with a negated condition is an `unless`. The following two expressions have the same meaning.

unless cond then	if not (cond) then
....	....
end	end

`unless` can also have attached `else` clauses but any `elsif` cannot be attached. Needless to say, `then` can be omitted.

`unless` also has a value and its condition to decide is completely the same as `if`. It means the entire value would be the value of the body of the matched clause. If there's no match or the matched

clause is empty, the value would be `nil`.

## ■ and && or ||

The most likely utilization of the `and` is probably a boolean operation. For instance in the conditional expression of an `if`.

```
if cond1 and cond2
  puts 'ok'
end
```

But as in Perl, sh or Lisp, it can also be used as a conditional branch expression. The two following expressions have the same meaning.

```
invalid?(key) and return nil
```

```
if invalid?(key)
  return nil
end
```

`&&` and `and` have the same meaning. Different is the binding order.

```
method arg0 && arg1  # method(arg0 && arg1)
method arg0 and arg1  # method(arg0) and arg1
```

Basically the symbolic operator creates an expression which can be an argument (`arg`). The alphabetical operator creates an expression which cannot become an argument (`expr`).

As for `and`, if the evaluation of the left hand side is true, the right hand side will also be evaluated.

On the other hand `or` is the opposite of `and`. If the evaluation of the

left hand side is false, the right hand side will also be evaluated.

```
valid?(key) or return nil
```

or and || have the same relationship as && and and. Only the precedence is different.

## ■ The Conditional Operator

There is a conditional operator similar to C:

```
cond ? iftrue : iffalse
```

The space between the symbols is important. If they bump together the following weirdness happens.

```
cond?iftrue:iffalse # cond?(iftrue(:iffalse))
```

The value of the conditional operator is the value of the last executed expression. Either the value of the true side or the value of the false side.

## ■ while until

Here's a while expression.

```
while cond do
  ...
end
```

This is the simplest loop syntax. As long as `cond` is true the body is executed. The `do` can be omitted.

```
until io_ready?(id) do
  sleep 0.5
end
```

`until` creates a loop whose condition definition is opposite. As long as the condition is false it is executed. The `do` can be omitted.

Naturally there is also jump syntaxes to exit a loop. `break` as in C/C++/Java is also `break`, but `continue` is `next`. Perhaps `next` has come from Perl.

```
i = 0
while true
  if i > 10
    break    # exit the loop
  elsif i % 2 == 0
    i *= 2
    next    # next loop iteration
  end
  i += 1
end
```

And there is another Perlism: the `redo`.

```
while cond
  # (A)
  ...
  redo
  ...
end
```

It will return to (A) and repeat from there. What differs from next is it does not check the condition.

I might come into the world top 100, if the amount of Ruby programs would be counted, but I haven't used `redo` yet. It does not seem to be necessary after all because I've lived happily despite of it.

## case

A special form of the `if` expression. It performs branching on a series of conditions. The following left and right expressions are identical in meaning.

```
case value
when cond1 then
  ...
when cond2 then
  ...
when cond3, cond4 then
  ...
else
  ...
end
```

```
if cond1 === value
  ...
elsif cond2 === value
  ...
elsif cond3 === value or cond4 ===
  ...
else
  ...
end
```

The threefold equals `==` is, as the same as the `=`, actually a method call. Notice that the receiver is the object on the left hand side. Concretely, if it is the `==` of an `Array`, it would check if it contains the `value` as its element. If it is a `Hash`, it tests whether it has the `value` as its key. If its is an regular expression, it tests if the `value` matches. And so on. Since `case` has many grammatical elements, to

list them all would be tedious, thus we will not cover them in this book.

## Exceptions

This is a control structure which can pass over method boundaries and transmit errors. Readers who are acquainted to C++ or Java will know about exceptions. Ruby exceptions are basically the same.

In Ruby exceptions come in the form of the function style method `raise`. `raise` is not a reserved word.

```
raise ArgumentError, "wrong number of argument"
```

In Ruby exception are instances of the `Exception` class and it's subclasses. This form takes an exception class as its first argument and an error message as its second argument. In the above case an instance of `ArgumentError` is created and “thrown”. Exception object would ditch the part after the `raise` and start to return upwards the method call stack.

```
def raise_exception
  raise ArgumentError, "wrong number of argument"
  # the code after the exception will not be executed
  puts 'after raise'
end
raise_exception()
```

If nothing blocks the exception it will move on and on and finally it

will reach the top level. When there's no place to return any more, ruby gives out a message and ends with a non-zero exit code.

```
% ruby raise.rb
raise.rb:2:in `raise_exception': wrong number of arguments (ArgumentError)
from raise.rb:7
```

However an exit would be sufficient for this, and for an exception there should be a way to set handlers. In Ruby, begin~rescue~end is used for this. It resembles the try~catch in C++ and Java.

```
def raise_exception
  raise ArgumentError, "wrong number of argument"
end

begin
  raise_exception()
rescue ArgumentError => err
  puts 'exception catched'
  p err
end
```

rescue is a control structure which captures exceptions, it catches exception objects of the specified class and its subclasses. In the above example, an instance of ArgumentError comes flying into the place where ArgumentError is targeted, so it matches this rescue. By =>err the exception object will be assigned to the local variable err, after that the rescue part is executed.

```
% ruby rescue.rb
exception catched
#<ArgumentError: wrong number of argument>
```

When an exception is rescued, it will go through the rescue and it will start to execute the subsequent as if nothing happened, but we can also make it retry from the begin. To do so, `retry` is used.

```
begin    # the place to return
  ...
rescue ArgumentError => err then
  retry # retry your life
end
```

We can omit the `=>err` and the `then` after `rescue`. We can also leave out the exception class. In this case, it means as the same as when the `StandardError` class is specified.

If we want to catch more exception classes, we can just write them in line. When we want to handle different errors differently, we can specify several `rescue` clauses.

```
begin
  raise IOError, 'port not ready'
rescue ArgumentError, TypeError
rescue IOError
rescue NameError
end
```

When written in this way, a `rescue` clause that matches the exception class is searched in order from the top. Only the matched clause will be executed. For instance, only the clause of `IOError` will be executed in the above case.

On the other hand, when there is an `else` clause, it is executed only when there is no exception.

```
begin
  nil    # Of course here will no error occur
rescue ArgumentError
  # This part will not be executed
else
  # This part will be executed
end
```

Moreover an `ensure` clause will be executed in every case: when there is no exception, when there is an exception, rescued or not.

```
begin
  f = File.open('/etc/passwd')
  # do stuff
ensure  # this part will be executed anyway
  f.close
end
```

By the way, this `begin` expression also has a value. The value of the whole `begin~end` expression is the value of the part which was executed last among `begin/rescue/else` clauses. It means the last statement of the clauses aside from `ensure`. The reason why the `ensure` is not counted is probably because `ensure` is usually used for cleanup (thus it is not a main line).

## Variables and Constants

Referring a variable or a constant. The value is the object the variable points to. We already talked in too much detail about the various behaviors.

```
lvar
@ivar
@@cvar
CONST
$gvar
```

I want to add one more thing. Among the variables starting with \$, there are special kinds. They are not necessarily global variables and some have strange names.

First the Perlish variables \$\_ and \$. \$\_ saves the return value of gets and other methods, \$. contains the last match of a regular expression. They are incredible variables which are local variables and simultaneously thread local variables.

And the \$! to hold the exception object when an error is occurred, the \$? to hold the status of a child process, the \$SAFE to represent the security level, they are all thread local.

## Assignment

Variable assignments are all performed by =. All variables are typeless. What is saved is a reference to an object. As its implementation, it was a VALUE (pointer).

```
var = 1
obj = Object.new
@ivar = 'string'
@@cvar = ['array']
PI = 3.1415926535
$gvar = {'key' => 'value'}
```

However, as mentioned earlier `obj.attr=val` is not an assignment but a method call.

## ■ Self Assignment

```
var += 1
```

This syntax is also in C/C++/Java. In Ruby,

```
var = var + 1
```

it is a shortcut of this code. Differing from C, the Ruby `+` is a method and thus part of the library. In C, the whole meaning of `+=` is built in the language processor itself. And in C++, `+=` and `*=` can be wholly overwritten, but we cannot do this in Ruby. In Ruby `+=` is always defined as an operation of the combination of `+` and assignment.

We can also combine self assignment and an attribute-access-flavor method. The result more looks like an attribute.

```
class C
  def i() @i end          # A method definition can be written i
  def i=(n) @i = n end
end

obj = C.new
obj.i = 1
obj.i += 2    # obj.i = obj.i + 2
p obj.i      # 3
```

If there is `+=` there might also be `++` but this is not the case. Why is that so? In Ruby assignment is dealt with on the language level. But on the other hand methods are in the library. Keeping these two, the world of variables and the world of objects, strictly apart is an important peculiarity of Ruby. If `++` were introduced the separation might easily be broken. That's why there's no `++`

Some people don't want to go without the brevity of `++`. It has been proposed again and again in the mailing list but was always turned down. I am also in favor of `++` but not as much as I can't do without, and I have not felt so much needs of `++` in Ruby in the first place, so I've kept silent and decided to forget about it.

## ■ `defined?`

`defined?` is a syntax of a quite different color in Ruby. It tells whether an expression value is “defined” or not at runtime.

```
var = 1
defined?(var)    #=> true
```

In other words it tells whether a value can be obtained from the expression received as its argument (is it okay to call it so?) when the expression is evaluated. That said but of course you can't write an expression causing a parse error, and it could not detect if the expression is something containing a method call which raises an error in it.

I would have loved to tell you more about `defined?` but it will not

appear again in this book. What a pity.

# Statements

---

A statement is what basically cannot be combined with the other syntaxes, in other words, they are lined vertically.

But it does not mean there's no evaluated value. For instance there are return values for class definition statements and method definition statements. However this is rarely recommended and isn't useful, you'd better regard them lightly in this way. Here we also skip about the value of each statement.

## The Ending of a statement

Up to now we just said “For now one line’s one statement”. But Ruby’s statement ending’s aren’t that straightforward.

First a statement can be ended explicitly with a semicolon as in C. Of course then we can write two and more statements in one line.

```
puts 'Hello, World!'; puts 'Hello, World once more!'
```

On the other hand, when the expression apparently continues, such as just after opened parentheses, dyadic operators, or a comma, the statement continues automatically.

```
# 1 + 3 * method(6, 7 + 8)
1 +
3 *
  method(
    6,
    7 + 8)
```

But it's also totally no problem to use a backslash to explicitly indicate the continuation.

```
p 1 + \
2
```

## ▀ The Modifiers **if** and **unless**

The **if** modifier is an irregular version of the normal **if**. The programs on the left and right mean exactly the same.

on_true() if cond	if cond
	on_true()
	end

The **unless** is the negative version. Guard statements ( statements which exclude exceptional conditions) can be conveniently written with it.

## ▀ The Modifiers **while** and **until**

**while** and **until** also have a back notation.

```
process() while have_content?
sleep(1) until ready?
```

Combining this with `begin` and `end` gives a do-while-loop like in C.

```
begin
  res = get_response(id)
end while need_continue?(res)
```

## Class Definition

```
class C < SuperClass
  ...
end
```

Defines the class `C` which inherits from `SuperClass`

We talked quite extensively about classes in Part 1. This statement will be executed, the class to be defined will become `self` within the statement, arbitrary expressions can be written within. Class definitions can be nested. They form the foundation of Ruby execution image.

## Method Definition

```
def m(arg)
end
```

I've already written about method definition and won't add more. This section is put to make it clear that they also belong to statements.

# Singleton method definition

We already talked a lot about singleton methods in Part 1. They do not belong to classes but to objects, in fact, they belong to singleton classes. We define singleton methods by putting the receiver in front of the method name. Parameter declaration is done the same way like with ordinary methods.

```
def obj.some_method
end
```

```
def obj.some_method2( arg1, arg2, darg = nil, *rest, &block )
end
```

## Definition of Singleton methods

```
class << obj
  ...
end
```

From the viewpoint of purposes, it is the statement to define some singleton methods in a bundle. From the viewpoint of measures, it is the statement in which the singleton class of `obj` becomes `self` when executed. In all over the Ruby program, this is the only place where a singleton class is exposed.

```
class << obj
  p self  #=> #<Class:#<Object:0x40156fcc>>  # Singleton Class
  def a() end  # def obj.a
  def b() end  # def obj.b
end
```

# Multiple Assignment

With a multiple assignment, several assignments can be done all at once. The following is the simplest case:

```
a, b, c = 1, 2, 3
```

It's exactly the same as the following.

```
a = 1
b = 2
c = 3
```

Just being concise is not interesting. in fact, when an array comes in to be mixed, it becomes something fun for the first time.

```
a, b, c = [1, 2, 3]
```

This also has the same result as the above. Furthermore, the right hand side does not need to be a grammatical list or a literal. It can also be a variable or a method call.

```
tmp = [1, 2, 3]
a, b, c = tmp
ret1, ret2 = some_method()    # some_method might probably return
```

Precisely speaking it is as follows. Here we'll assume `obj` is (the object of) the value of the left hand side,

1. `obj` if it is an array
2. if its `to_ary` method is defined, it is used to convert `obj` to an

array.

### 3. [obj]

Decide the right-hand side by following this procedure and perform assignments. It means the evaluation of the right-hand side and the operation of assignments are totally independent from each other.

And it goes on, both the left and right hand side can be infinitely nested.

```
a, (b, c, d) = [1, [2, 3, 4]]  
a, (b, (c, d)) = [1, [2, [3, 4]]]  
(a, b), (c, d) = [[1, 2], [3, 4]]
```

As the result of the execution of this program, each line will be a=1  
b=2 c=3 d=4.

And it goes on. The left hand side can be index or parameter assignments.

```
i = 0  
arr = []  
arr[i], arr[i+1], arr[i+2] = 0, 2, 4  
p arr    # [0, 2, 4]  
  
obj.attr0, obj.attr1, obj.attr2 = "a", "b", "c"
```

And like with method parameters, \* can be used to receive in a bundle.

```
first, *rest = 0, 1, 2, 3, 4
```

```
p first # 0
p rest # [1, 2, 3, 4]
```

When all of them are used all at once, it's extremely confusing.

## Block parameter and multiple assignment

We brushed over block parameters when we were talking about iterators. But there is a deep relationship between them and multiple assignment. For instance in the following case.

```
array.each do |i|
  ...
end
```

Every time when the block is called, the `yielded` arguments are multi-assigned to `i`. Here there's only one variable on the left hand side, so it does not look like multi assignment. But if there are two or more variables, it would look like it. For instance, `Hash#each` is an repeated operation on the pairs of keys and values, so usually we call it like this:

```
hash.each do |key, value|
  ...
end
```

In this case, each array consist of a key and a value is `yielded` from the hash.

Hence we can also do the following thing by using nested multiple assignment.

```
# [[key,value],index] are yielded
hash.each_with_index do |(key, value), index|
  ....
end
```

## alias

```
class C
  alias new orig
end
```

Defining another method `new` with the same body as the already defined method `orig`. `alias` are similar to hardlinks in a unix file system. They are a means of assigning multiple names to one method body. To say this inversely, because the names themselves are independent of each other, even if one method name is overwritten by a subclass method, the other one still remains with the same behavior.

## undef

```
class C
  undef method_name
end
```

Prohibits the calling of `C#method_name`. It's not just a simple revoking of the definition. If there even were a method in the superclass it would also be forbidden. In other words the method is exchanged

for a sign which says “This method must not be called”.

`undef` is extremely powerful, once it is set it cannot be deleted from the Ruby level because it is used to cover up contradictions in the internal structure. Only one left measure is inheriting and defining a method in the lower class. Even in that case, calling `super` would cause an error occurring.

The method which corresponds to `unlink` in a file system is `Module#remove_method`. While defining a class, `self` refers to that class, we can call it as follows (Remember that `Class` is a subclass of `Module`.)

```
class C
  remove_method(:method_name)
end
```

But even with a `remove_method` one cannot cancel the `undef`. It's because the sign put up by `undef` prohibits any kind of searches.

((errata: It can be redefined by using `def`))

## **Some more small topics**

---

### **Comments**

```
# examples of bad comments.
```

```
1 + 1          # compute 1+1.  
alias my_id id # my_id is an alias of id.
```

From a `#` to the end of line is a comment. It doesn't have a meaning for the program.

## ■ Embedded documents

```
=begin  
This is an embedded document.  
It's so called because it is embedded in the program.  
Plain and simple.  
=end
```

An embedded document stretches from an `=begin` outside a string at the beginning of a line to a `=end`. The interior can be arbitrary. The program ignores it as a mere comment.

## ■ Multi-byte strings

When the global variable `$KCODE` is set to either `EUC`, `SJIS` or `UTF8`, strings encoded in `euc-jp`, `shift_jis`, or `utf8` respectively can be used in a string of a data.

And if the option `-Ke`, `-Ks` or `-Ku` is given to the `ruby` command multibyte strings can be used within the Ruby code. String literals, regular expressions and even operator names can contain multibyte characters. Hence it is possible to do something like this:

```
def 表示( arg )  
  puts arg
```

end

表示 'にほんご'

But I really cannot recommend doing things like that.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License](#)

# Ruby Hacking Guide

Translated by Vincent ISAMBART & ocha-

# Chapter 9: yacc crash course

## Outline

---

### Parser and scanner

How to write parsers for programming languages has been an active area of research for a long time, and there is a quite firm established tactic for doing it. If we limit ourselves to a grammar not too strange (or ambiguous), we can solve this problem by following this method.

The first part consists in splitting a string in a list of words (or tokens). This is called a scanner or lexer. The term “lexical analyzer” is also used, but is too complicated to say so we’ll use the name scanner.

When speaking about scanners, the common sense first says “there are generally spaces at the end of a word”. And in practice, it was made like this in most programming languages, because it’s the easiest way.

There can also be exceptions. For example, in the old Fortran, white spaces did not have any meaning. This means a white space did not end a word, and you could put spaces in the name of a variable. However that made the parsing very complicated so the compiler vendors, one by one, started ignoring that standard. Finally Fortran 90 followed this trend and made the fact that white spaces have an impact the standard.

By the way, it seems the reason white spaces had not meaning in Fortran 77 was that when writing programs on punch cards it was easy to make errors in the number of spaces.

## ■ List of symbols

I said that the scanner spits out a list of words (tokens), but, to be exact, what the scanner creates is a list of “symbols”, not words.

What are symbols? Let’s take numbers as an example. In a programming language, 1, 2, 3, 99 are all “numbers”. They can all be handled the same way by the grammar. Where we can write 1, we can also write 2 or 3. That’s why the parser does not need to handle them in different ways. For numbers, “number” is enough.

“number”, “identifier” and others can be grouped together as “symbol”. But be careful not to mix this with the `Symbol` class.

The scanner first splits the string into words and determines what these symbols are. For example, `NUMBER` or `DIGIT` for numbers, `IDENTIFIER` for names like “name”, `IF` for the reserved word `if`. These

symbols are then given to the next phase.

## Parser generator

The list of words and symbols spitted out by the scanner are going to be used to form a tree. This tree is called a syntax tree.

The name “parser” is also sometimes used to include both the scanner and the creation of the syntax tree. However, we will use the narrow sense of “parser”, the creation of the syntax tree. How does this parser make a tree from the list of symbols? In other words, on what should we focus to find the tree corresponding to a piece of code?

The first way is to focus on the meaning of the words. For example, let's suppose we find the word `var`. If the definition of the local variable `var` has been found before this, we'll understand it's the reading of a local variable.

An other ways is to only focus on what we see. For example, if after an identified comes a ‘=’, we'll understand it's an assignment. If the reserved word `if` appears, we'll understand it's the start of an `if` statement.

The later method, focusing only on what we see, is the current trend. In other words the language must be designed to be analyzed just by looking at the list of symbols. The choice was because this way is simpler, can be more easily generalized and can therefore be

automatized using tools. These tools are called parser generators.

The most used parser generator under UNIX is yacc. Like many others, ruby's parser is written using yacc. The input file for this tool is `parser.y`. That's why to be able to read ruby's parser, we need to understand yacc to some extent. (Note: Starting from 1.9, ruby requires `bison` instead of `yacc`. However, `bison` is mainly `yacc` with additional functionality, so this does not diminish the interest of this chapter.)

This chapter will be a simple presentation of yacc to be able to understand `parse.y`, and therefore we will limit ourselves to what's needed to read `parse.y`. If you want to know more about parsers and parser generators, I recommend you a book I wrote called “Rubyを256倍使 うための本 無道編” (The book to use 256 times more of Ruby - Unreasonable book). I do not recommend it because I wrote it, but because in this field it's the easiest book to understand. And besides it's cheap so stakes will be low.

Nevertheless, if you would like a book from someone else (or can't read Japanese), I recommend O'Reilly's “lex & yacc programming” by John R. Levine, Tony Mason and Doug Brown. And if you are still not satisfied, you can also read “Compilers” (also known as the “dragon book” because of the dragon on its cover) by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman.

# Grammar

## Grammar file

The input file for yacc is called “grammar file”, as it’s the file where the grammar is written. The convention is to name this grammar file `*.y`. It will be given to yacc who will generate C source code. This file can then be compiled as usual (figure 1 shows the full process).

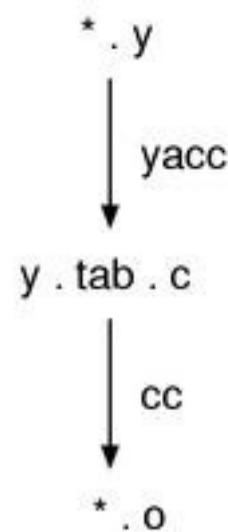


Figure 1: File dependencies

The output file name is always `y.tab.c` and can’t be changed. The recent versions of yacc usually allow to change it on the command line, but for compatibility it was safer to keep `y.tab.c`. By the way, it seems the `tab` of `y.tab.c` comes from `table`, as lots of huge tables are defined in it. It’s good to have a look at the file once.

The grammar file’s content has the following form:

## ▼ General form of the grammar file

```
%{  
Header  
%}  
%union ....  
%token ....  
%type ....  
  
%%  
Rules part  
%%  
User defined part
```

yacc's input file is first divided in 3 parts by `%%`. The first part is called the definition part, has a lot of definitions and setups. Between `%{` and `%}` we can write anything we want in C, like for example necessary macros. After that, the instructions starting with `%` are special yacc instructions. Every time we use one, we'll explain it.

The middle part of the file is called the rules part, and is the most essential part for yacc. It's where is written the grammar we want to parse. We'll explain it in details in the next section.

The last part of the file, the user defined part, can be used freely by the user. yacc just copies this part verbatim in the output file. It's used for example to put auxiliary routines needed by the parser.

## ■ **What does yacc do.**

What yacc takes care of is mainly this rules part in the middle. yacc

takes the grammar written there and use it to make a function called `yyparse()`. It's the parser, in the narrow sense of the word.

In the narrow sense, so it means a scanner is needed. However, `yacc` won't take care of it, it must be done by the user. The scanner is the function named `yylex()`.

Even if `yacc` creates `yyparse()`, it only takes care of its core part. The “actions” we'll mention later is out of its scope. You can think the part done by `yacc` is too small, but that's not the case. That's because this “core part” is overly important that `yacc` survived to this day even though we keep complaining about it.

But what on earth is this core part? That's what we're going to see.

## BNF

When we want to write a parser in C, its code will be “cut the string this way, make this an `if` statement...” When using parser generators, we say the opposite, that is “I would like to parse this grammar.” Doing this creates for us a parser to handle the grammar. This means telling the specification gives us the implementation. That's the convenient point of `yacc`.

But how can we tell the specification? With `yacc`, the method of description used is the BNF (Backus-Naur Form). Let's look at a very simple example.

```
if_stmt: IF expr THEN stmt END
```

Let's see separately what's at the left and at the right of the ":". The part on the left side, `if_stmt`, is equal to the right part... is what I mean here. In other words, I'm saying that:

`if_stmt` and `IF expr THEN stmt END` are equivalent.

Here, `if_stmt`, `IF`, `expr`... are all "symbols". `expr` is the abbreviation of expression, `stmt` of statement. It must be for sure the declaration of the `if` statement.

One definition is called a rule. The part at the left of ":" is called the left side and the right part called the right side. This is quite easy to remember.

But something is missing. We do not want an `if` statement without being able to use `else`. And even if we could write `else`, having to always write the `else` even when it's useless would be cumbersome. In this case we could do the following:

```
if_stmt: IF expr THEN stmt END
       | IF expr THEN stmt ELSE stmt END
```

"|" means "or".

`if_stmt` is either "`IF expr THEN stmt END`" or "`IF expr THEN stmt ELSE stmt END`".

That's it.

Here I would like you to pay attention to the split done with `|`. With just this, one more rule is added. In fact, punctuating with `|` is just a shorter way to repeat the left side. The previous example has exactly the same meaning as the following:

```
if_stmt: IF expr THEN stmt END
if_stmt: IF expr THEN stmt ELSE stmt END
```

This means two rules are defined in the example.

This is not enough to complete the definition of the `if` statement. That's because the symbols `expr` and `stmt` are not sent by the scanner, their rules must be defined. To be closer to Ruby, let's boldly add some rules.

```
stmt    : if_stmt
        | IDENTIFIER '=' expr /* assignment */
        | expr

if_stmt: IF expr THEN stmt END
        | IF expr THEN stmt ELSE stmt END

expr   : IDENTIFIER      /* reading a variable */
        | NUMBER        /* integer constant */
        | funcall       /* FUNction CALL */

funcall: IDENTIFIER '(' args ')'

args   : expr            /* only one parameter */
```

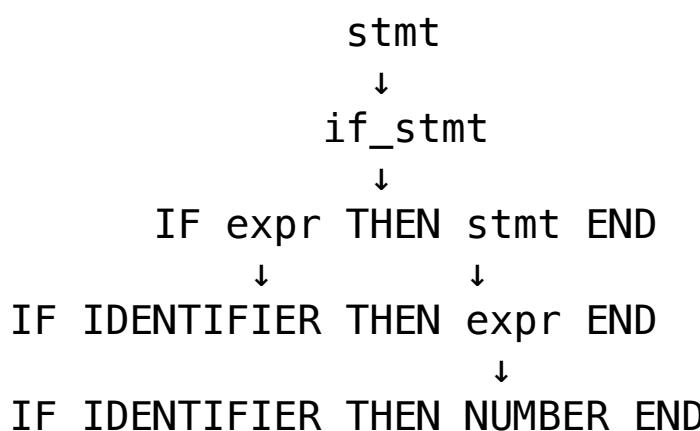
I used two new elements. First, comments of the same form as in C, and character expressed using `'=`'. This `'=`' is also of course a symbol. Symbols like `=` are different from numbers as there is

only one variety for them. That's why for symbols where we can also use '='. It would be great to be able to use for strings for, for example, reserved words, but due to limitations of the C language this cannot be done.

We add rules like this, to the point we complete writing all the grammar. With yacc, the left side of the first written rule is “the whole grammar we want to express”. So in this example, `stmt` expresses the whole program.

It was a little too abstract. Let's explain this a little more concretely. By “`stmt` expresses the whole program”, I mean `stmt` and the rows of symbols expressed as equivalent by the rules, are all recognized as grammar. For example, `stmt` and `stmt` are equivalent. Of course. Then `expr` is equivalent to `stmt`. That's expressed like this in the rule. Then, `NUMBER` and `stmt` are equivalent. That's because `NUMBER` is `expr` and `expr` is `stmt`.

We can also say that more complicated things are equivalent.



When it has expanded until here, all elements become the symbols

sent by the scanner. It means such sequence of symbols is correct as a program. Or putting it the other way around, if this sequence of symbols is sent by the scanner, the parser can understand it in the opposite order of expanding.

IF IDENTIFIER THEN NUMBER END

↓

IF IDENTIFIER THEN expr END

↓

↓

IF expr THEN stmt END

↓

if\_stmt

↓

stmt

And `stmt` is a symbol expressing the whole program. That's why this sequence of symbols is a correct program for the parser. When it's the case, the parsing routine `yyparse()` ends returning 0.

By the way, the technical term expressing that the parser succeeded is that it “accepted” the input. The parser is like a government office: if you do not fill the documents in the boxes exactly like he asked you to, he'll refuse them. The accepted sequences of symbols are the ones for which the boxes where filled correctly. Parser and government office are strangely similar for instance in the fact that they care about details in specification and that they use complicated terms.

## Terminal symbols and nonterminal symbols

Well, in the confusion of the moment I used without explaining it the expression “symbols coming from the scanner”. So let’s explain this. I use one word “symbol” but there are two types.

The first type of the symbols are the ones sent by the scanner. They are for example, IF, THEN, END, '=' , ... They are called terminal symbols. That’s because like before when we did the quick expansion we find them aligned at the end. In this chapter terminal symbols are always written in capital letters. However, symbols like '=' between quotes are special. Symbols like this are all terminal symbols, without exception.

The other type of symbols are the ones that never come from the scanner, for example if\_stmt, expr or stmt. They are called nonterminal symbols. As they don’t come from the scanner, they only exist in the parser. Nonterminal symbols also always appear at one moment or the other as the left side of a rule. In this chapter, nonterminal symbols are always written in lower case letters.

## How to test

I’m now going to tell you the way to process the grammar file with yacc.

```
%token A B C D E
%%
list: A B C
     | de
de   : D E
```

First, put all terminal symbols used after `%token`. However, you do not have to type the symbols with quotes (like `'='`). Then, put `%%` to mark a change of section and write the grammar. That's all.

Let's now process this.

```
% yacc first.y
% ls
first.y  y.tab.c
%
```

Like most Unix tools, “silence means success”.

There's also implementations of `yacc` that need semicolons at the end of (groups of) rules. When it's the case we need to do the following:

```
%token A B C D E
%%
list: A B C
      | de
      ;
de   : D E
      ;
```

I hate these semicolons so in this book I'll never use them.

## Void rules

Let's now look a little more at some of the established ways of grammar description. I'll first introduce void rules.

```
void:
```

There's nothing on the right side, this rule is “void”. For example, the two following targets means exactly the same thing.

```
target: A B C
```

```
target: A void B void C
void :
```

What is the use of such a thing? It's very useful. For example in the following case.

```
if_stmt : IF expr THEN stmts opt_else END
```

```
opt_else:
    | ELSE stmts
```

Using void rules, we can express cleverly the fact that “the else section may be omitted”. Compared to the rules made previously using two definitions, this way is shorter and we do not have to disperse the burden.

## Recursive definitions

The following example is still a little hard to understand.

```
list: ITEM          /* rule 1 */
    | list ITEM  /* rule 2 */
```

This expresses a list of one or more items, in other words any of

the following lists of symbols:

```
ITEM
ITEM ITEM
ITEM ITEM ITEM
ITEM ITEM ITEM ITEM
:
```

Do you understand why? First, according to rule 1 list can be read ITEM. If you merge this with rule 2, list can be ITEM ITEM.

```
list: list ITEM
      = ITEM ITEM
```

We now understand that the list of symbols ITEM ITEM is similar to list. By applying again rule 2 to list, we can say that 3 ITEM are also similar to list. By quickly continuing this process, the list can grow to any size. This is something like mathematical induction.

I'll now show you the next example. The following example expresses the lists with 0 or more ITEM.

```
list:
| list ITEM
```

First the first line means “list is equivalent to (void)”. By void I mean the list with 0 ITEM. Then, by looking at rule 2 we can say that “list ITEM” is equivalent to 1 ITEM. That's because list is equivalent to void.

```
list: list ITEM
```

```
= (void) ITEM
=           ITEM
```

By applying the same operations of replacement multiple times, we can understand that `list` is the expression a list of 0 or more items.

With this knowledge, “lists of 2 or more `ITEM`” or “lists of 3 or more `ITEM`” are easy, and we can even create “lists of an even number of elements”.

```
list:
| list ITEM ITEM
```

## Construction of values

---

This abstract talk lasted long enough so in this section I’d really like to go on with a more concrete talk.

### ■ Shift and reduce

Up until now, various ways to write grammars have been explained, but what we want is being able to build a syntax tree. However, I’m afraid to say, only telling it the rules is not enough to be able to let it build a syntax tree, as might be expected. Therefore, this time, I’ll tell you the way to build a syntax tree by adding something to the rules.

We'll first see what the parser does during the execution. We'll use the following simple grammar as an example.

```
%token A B C
%%
program: A B C
```

In the parser there is a stack called the semantic stack. The parser pushes on it all the symbols coming from the scanner. This move is called “shifting the symbols”.

[ A B ] ← C shift

And when any of the right side of a rule is equal to the end of the stack, it is “interpreted”. When this happens, the sequence of the right-hand side is replaced by the symbol of the left-hand side.

[ A B C ]  
↓ reduction  
[ program ]

This move is called “reduce A B C” to program”. This term is a little presumptuous, but in short it is like, when you have enough number of tiles of haku and hatsu and chu respectively, it becomes “Big three dragons” in Japanese Mahjong, ... this might be irrelevant.

And since program expresses the whole program, if there's only a program on the stack, it probably means the whole program is found out. Therefore, if the input is just finished here, it is accepted.

Let's try with a little more complicated grammar.

```
%token IF E S THEN END
%%
program : if
if      : IF expr THEN stmts END
expr    : E
stmts  : S
| stmts S
```

The input from the scanner is this.

IF E THEN S S S END

The transitions of the semantic stack in this case are shown below.

Stack	Move
empty at first	
IF	shift IF
IF E	shift E
IF expr	reduce E to expr
IF expr THEN	shift THEN
IF expr THEN S	shift S
IF expr THEN stmts	reduce S to stmts
IF expr THEN stmts S	shift S
IF expr THEN stmts S	reduce stmts S to stmts
IF expr THEN stmts S	shift S
IF expr THEN stmts	reduce stmts S to stmts
IF expr THEN stmts END	shift END
if	reduce IF expr THEN stmts END to if

program reduce if to program  
accept.

As the end of this section, there's one thing to be cautious with. a reduction does not always means decreasing the symbols. If there's a void rule, it's possible that a symbol is generated out of "void".

## Action

Now, I'll start to describe the important parts. Whichever shifting or reducing, doing several things only inside of the semantic stack is not meaningful. Since our ultimate goal was building a syntax tree, it cannot be sufficient without leading to it. How does yacc do it for us? The answer yacc made is that "we shall enable to hook the moment when the parser performing a reduction." The hooks are called actions of the parser. An action can be written at the last of the rule as follows.

```
program: A B C { /* Here is an action */ }
```

The part between { and } is the action. If you write like this, at the moment reducing A B C to program this action will be executed. Whatever you do as an action is free. If it is a C code, almost all things can be written.

## The value of a symbol

This is further more important but, each symbol has "its value".

Both terminal and nonterminal symbols do. As for terminal symbols, since they come from the scanner, their values are also given by the scanner. For example, 1 or 9 or maybe 108 for a NUMBER symbol. For an IDENTIFIER symbol, it might be "attr" or "name" or "sym". Anything is fine. Each symbol and its value are pushed together on the semantic stack. The next figure shows the state just the moment *s* is shifted with its value.

IF	expr	THEN	stmts	S
value	value	value	value	value

According to the previous rule, *stmts S* can be reduced to *stmts*. If an action is written at the rule, it would be executed, but at that moment, the values of the symbols corresponding to the right-hand side are passed to the action.

IF	expr	THEN	stmts	S	/* Stack */
v1	v2	v3	v4	v5	
			↓	↓	
		stmts:	stmts	S	/* Rule */
			↓	↓	
			{	\$1 + \$2; }	/* Action */

This way an action can take the value of each symbol corresponding to the right-hand side of a rule through \$1, \$2, \$3, ... yacc will rewrite the kinds of \$1 and \$2 to the notation to point to the stack. However because it is written in c language it needs to handle, for instance, types, but because it is tiresome, let's assume their types are of `int` for the moment.

Next, instead it will push the symbol of the left-hand side, but because all symbols have their values the left-hand side symbol must also have its value. It is expressed as \$\$ in actions, the value of \$\$ when leaving an action will be the value of the left-hand side symbol.

```
IF      expr      THEN      stmts      S      /* the stack just before reduc
v1      v2          v3          v4          v5
                  ↓          ↓
stmts:  stmts      S      /* the rule that the right-har
                  ↑          ↓          ↓
{ $$      = $1      +      $2; }  /* its action */

IF      expr      THEN      stmts      /* the stack after reducing */
v1      v2          v3          (v4+v5)
```

As the end of this section, this is just an extra. The value of a symbol is sometimes called “semantic value”. Therefore the stack to put them is the “semantic value stack”, and it is called “semantic stack” for short.

## yacc and types

It's really cumbersome but without talking about types we cannot finish this talk. What is the type of the value of a symbol? To say the bottom line first, it will be the type named YYSTYPE. This must be the abbreviation of either YY Stack TYPE or Semantic value TYPE. And YYSTYPE is obviously the `typedef` of somewhat another type. The type is the union defined with the instruction named `%union` in the definition part.

We have not written `%union` before but it did not cause an error. Why? This is because yacc considerately process with the default value without asking. The default value in C should naturally be `int`. Therefore, `YYSTYPE` is `int` by default.

As for an example of a yacc book or a calculator, `int` can be used unchanged. But in order to build a syntax tree, we want to use structs and pointers and the other various things. Therefore for instance, we use `%union` as follows.

```
%union {  
    struct node {  
        int type;  
        struct node *left;  
        struct node *right;  
    } *node;  
    int num;  
    char *str;  
}
```

Because this is not for practical use, the arbitrary names are used for types and members. Notice that it is different from the ordinal C but there's no semicolon at the end of the `%union` block.

And, if this is written, it would look like the following in `y.tab.c`.

```
typedef union {  
    struct node {  
        int type;  
        struct node *left;  
        struct node *right;  
    } *node;  
    int num;  
    char *str;
```

```
} YYSTYPE;
```

And, as for the semantic stack,

```
YYSTYPE yyvs[256];      /* the substance of the stack (yyvs = Y
YYSTYPE *yyvsp = yyvs;  /* the pointer to the end of the stack
```

we can expect something like this. Therefore, the values of the symbols appear in actions would be

```
/* the action before processed by yacc */
target: A B C { func($1, $2, $3); }

/* after converted, its appearance in y.tab.c */
{ func(yyvsp[-2], yyvsp[-1], yyvsp[0]); ;
```

naturally like this.

In this case, because the default value `int` is used, it can be accessed just by referring to the stack. If `YYSTYPE` is a union, it is necessary to also specify one of its members. There are two ways to do that, one way is associating with each symbol, another way is specifying every time.

Generally, the way of associating with each type is used. By using `%token` for terminal symbols and using `%type` for nonterminal symbols, it is written as follows.

```
%token<num> A B C      /* All of the values of A B C is of type ir
%type<str> target       /* All of the values of target is of type c
```

On the other hand, if you'd like to specify everytime, you can write a member name into next to \$ as follows.

```
%union { char *str; }  
%%  
target: { $<str>$ = "In short, this is like typecasting"; }
```

You'd better avoid using this method if possible.  
Defining a member for each symbol is basic.

## Coupling the parser and the scanner together

After all, I've finished to talk all about this and that of the values inside the parser. For the rest, I'll talking about the connecting protocol with the scanner, then the heart of this story will be all finished.

First, we'd like to make sure that I mentioned that the scanner was the `yylex()` function. each (terminal) symbol itself is returned (as `int`) as a return value of the function. Since the constants with the same names of symbols are defined (`#define`) by yacc, we can write `NUMBER` for a `NUMBER`. And its value is passed by putting it into a global variable named `yyval`. This `yyval` is also of type `YYSTYPE`, and the exactly same things as the parser can be said. In other words, if it is defined in `%union` it would become a union. But this time the member is not automatically selected, its member name has to be manually written. The very simple examples would look like the

following.

```
static int
yylex()
{
    yylval.str = next_token();
    return STRING;
}
```

Figure 2 summarizes the relationships described by now. I'd like you to check one by one. `yylval`, `$$`, `$1`, `$2` ... all of these variables that become the interfaces are of type `YYSTYPE`.

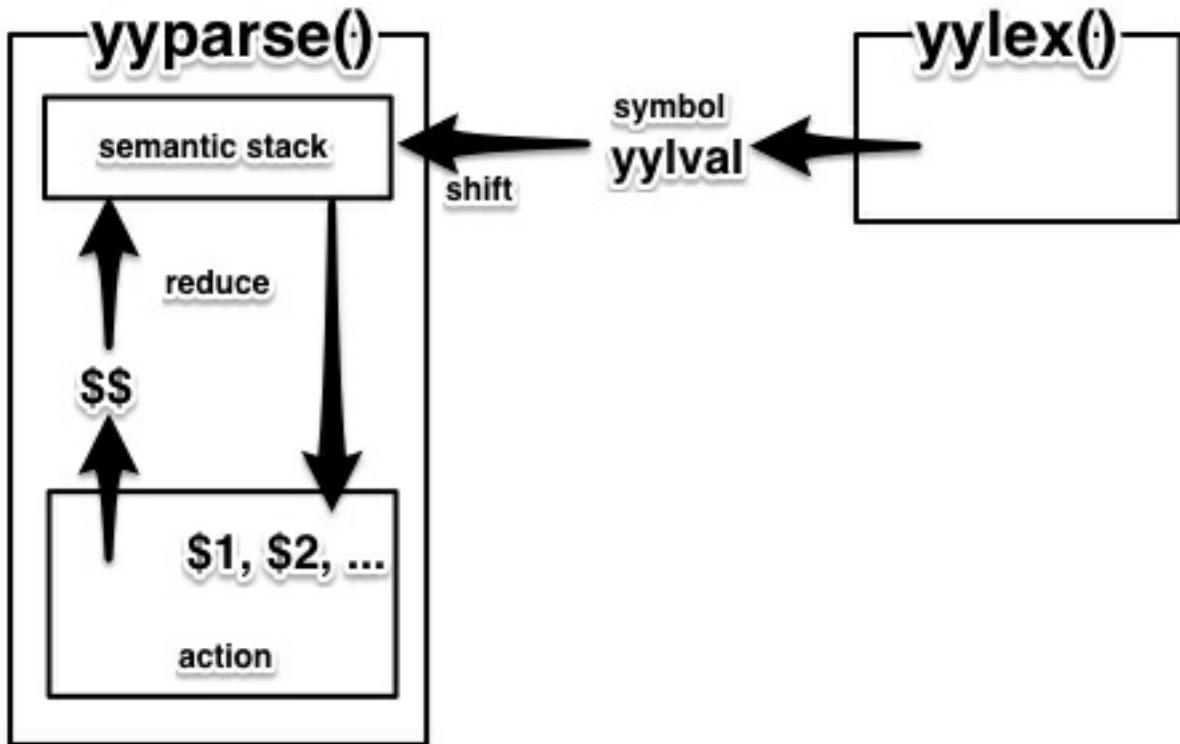


Figure 2: Relationships among yacc related variables & functions

## ■ Embedded Action

An action is written at the last of a rule, is how it was explained. However, actually it can be written in the middle of a rule.

```
target: A B { puts("embedded action"); } C D
```

This is called “embedded action”.

An embedded action is merely a syntactic sugar of the following definition:

```
target: A B dummy C D
dummy :      /* void rule */
{
    puts("embedded action");
}
```

From this example, you might be able to tell everything including when it is executed. The value of a symbol can also be taken. In other words, in this example, the value of the embedded action will come out as \$3.

## Practical Topics

---

### Conflicts

I'm not afraid of yacc anymore.

If you thought so, it is too naive. Why everyone is afraid so much

about yacc, the reason is going to be revealed.

Up until now, I wrote not so carefully “when the right-hand side of the rule matches the end of the stack”, but what happens if there’s a rule like this:

```
target : A B C
| A B C
```

When the sequence of symbols A B C actually comes out, it would be hard to determine which is the rule to match. Such thing cannot be interpreted even by humans. Therefore yacc also cannot understand this. When yacc find out an odd grammar like this, it would complain that a reduce/reduce conflict occurs. It means multiple rules are possible to reduce at the same time.

```
% yacc rrconf.y
conflicts: 1 reduce/reduce
```

But usually, I think you won’t do such things except as an accident. But how about the next example? The described symbol sequence is completely the same.

```
target : abc
| A bc
```

```
abc : A B C
```

```
bc : B C
```

This is relatively possible. Especially when each part is

complicatedly moved while developing rules, it is often the case that this kind of rules are made without noticing.

There's also a similar pattern, as follows:

```
target  : abc
| ab C
```

```
abc     : A B C
```

```
ab      : A B
```

When the symbol sequence A B C comes out, it's hard to determine whether it should choose one abc or the combination of ab and c. In this case, yacc will complain that a shift/reduce conflict occurs. This means there're both a shift-able rule and a reduce-able rule at the same time.

```
% yacc srconf.y
conflicts: 1 shift/reduce
```

The famous example of shift/reduce conflicts is “the hanging else problem”. For example, the if statement of C language causes this problem. I'll describe it by simplifying the case:

```
stmt     : expr ';'
| if
```

```
expr    : IDENTIFIER
```

```
if      : IF '(' expr ')' stmt
| IF '(' expr ')' stmt ELSE stmt
```

In this rule, the expression is only IDENTIFIER (variable), the substance of `if` is only one statement. Now, what happens if the next program is parsed with this grammar?

```
if (cond)
    if (cond)
        true_stmt;
else
    false_stmt;
```

If it is written this way, we might feel like it's quite obvious. But actually, this can be interpreted as follows.

```
if (cond) {
    if (cond)
        true_stmt;
}
else {
    false_stmt;
}
```

The question is “between the two `ifs`, inside one or outside one, which is the one to which the `else` should be attached?”.

However shift/reduce conflicts are relatively less harmful than reduce/reduce conflicts, because usually they can be solved by choosing shift. Choosing shift is almost equivalent to “connecting the elements closer to each other” and it is easy to match human instincts. In fact, the hanging `else` can also be solved by shifting it. Hence, the `yacc` follows this trend, it chooses shift by default when a shift/reduce conflict occurs.

# Look-ahead

As an experiment, I'd like you to process the next grammar with yacc.

```
%token A B C
%%
target : A B C /* rule 1 */
        | A B      /* rule 2 */
```

We can't help expecting there should be a conflict. At the time when it has read until `A B`, the rule 1 would attempt to shift, the rule 2 would attempt to reduce. In other words, this should cause a shift/reduce conflict. However, ....

```
% yacc conf.y
%
```

It's odd, there's no conflict. Why?

In fact, the parser created with `yacc` can look ahead only one symbol. Before actually doing shift or reduce, it can decide what to do by peeking the next symbol.

Therefore, it is also considered for us when generating the parser, if the rule can be determined by a single look-ahead, conflicts would be avoided. In the previous rules, for instance, if `C` comes right after `A B`, only the rule 1 is possible and it would be chose (shift). If the input has finished, the rule 2 would be chose (reduce).

Notice that the word “look-ahead” has two meanings: one thing is the look-ahead while processing `*.y` with yacc. The other thing is the look-ahead while actually executing the generated parser. The look-ahead during the execution is not so difficult, but the look-ahead of yacc itself is pretty complicated. That’s because it needs to predict all possible input patterns and decides its behaviors from only the grammar rules.

However, because “all possible” is actually impossible, it handles “most of” patterns. How broad range over all patterns it can cover up shows the strength of a look-ahead algorithm. The look-ahead algorithm that yacc uses when processing grammar files is LALR, which is relatively powerful among currently existing algorithms to resolve conflicts.

A lot things have been introduced, but you don’t have to so worry because what to do in this book is only reading and not writing. What I wanted to explain here is not the look-ahead of grammars but the look-ahead during executions.

## Operator Precedence

Since abstract talks have lasted for long, I’ll talk more concretely. Let’s try to define the rules for infix operators such as `+` or `*`. There are also established tactics for this, we’d better tamely follow it. Something like a calculator for arithmetic operations is defined below:

```
expr      : expr '+' expr
          | expr '-' expr
          | expr '*' expr
          | expr '/' expr
          | primary
```

```
primary : NUMBER
          | '(' expr ')' '
```

`primary` is the smallest grammar unit. The point is that `expr` between parentheses becomes a `primary`.

Then, if this grammar is written to an arbitrary file and compiled, the result would be this.

```
% yacc infix.y
16 shift/reduce conflicts
```

They conflict aggressively. Thinking for 5 minutes is enough to see that this rule causes a problem in the following and similar cases:

1 - 1 - 1

This can be interpreted in both of the next two ways.

(1 - 1) - 1  
1 - (1 - 1)

The former is natural as a numerical expression. But what yacc does is the process of their appearances, there does not contain any meanings. As for the things such as the meaning the - symbol has, it is absolutely not considered at all. In order to correctly reflect a

human intention, we have to specify what we want step by step.

Then, what we can do is writing this in the definition part.

```
%left '+' '-'  
%left '*' '/'
```

These instructions specifies both the precedence and the associativity at the same time.

I'll explain them in order.

I think that the term “precedence” often appears when talking about the grammar of a programming language. Describing it logically is complicated, so if I put it instinctively, it is about to which operator parentheses are attached in the following and similar cases.

1 + 2 \* 3

If \* has higher precedence, it would be this.

1 + (2 \* 3)

If + has higher precedence, it would be this.

(1 + 2) \* 3

As shown above, resolving shift/reduce conflicts by defining the stronger ones and weaker ones among operators is operator precedence.

However, if the operators has the same precedence, how can it be resolved? Like this, for instance,

1 – 2 – 3

because both operators are  $-$ , their precedences are the completely same. In this case, it is resolved by using the associativity. Associativity has three types: left right nonassoc, they will be interpreted as follows:

Associativity	Interpretation
left (left-associative)	$(1 - 2) - 3$
right (right-associative)	$1 - (2 - 3)$
nonassoc (non-associative)	parse error

Most of the operators for numerical expressions are left-associative. The right-associative is used mainly for  $=$  of assignment and  $\text{not}$  of denial.

```
a = b = 1    # (a = (b = 1))
not not a    # (not (not a))
```

The representatives of non-associative are probably the comparison operators.

```
a == b == c    # parse error
a <= b <= c    # parse error
```

However, this is not the only possibility. In Python, for instance, comparisons between three terms are possible.

Then, the previous instructions named %left %right %noassoc are used to specify the associativities of their names. And, precedence is specified as the order of the instructions. The lower the operators written, the higher the precedences they have. If they are written in the same line, they have the same level of precedence.

```
%left  '+' '-'  /* left-associative and third precedence */
%left  '*' '/'  /* left-associative and second precedence */
%right '!'      /* right-associative and first precedence */
```

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](#)

# Ruby Hacking Guide

Translated by Robert GRAVINA & ocha-

# Chapter 10: Parser

## Outline of this chapter

---

### Parser construction

The main source of the parser is `parser.y`. Because it is `*.y`, it is the input for `yacc` and `parse.c` is generated from it.

Although one would expect `lex.c` to contain the scanner, this is not the case. This file is created by `gperf`, taking the file `keywords` as input, and defines the reserved word hashtable. This tool-generated `lex.c` is `#included` in (the also tool-generated) `parse.c`. The details of this process is somewhat difficult to explain at this time, so we shall return to this later.

Figure 1 shows the parser construction process. For the benefit of those readers using Windows who may not be aware, the `mv` (move) command creates a new copy of a file and removes the original. `cc` is, of course, the C compiler and `cpp` the C pre-processor.

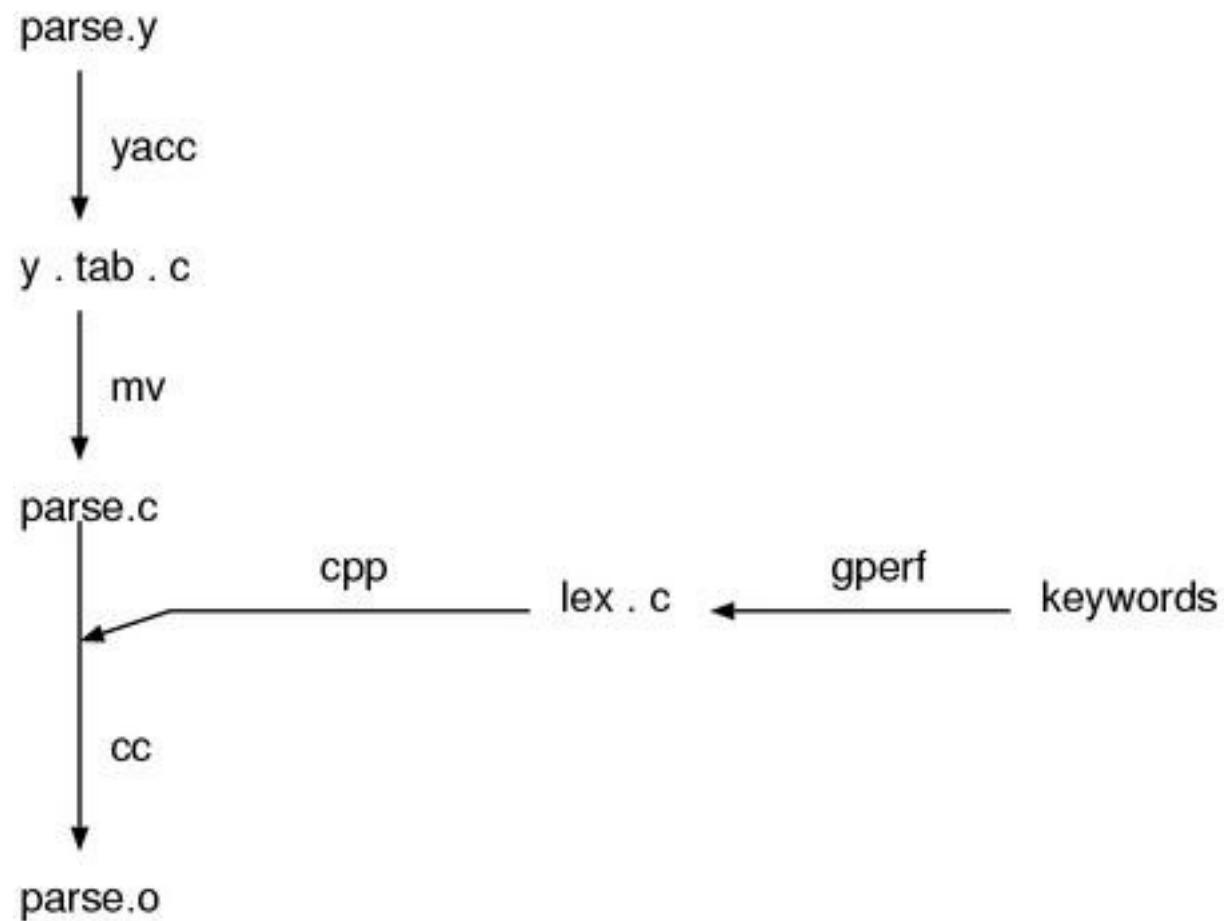


Figure 1: Parser construction process

## ■ Dissecting `parse.y`

Let's now look at `parse.y` in a bit more detail. The following figure presents a rough outline of the contents of `parse.y`.

### ▼ `parse.y`

```

%{
header
%}
%union ....
%token ....
%type ....
%%
rules
  
```

%%

user code section  
parser interface  
scanner (character stream processing)  
syntax tree construction  
semantic analysis  
local variable management  
ID implementation

As for the rules and definitions part, it is as previously described. Since this part is indeed the heart of the parser, I'll start to explain it ahead of the other parts in the next section.

There are a considerable number of support functions defined in the user code section, but roughly speaking, they can be divided into the six parts written above. The following table shows where each of parts are explained in this book.

<b>Part</b>	<b>Chapter</b>	<b>Section</b>
Parser interface	This chapter	Section 3 “Scanning”
Scanner	This chapter	Section 3 “Scanning”
Syntax tree construction	Chapter 12 “Syntax tree construction”	Section 2 “Syntax tree construction”
Semantic analysis	Chapter 12 “Syntax tree construction”	Section 3 “Semantic analysis”
Local variable management	Chapter 12 “Syntax tree construction”	Section 4 “Local variables”
ID implementation	Chapter 3 “Names and name tables”	Section 2 “ID and symbols”

# General remarks about grammar rules

## Coding rules

The grammar of `ruby` conforms to a coding standard and is thus easy to read once you are familiar with it.

Firstly, regarding symbol names, all non-terminal symbols are written in lower case characters. Terminal symbols are prefixed by some lower case character and then followed by upper case. Reserved words (keywords) are prefixed with the character `k`. Other terminal symbols are prefixed with the character `t`.

### ▼ Symbol name examples

Token (non-terminal symbol)	Symbol name
<code>if</code>	<code>kIF</code>
<code>def</code>	<code>kDEF</code>
<code>rescue</code>	<code>kRESCUE</code>
<code>varname</code>	<code>tIDENTIFIER</code>
<code>ConstName</code>	<code>tCONST</code>
<code>1</code>	<code>tINTEGER</code>

The only exceptions to these rules are `kLBEGIN` and `kLEND`. These symbol names refer to the reserved words for “BEGIN” and “END”, respectively, and the `l` here stands for `large`. Since the reserved words `begin` and `end` already exist (naturally, with symbol names `kBEGIN` and `kEND`), these non-standard symbol names were required.

# Important symbols

parse.y contains both grammar rules and actions, however, for now I would like to concentrate on the grammar rules alone. The script sample/exyacc.rb can be used to extract the grammar rules from this file. Aside from this, running yacc -v will create a logfile y.output which also contains the grammar rules, however it is rather difficult to read. In this chapter I have used a slightly modified version of exyacc.rb\footnote{modified exyacc.rb:tools/exyacc2.rb located on the attached CD-ROM} to extract the grammar rules.

## ▼ parse.y(rules)

```
program      : compstmt
bodystmt     : compstmt
              opt_rescue
              opt_else
              opt_ensure
compstmt     : stmts opt_terms
              :
              :
```

The output is quite long – over 450 lines of grammar rules – and as such I have only included the most important parts in this chapter.

Which symbols, then, are the most important? The names such as program, expr, stmt, primary, arg etc. are always very important. It's

because they represent the general parts of the grammatical elements of a programming language. The following table outlines the elements we should generally focus on in the syntax of a program.

<b>Syntax element</b>	<b>Predicted symbol names</b>
Program	program prog file input stmts whole
Sentence	statement stmt
Expression	expression expr exp
Smallest element	primary prim
Left hand side of an expression	lhs(left hand side)
Right hand side of an expression	rhs(right hand side)
Function call	funcall function_call call function
Method call	method method_call call
Argument	argument arg
Function definition	defun definition function fndef
Declarations	declaration decl

In general, programming languages tend to have the following hierarchy structure.

<b>Program element</b>	<b>Properties</b>
Program	Usually a list of statements
Statement	What can not be combined with the others. A syntax tree trunk.
Expression	What is a combination by itself and can also be a part of another expression. A syntax tree internal node.

**Primary** An element which can not be further decomposed. A syntax tree leaf node.

The statements are things like function definitions in C or class definitions in Java. An expression can be a procedure call, an arithmetic expression etc., while a primary usually refers to a string literal or number. Some languages do not contain all of these symbol types, however they generally contain some kind of hierarchy of symbols such as `program→stmt→expr→primary`.

However, a structure at a low level can be contained by a superior structure. For example, in C a function call is an expression but it can solely be put. It means it is an expression but it can also be a statement.

Conversely, when surrounded in parentheses, expressions become primaries. It is because the lower the level of an element the higher the precedence it has.

The range of statements differ considerably between programming languages. Let's consider assignment as an example. In C, because it is part of expressions, we can use the value of the whole assignment expression. But in Pascal, assignment is a statement, we cannot do such thing. Also, function and class definitions are typically statements however in languages such as Lisp and Scheme, since everything is an expression, they do not have statements in the first place. Ruby is close to Lisp's design in this regard.

# Program structure

Now let's turn our attention to the grammar rules of ruby. Firstly, in yacc, the left hand side of the first rule represents the entire grammar. Currently, it is `program`. Following further and further from here, as the same as the established tactic, the four `program` `stmt` `expr` `primary` will be found. With adding `arg` to them, let's look at their rules.

## ▼ ruby grammar (outline)

```
program      : compstmt
compstmt     : stmts opt_terms
stmts        : none
             | stmt
             | stmts terms stmt
stmt         : kALIAS fitem fitem
             | kALIAS tGVAR tGVAR
             :
             :
             | expr
expr         : kRETURN call_args
             | kBREAK call_args
             :
             :
             | '!' command_call
             | arg
arg          : lhs '=' arg
             | var_lhs t0P_ASSIGN arg
             | primary_value '[' aref_args ']' t0P_ASSIGN arg
             :
             :
```

```

| arg '?' arg ':' arg
| primary

primary      : literal
| strings
:
:
| tLPAREN_ARG expr  ')'
| tLPAREN compstmt ')'
:
:
| kREDO
| kRETRY

```

If we focus on the last rule of each element, we can clearly make out a hierarchy of `program`→`stmt`→`expr`→`arg`→ `primary`.

Also, we'd like to focus on this rule of `primary`.

```

primary      : literal
:
:
| tLPAREN_ARG expr  ')'      /* here */

```

The name `tLPAREN_ARG` comes from `t` for terminal symbol, `L` for left and `PAREN` for parentheses – it is the open parenthesis. Why this isn't `'('` is covered in the next section “Context-dependent scanner”. Anyway, the purpose of this rule is demote an `expr` to a `primary`. This creates a cycle which can be seen in Figure 2, and the arrow shows how this rule is reduced during parsing.

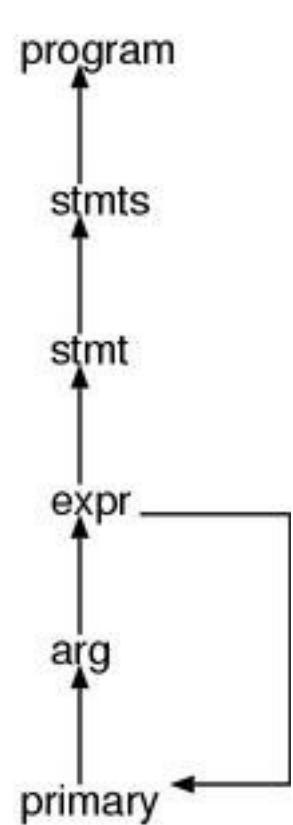


Figure 2: expr demotion

The next rule is also particularly interesting.

```

primary      : literal
              :
              :
              | tLPAREN compstmt ')'* /* here */
  
```

A `compstmt`, which equals to the entire program (`program`), can be demoted to a `primary` with this rule. The next figure illustrates this rule in action.

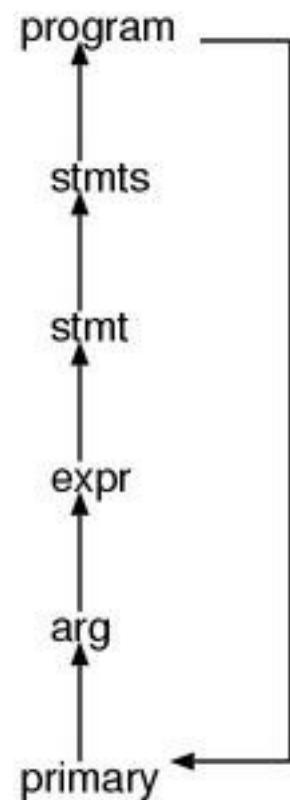


Figure 3: program demotion

This means that for any syntax element in Ruby, if we surround it with parenthesis it will become a `primary` and can be passed as an argument to a function, be used as the right hand side of an expression etc. This is an incredible fact. Let's actually confirm it.

```

p((class C; end))
p((def a() end))
p((alias ali gets))
p((if true then nil else nil end))
p((1 + 1 * 1 ** 1 - 1 / 1 ^ 1))
  
```

If we invoke `ruby` with the `-c` option (syntax check), we get the following output.

```

% ruby -c primprog.rb
Syntax OK
  
```

Indeed, it's hard to believe but, it could actually pass. Apparently, we did not get the wrong idea.

If we care about the details, since there are what rejected by the semantic analysis (see also Chapter 12 “Syntax tree construction”), it is not perfectly possible. For example passing a `return` statement as an argument to a function will result in an error. But at least at the level of the outlooks, the “surrounding anything in parenthesis means it can be passed as an argument to a function” rule does hold.

In the next section I will cover the contents of the important elements one by one.

## program

### ▼ program

```
program      : compstmt
compstmt     : stmts opt_terms
stmts        : none
             | stmt
             | stmts terms stmt
```

As mentioned earlier, `program` represents the entire grammar that means the entire program. That `program` equals to `compstmts`, and `compstmts` is almost equivalent to `stmts`. That `stmts` is a list of `stmts` delimited by `terms`. Hence, the entire program is a list of `stmts` delimited by `terms`.

terms is (of course) an abbreviation for “terminators”, the symbols that terminate the sentences, such as semicolons or newlines. opt\_terms means “OPTIONal terms”. The definitions are as follows:

### ▼ opt\_terms

```
opt_terms      : terms
               | terms
terms         : term
               | terms ';'
term          : ';'
               | '\n'
```

The initial ; or \n of a terms can be followed by any number of ; only; based on that, you might start thinking that if there are 2 or more consecutive newlines, it could cause a problem. Let's try and see what actually happens.

```
1 + 1  # first newline
       # second newline
       # third newline
1 + 1
```

Run that with ruby -c.

```
% ruby -c optterms.rb
Syntax OK
```

Strange, it worked! What actually happens is this: consecutive newlines are simply discarded by the scanner, which returns only

the first newline in a series.

By the way, although we said that `program` is the same as `compstmt`, if that was really true, you would question why `compstmt` exists at all. Actually, the distinction is there only for execution of semantic actions. `program` exists to execute any semantic actions which should be done once in the processing of an entire program. If it was only a question of parsing, `program` could be omitted with no problems at all.

To generalize this point, the grammar rules can be divided into 2 groups: those which are needed for parsing the program structure, and those which are needed for execution of semantic actions. The `none` rule which was mentioned earlier when talking about `stmts` is another one which exists for executing actions — it's used to return a `NULL` pointer for an empty list of type `NODE*`.

## ▀ `stmt`

Next is `stmt`. This one is rather involved, so we'll look into it a bit at a time.

### ▼ `stmt(1)`

```
stmt      : kALIAS fitem  fitem
          | kALIAS tGVAR tGVAR
          | kALIAS tGVAR tBACK_REF
          | kALIAS tGVAR tNTH_REF
          | kUNDEF undef_list
          | stmt kIF_MOD expr_value
          | stmt kUNLESS_MOD expr_value
```

```

| stmt kWHILE_MOD expr_value
| stmt kUNTIL_MOD expr_value
| stmt kRESCUE_MOD stmt
| klBEGIN '{' compstmt '}'
| klEND '{' compstmt '}'

```

Looking at that, somehow things start to make sense. The first few have `alias`, then `undef`, then the next few are all something followed by `_MOD` — those should be statements with postposition modifiers, as you can imagine.

`expr_value` and `primary_value` are grammar rules which exist to execute semantic actions. For example, `expr_value` represents an `expr` which has a value. Expressions which don't have values are `return` and `break`, or `return/break` followed by a postposition modifier, such as an `if` clause. For a detailed definition of what it means to “have a value”, see chapter 12, “Syntax Tree Construction”. In the same way, `primary_value` is a `primary` which has a value.

As explained earlier, `klBEGIN` and `klEND` represent `BEGIN` and `END`.

▼ `stmt(2)`

```

| lhs '=' command_call
| mlhs '=' command_call
| var_lhs t0P_ASgn command_call
| primary_value '[' aref_args ']' t0P_ASgn command
| primary_value '.' tIDENTIFIER t0P_ASgn command_c
| primary_value '.' tCONSTANT t0P_ASgn command_c
| primary_value tCOLON2 tIDENTIFIER t0P_ASgn comma
| backref t0P_ASgn command_call

```

Looking at these rules all at once is the right approach. The common point is that they all have `command_call` on the right-hand side. `command_call` represents a method call with the parentheses omitted. The new symbols which are introduced here are explained in the following table. I hope you'll refer to the table as you check over each grammar rule.

<code>lhs</code>	the left hand side of an assignment (Left Hand Side)
<code>mlhs</code>	the left hand side of a multiple assignment (Multiple Left Hand Side)
<code>var_lhs</code>	the left hand side of an assignment to a kind of variable (VARiable Left Hand Side)
<code>top_ASGN</code>	compound assignment operator like <code>+=</code> or <code>*=</code> (OPerator ASsIGN)
<code>aref_args</code>	argument to a <code>[]</code> method call (Array REFerence)
<code>tIDENTIFIER</code>	identifier which can be used as a local variable
<code>tCONSTANT</code>	constant identifier (with leading uppercase letter)
<code>tCOLON2</code>	<code>::</code>
<code>backref</code>	<code>\$1 \$2 \$3...</code>

`aref` is a `Lisp` jargon. There's also `aset` as the other side of a pair, which is an abbreviation of “array set”. This abbreviation is used at a lot of places in the source code of `ruby`.

## ▼ `stmt` (3)

```
| lhs '=' mrhs_basic
| mlhs '=' mrhs
```

These two are multiple assignments. `mrhs` has the same structure as

`mlhs` and it means multiple `rhs` (the right hand side). We've come to recognize that knowing the meanings of names makes the comprehension much easier.

## ▼ `stmt` (4)

```
| expr
```

Lastly, it joins to `expr`.

## expr

### ▼ `expr`

```
expr      : kRETURN call_args
          | kBREAK call_args
          | kNEXT call_args
          | command_call
          | expr kAND expr
          | expr kOR expr
          | kNOT expr
          | '!' command_call
          | arg
```

Expression. The expression of `ruby` is very small in grammar. That's because those ordinary contained in `expr` are mostly went into `arg`. Conversely speaking, those who could not go to `arg` are left here. And what are left are, again, method calls without parentheses. `call_args` is an bare argument list, `command_call` is, as previously mentioned, a method without parentheses. If this kind of things was contained in the “small” unit, it would cause conflicts

tremendously.

However, these two below are of different kind.

```
expr kAND expr
expr k0R expr
```

kAND is “and”, and k0R is “or”. Since these two have their roles as control structures, they must be contained in the “big” syntax unit which is larger than `command_call`. And since `command_call` is contained in `expr`, at least they need to be `expr` to go well. For example, the following usage is possible ...

```
valid_items.include? arg or raise ArgumentError, 'invalid arg'
# valid_items.include?(arg) or raise(ArgumentError, 'invalid arg')
```

However, if the rule of k0R existed in `arg` instead of `expr`, it would be joined as follows.

```
valid_items.include?((arg or raise)) ArgumentError, 'invalid arg'
```

Obviously, this would end up a parse error.

█ arg

▼ arg

```
arg           : lhs '=' arg
| var_lhs t0P_ASgn arg
| primary_value '[' aref_args ']' t0P_ASgn arg
| primary_value '.' tIDENTIFIER t0P_ASgn arg
```

```

| primary_value '.' tCONSTANT t0P_ASgn arg
| primary_value tCOLON2 tIDENTIFIER t0P_ASgn arg
| backref t0P_ASgn arg
| arg tDOT2 arg
| arg tDOT3 arg
| arg '+' arg
| arg '-' arg
| arg '*' arg
| arg '/' arg
| arg '%' arg
| arg tPOW arg
| tUPLUS arg
| tUMINUS arg
| arg '|' arg
| arg '^' arg
| arg '&' arg
| arg tCMP arg
| arg '>' arg
| arg tGEQ arg
| arg '<' arg
| arg tLEQ arg
| arg tEQ arg
| arg tEQQ arg
| arg tNEQ arg
| arg tMATCH arg
| arg tNMATCH arg
| '!' arg
| '~' arg
| arg tLShFT arg
| arg tRShFT arg
| arg tANDOP arg
| arg tOROP arg
| kDEFINED opt_nl arg
| arg '?' arg ':' arg
| primary

```

Although there are many rules here, the complexity of the grammar is not proportionate to the number of rules. A grammar that merely has a lot of cases can be handled very easily by yacc, rather, the depth or recursive of the rules has more influences the

complexity.

Then, it makes us curious about the rules are defined recursively in the form of `arg OP arg` at the place for operators, but because for all of these operators their operator precedences are defined, this is virtually only a mere enumeration. Let's cut the “mere enumeration” out from the `arg` rule by merging.

```
arg: lhs '=' arg          /* 1 */
  | primary T_opeq arg   /* 2 */
  | arg T_infix arg      /* 3 */
  | T_pre arg             /* 4 */
  | arg '?' arg ':' arg  /* 5 */
  | primary               /* 6 */
```

There's no meaning to distinguish terminal symbols from lists of terminal symbols, they are all expressed with symbols with `T_`. `opeq` is operator + equal, `T_pre` represents the prepositional operators such as `!!` and `~`, `T_infix` represents the infix operators such as `*` and `%`.

To avoid conflicts in this structure, things like written below become important (but, these does not cover all).

- `T_infix` should not contain `=`.

Since `args` partially overlaps `lhs`, if `=` is contained, the rule 1 and the rule 3 cannot be distinguished.

- `T_opeq` and `T_infix` should not have any common rule.

Since `args` contains `primary`, if they have any common rule, the rule 2 and the rule 3 cannot be distinguished.

- `T_infix` should not contain '?'.

If it contains, the rule 3 and 5 would produce a shift/reduce conflict.

- `T_pre` should not contain '?' or ':'.

If it contains, the rule 4 and 5 would conflict in a very complicated way.

The conclusion is all requirements are met and this grammar does not conflict. We could say it's a matter of course.

## ■ `primary`

Because `primary` has a lot of grammar rules, we'll split them up and show them in parts.

### ▼ `primary (1)`

```
primary      : literal
              | strings
              | xstring
              | regexp
              | words
              | qwords
```

Literals. `literal` is for symbol literals (`:sym`) and numbers.

## ▼ primary (2)

```
| var_ref
| backref
| tFID
```

Variables. `var_ref` is for local variables and instance variables and etc. `backref` is for `$1 $2 $3 ...` `tFID` is for the identifiers with `!` or `?`, say, `include?` `reject!`. There's no possibility of `tFID` being a local variable, even if it appears solely, it becomes a method call at the parser level.

## ▼ primary (3)

```
| kBEGIN
| bodystmt
| kEND
```

`bodystmt` contains `rescue` and `ensure`. It means this is the `begin` of the exception control.

## ▼ primary (4)

```
| tLPAREN_ARG expr ')'
| tLPAREN compstmt ')'
```

This has already described. Syntax demoting.

## ▼ primary (5)

```
| primary_value tCOLON2 tCONSTANT
```

```
| tCOLON3 cname
```

Constant references. tCONSTANT is for constant names (capitalized identifiers).

Both tCOLON2 and tCOLON3 are ::, but tCOLON3 represents only the :: which means the top level. In other words, it is the :: of ::Const. The :: of Net::SMTP is tCOLON2.

The reason why different symbols are used for the same token is to deal with the methods without parentheses. For example, it is to distinguish the next two from each other:

```
p Net::HTTP    # p(Net::HTTP)
p Net  ::HTTP  # p(Net(::HTTP))
```

If there's a space or a delimiter character such as an open parenthesis just before it, it becomes tCOLON3. In the other cases, it becomes tCOLON2.

▼ primary (6)

```
| primary_value '[' aref_args ']'
```

Index-form calls, for instance, arr[i].

▼ primary (7)

```
| tLBRACK aref_args ']'
| tLBRACE assoc_list '}'
```

Array literals and Hash literals. This tLBRACK represents also '['. '[]' means a '[' without a space in front of it. The necessity of this differentiation is also a side effect of method calls without parentheses.

The terminal symbols of this rule is very incomprehensible because they differs in just a character. The following table shows how to read each type of parentheses, so I'd like you to make use of it when reading.

## ▼ English names for each parentheses

Symbol English Name

(	parentheses
{}	braces
[]	brackets

## ▼ primary (8)

```
| kRETURN
| kYIELD '(' call_args ')'
| kYIELD '(' ')'
| kYIELD
| kDEFINED opt_nl '(' expr ')'
```

Syntaxes whose forms are similar to method calls. Respectively, `return`, `yield`, `defined?`.

There arguments for `yield`, but `return` does not have any arguments. Why? The fundamental reason is that `yield` itself has its return value but `return` does not. However, even if there's not

any arguments here, it does not mean you cannot pass values, of course. There was the following rule in `expr`.

```
kRETURN call_args
```

`call_args` is a bare argument list, so it can deal with `return 1` or `return nil`. Things like `return(1)` are handled as `return (1)`. For this reason, surrounding the multiple arguments of a `return` with parentheses as in the following code should be impossible.

```
return(1, 2, 3)  # interpreted as return (1,2,3) and results in
```

You could understand more about around here if you will check this again after reading the next chapter “Finite-State Scanner”.

## ▼ primary (9)

```
| operation brace_block
| method_call
| method_call brace_block
```

Method calls. `method_call` is with arguments (also with parentheses), `operation` is without both arguments and parentheses, `brace_block` is either `{ ~ }` or `do ~ end` and if it is attached to a method, the method is an iterator. For the question “Even though it is brace, why is `do ~ end` contained in it?”, there’s a reason that is more abyssal than Marian Trench, but again the only way to understand is reading the next chapter “Finite-State Scanner”.

▼ primary (10)

```
| kIF expr_value then compstmt if_tail kEND          # if
| kUNLESS expr_value then compstmt opt_else kEND    # unless
| kWHILE expr_value do compstmt kEND                # while
| kUNTIL expr_value do compstmt kEND                # until
| kCASE expr_value opt_terms case_body kEND          # case
| kCASE opt_terms case_body kEND                    # case(Form2)
| kFOR block_var kIN expr_value do compstmt kEND    # for
```

The basic control structures. A little unexpectedly, things appear to be this big are put inside `primary`, which is “small”. Because `primary` is also `arg`, we can also do something like this.

```
p(if true then 'ok' end)  # shows "ok"
```

I mentioned “almost all syntax elements are expressions” was one of the traits of Ruby. It is concretely expressed by the fact that `if` and `while` are in `primary`.

Why is there no problem if these “big” elements are contained in primary? That’s because the Ruby’s syntax has a trait that “it begins with the terminal symbol A and ends with the terminal symbol B”. In the next section, we’ll think about this point again.

## ▼ primary (11)

Definition statements. I've called them the class statements and the class statements, but essentially I should have been called them the class primaries, probably. These are all fit the pattern “beginning with the terminal symbol A and ending with B”, even if such rules are increased a lot more, it would never be a problem.

## ▼ primary (12)

```
| kBREAK
| kNEXT
| kREDO
| kRETRY
```

Various jumps. These are, well, not important from the viewpoint of grammar.

## ■ Conflicting Lists

In the previous section, the question “is it all right that `if` is in such primary?” was suggested. To proof precisely is not easy, but explaining instinctively is relatively easy. Here, let's simulate with a small rule defined as follows:

```
%token A B o
%%
element  : A item_list B
item_list :
          | item_list item
item     : element
```

| o

element is the element that we are going to examine. For example, if we think about `if`, it would be `if`. `element` is a list that starts with the terminal symbol `A` and ends with `B`. As for `if`, it starts with `if` and ends with `end`. The `o` contents are methods or variable references or literals. For an element of the list, the `o` or `element` is nesting.

With the parser based on this grammar, let's try to parse the following input.

A A o o o B o A o A o o o B o B B

They are nesting too many times for humans to comprehend without some helps such as indents. But it becomes relatively easy if you think in the next way. Because it's certain that `A` and `B` which contain only several `o` between them are going to appear, replace them to a single `o` when they appear. All we have to do is repeating this procedure. Figure 4 shows the consequence.

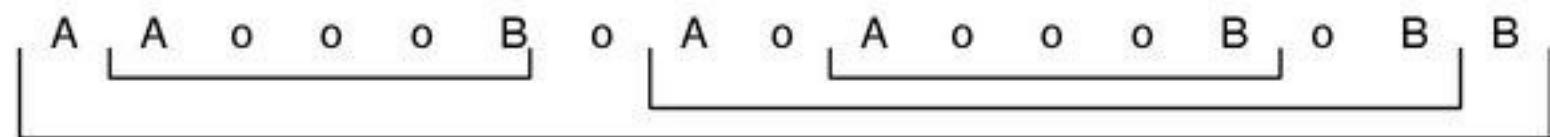


Figure 4: parse a list which starts with A and ends with B

However, if the ending `B` is missing, ...

%token A o

```
%%
element  : A item_list    /* B is deleted for an experiment */

item_list :
| item_list item

item    : element
| o
```

I processed this with yacc and got 2 shift/reduce conflicts. It means this grammar is ambiguous. If we simply take B out from the previous one, The input would be as follows.

A A o o o o A o A o o o o

This is hard to interpret in any way. However, there was a rule that “choose shift if it is a shift/reduce conflict”, let’s follow it as an experiment and parse the input with shift (meaning interior) which takes precedence. (Figure 5)

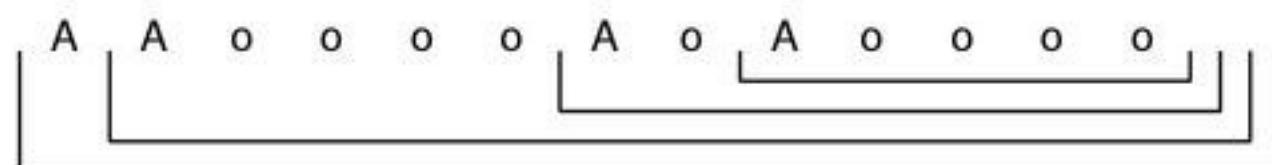


Figure 5: parse a list of lists which start with A

It could be parsed. However, this is completely different from the intention of the input, there becomes no way to split the list in the middle.

Actually, the methods without parentheses of Ruby is in the similar situation to this. It’s not so easy to understand but a pair of

a method name and its first argument is `A`. This is because, since there's no comma only between the two, it can be recognized as the start of a new list.

Also, the “practical” HTML contains this pattern. It is, for instance, when `</p>` or `</i>` is omitted. That's why `yacc` could not be used for ordinary HTML at all.

## Scanner

---

## Parser Outline

I'll explain about the outline of the parser before moving on to the scanner. Take a look at Figure 6.

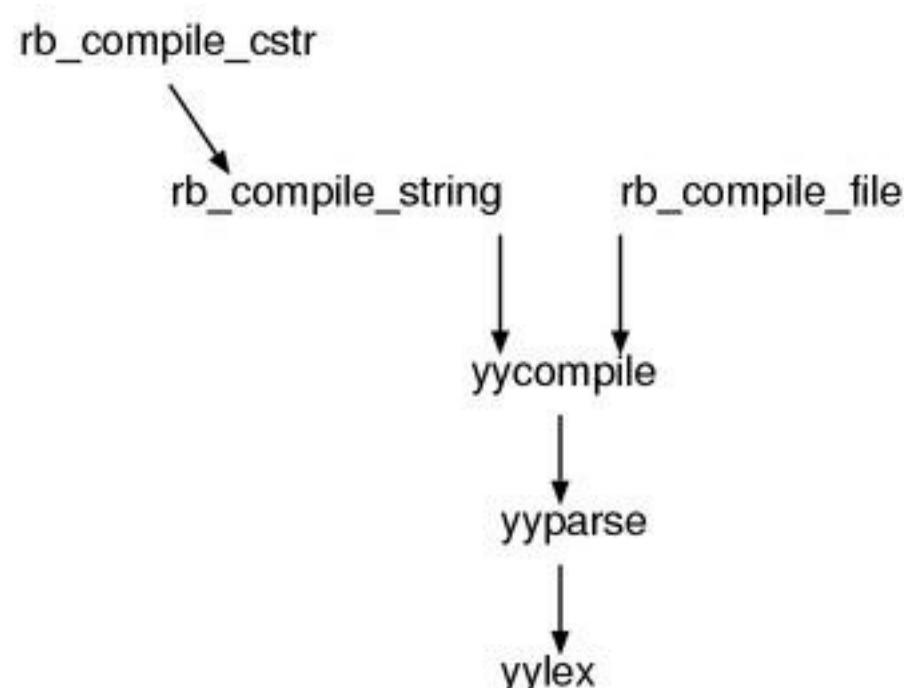


Figure 6: Parser Interface (Call Graph)

There are three official interfaces of the parser: `rb_compile_cstr()`, `rb_compile_string()`, `rb_compile_file()`. They read a program from C string, a Ruby string object and a Ruby `IO` object, respectively, and compile it.

These functions, directly or indirectly, call `yycompile()`, and in the end, the control will be completely moved to `yyparse()`, which is generated by yacc. Since the heart of the parser is nothing but `yyparse()`, it's nice to understand by placing `yyparse()` at the center. In other words, functions before moving on to `yyparse()` are all preparations, and functions after `yyparse()` are merely chore functions being pushed around by `yyparse()`.

The rest functions in `parse.y` are auxiliary functions called by `yylex()`, and these can also be clearly categorized.

First, the input buffer is at the lowest level of the scanner. `ruby` is designed so that you can input source programs via both Ruby `IO` objects and strings. The input buffer hides that and makes it look like a single byte stream.

The next level is the token buffer. It reads 1 byte at a time from the input buffer, and keeps them until it will form a token.

Therefore, the whole structure of `yylex` can be depicted as Figure 7.

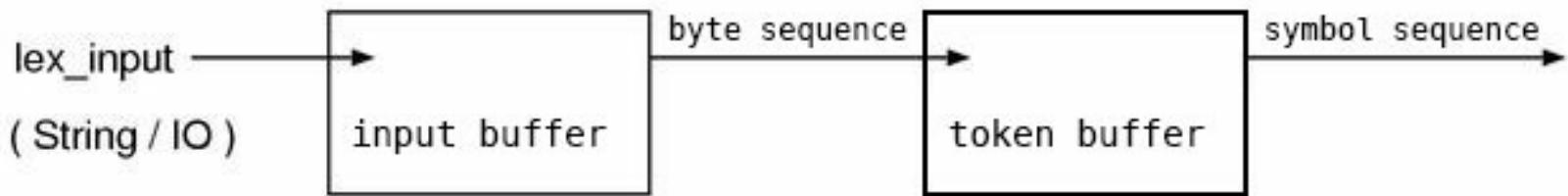


Figure 7: The whole picture of the scanner

## The input buffer

Let's start with the input buffer. Its interfaces are only the three: `nextc()`, `pushback()`, `peek()`.

Although this is sort of insistent, I said the first thing is to investigate data structures. The variables used by the input buffer are the followings:

### ▼ the input buffer

```

2279 static char *lex_pbeg;
2280 static char *lex_p;
2281 static char *lex_pend;

(parse.y)
  
```

The beginning, the current position and the end of the buffer. Apparently, this buffer seems a simple single-line string buffer (Figure 8).

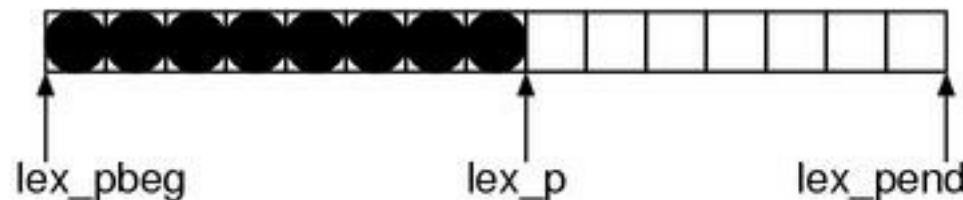


Figure 8: The input buffer

## nextc()

Then, let's look at the places using them. First, I'll start with `nextc()` that seems the most orthodox.

### ▼ `nextc()`

```
2468 static inline int
2469 nextc()
2470 {
2471     int c;
2472
2473     if (lex_p == lex_pend) {
2474         if (lex_input) {
2475             VALUE v = lex_getline();
2476
2477             if (NIL_P(v)) return -1;
2478             if (heredoc_end > 0) {
2479                 ruby_sourceline = heredoc_end;
2480                 heredoc_end = 0;
2481             }
2482             ruby_sourceline++;
2483             lex_pbeg = lex_p = RSTRING(v)->ptr;
2484             lex_pend = lex_p + RSTRING(v)->len;
2485             lex_lastline = v;
2486         }
2487         else {
2488             lex_lastline = 0;
2489             return -1;
2490         }
2491     }
2492     c = (unsigned char)*lex_p++;
2493     if (c == '\r' && lex_p <= lex_pend && *lex_p == '\n') {
2494         lex_p++;
2495         c = '\n';
2496     }
2497
2498     return c;
2499 }
```

It seems that the first `if` is to test if it reaches the end of the input buffer. And, the `if` inside of it seems, since the `else` returns `-1 (EOF)`, to test the end of the whole input. Conversely speaking, when the input ends, `lex_input` becomes `0`. ((errata: it does not. `lex_input` will never become `0` during ordinary scan.))

From this, we can see that strings are coming bit by bit into the input buffer. Since the name of the function which updates the buffer is `lex_getline`, it's definite that each line comes in at a time.

Here is the summary:

```

if ( reached the end of the buffer )
    if (still there's more input)
        read the next line
    else
        return EOF
move the pointer forward
skip reading CR of CRLF
return c

```

Let's also look at the function `lex_getline()`, which provides lines. The variables used by this function are shown together in the following.

## ▼ `lex_getline()`

```

2276 static VALUE (*lex_gets)();      /* gets function */
2277 static VALUE lex_input;          /* non-nil if File */
2420 static VALUE

```

```
2421 lex_getline()
2422 {
2423     VALUE line = (*lex_gets)(lex_input);
2424     if (ruby_debug_lines && !NIL_P(line)) {
2425         rb_ary_push(ruby_debug_lines, line);
2426     }
2427     return line;
2428 }
```

(parse.y)

Except for the first line, this is not important. Apparently, `lex_gets` should be the pointer to the function to read a line, `lex_input` should be the actual input. I searched the place where setting `lex_gets` and this is what I found:

## ▼ set `lex_gets`

```
2430 NODE*
2431 rb_compile_string(f, s, line)
2432     const char *f;
2433     VALUE s;
2434     int line;
2435 {
2436     lex_gets = lex_get_str;
2437     lex_gets_ptr = 0;
2438     lex_input = s;

2454 NODE*
2455 rb_compile_file(f, file, start)
2456     const char *f;
2457     VALUE file;
2458     int start;
2459 {
2460     lex_gets = rb_io_gets;
2461     lex_input = file;
```

(parse.y)

`rb_io_gets()` is not a exclusive function for the parser but one of the general-purpose library of Ruby. It is the function to read a line from an `IO` object.

On the other hand, `lex_get_str()` is defined as follows:

▼ `lex_get_str()`

```
2398 static int lex_gets_ptr;  
  
2400 static VALUE  
2401 lex_get_str(s)  
2402     VALUE s;  
2403 {  
2404     char *beg, *end, *pend;  
2405  
2406     beg = RSTRING(s)->ptr;  
2407     if (lex_gets_ptr) {  
2408         if (RSTRING(s)->len == lex_gets_ptr) return Qnil;  
2409         beg += lex_gets_ptr;  
2410     }  
2411     pend = RSTRING(s)->ptr + RSTRING(s)->len;  
2412     end = beg;  
2413     while (end < pend) {  
2414         if (*end++ == '\n') break;  
2415     }  
2416     lex_gets_ptr = end - RSTRING(s)->ptr;  
2417     return rb_str_new(beg, end - beg);  
2418 }
```

(parse.y)

`lex_gets_ptr` remembers the place it have already read. This moves it to the next `\n`, and simultaneously cut out at the place and return it.

Here, let's go back to `nextc`. As described, by preparing the two functions with the same interface, it switch the function pointer when initializing the parser, and the other part is used in common. It can also be said that the difference of the code is converted to the data and absorbed. There was also a similar method of `st_table`.

## pushback()

With the knowledge of the physical structure of the buffer and `nextc`, we can understand the rest easily. `pushback()` writes back a character. If put it in C, it is `ungetc()`.

### ▼ pushback()

```
2501 static void
2502 pushback(c)
2503     int c;
2504 {
2505     if (c == -1) return;
2506     lex_p--;
2507 }
```

(`parse.y`)

## peek()

`peek()` checks the next character without moving the pointer forward.

### ▼ peek()

```
2509 #define peek(c) (lex_p != lex_pend && (c) == *lex_p)  
(parse.y)
```

## ▀ The Token Buffer

The token buffer is the buffer of the next level. It keeps the strings until a token will be able to cut out. There are the five interfaces as follows:

newtok begin a new token  
tokadd add a character to the buffer  
tokfix fix a token  
tok the pointer to the beginning of the buffered string  
toklen the length of the buffered string  
toklast the last byte of the buffered string

Now, we'll start with the data structures.

### ▼ The Token Buffer

```
2271 static char *tokenbuf = NULL;  
2272 static int    tokidx, toksiz = 0;  
(parse.y)
```

tokenbuf is the buffer, tokidx is the end of the token (since it is of int, it seems an index), and toksiz is probably the buffer length. This is also simply structured. If depicting it, it would look like Figure 9.

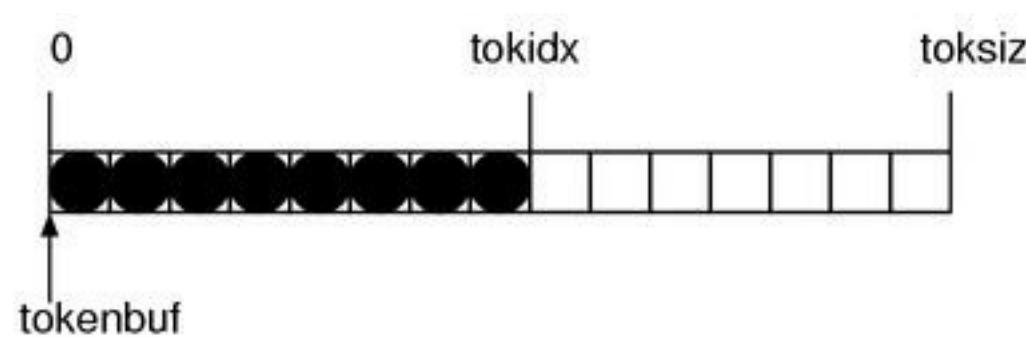


Figure 9: The token buffer

Let's continuously go to the interface and read `newtok()`, which starts a new token.

### ▼ `newtok()`

```

2516 static char*
2517 newtok()
2518 {
2519     tokidx = 0;
2520     if (!tokbuf) {
2521         toksiz = 60;
2522         tokbuf = ALLOC_N(char, 60);
2523     }
2524     if (toksiz > 4096) {
2525         toksiz = 60;
2526         REALLOC_N(tokbuf, char, 60);
2527     }
2528     return tokbuf;
2529 }
```

`(parse.y)`

The initializing interface of the whole buffer does not exist, it's possible that the buffer is not initialized. Therefore, the first `if` checks it and initializes it. `ALLOC_N()` is the macro ruby defines and is almost the same as `calloc`.

The initial value of the allocating length is 60, and if it becomes too big ( $> 4096$ ), it would be returned back to small. Since a token becoming this long is unlikely, this size is realistic.

Next, let's look at the `tokadd()` to add a character to token buffer.

### ▼ `tokadd()`

```
2531 static void
2532 tokadd(c)
2533     char c;
2534 {
2535     tokenbuf[tokidx++] = c;
2536     if (tokidx >= toksiz) {
2537         toksiz *= 2;
2538         REALLOC_N(tokenbuf, char, toksiz);
2539     }
2540 }
```

(`parse.y`)

At the first line, a character is added. Then, it checks the token length and if it seems about to exceed the buffer end, it performs `REALLOC_N()`. `REALLOC_N()` is a `realloc()` which has the same way of specifying arguments as `calloc()`.

The rest interfaces are summarized below.

### ▼ `tokfix()` `tok()` `toklen()` `toklast()`

```
2511 #define tokfix() (tokenbuf[tokidx]='\0')
2512 #define tok() tokenbuf
2513 #define toklen() tokidx
2514 #define toklast() (tokidx>0?tokenbuf[tokidx-1]:0)
```

There's probably no question.

## ■ `yylex()`

`yylex()` is very long. Currently, there are more than 1000 lines. The most of them is occupied by a huge `switch` statement, it branches based on each character. First, I'll show the whole structure that some parts of it are left out.

### ▼ `yylex` outline

```
3106 static int
3107 yylex()
3108 {
3109     static ID last_id = 0;
3110     register int c;
3111     int space_seen = 0;
3112     int cmd_state;
3113
3114     if (lex_strterm) {
3115         /* ... string scan ... */
3116         return token;
3117     }
3118     cmd_state = command_start;
3119     command_start = Qfalse;
3120     retry:
3121     switch (c = nextc()) {
3122         case '\0':                      /* NUL */
3123         case '\004':                   /* ^D */
3124         case '\032':                   /* ^Z */
3125         case -1:                      /* end of script. */
3126             return 0;
3127
3128         /* white spaces */
```

```

3144     case ' ': case '\t': case '\f': case '\r':
3145     case '\13': /* '\v' */
3146         space_seen++;
3147         goto retry;
3148
3149     case '#':          /* it's a comment */
3150         while ((c = nextc()) != '\n') {
3151             if (c == -1)
3152                 return 0;
3153         }
3154         /* fall through */
3155     case '\n':
3156         /* ... omission ... */
3157
3158     case xxxx:
3159     :
3160         break;
3161     :
3162     /* branches a lot for each character */
3163     :
3164     :
3165     :
3166     default:
3167         if (!is_identchar(c) || ISDIGIT(c)) {
3168             rb_compile_error("Invalid char '\\%03o' in expr");
3169             goto retry;
3170         }
3171
3172         newtok();
3173         break;
3174     }
3175
3176     /* ... deal with ordinary identifiers ... */
3177 }

```

(parse.y)

As for the return value of `yylex()`, zero means that the input has finished, non-zero means a symbol.

Be careful that a extremely concise variable named “c” is used all

over this function. `space_seen++` when reading a space will become helpful later.

All it has to do as the rest is to keep branching for each character and processing it, but since continuous monotonic procedure is lasting, it is boring for readers. Therefore, we'll narrow them down to a few points. In this book not all characters will be explained, but it is easy if you will amplify the same pattern.

! ! !

Let's start with what is simple first.

▼ `yylex - '!'`

```
3205      case '!':
3206          lex_state = EXPR_BEG;
3207          if ((c = nextc()) == '=') {
3208              return tNEQ;
3209          }
3210          if (c == '~') {
3211              return tNMATCH;
3212          }
3213          pushback(c);
3214          return '!';
```

`(parse.y)`

I wrote out the meaning of the code, so I'd like you to read them by comparing each other.

```
case '!':
    move to EXPR_BEG
    if (the next character is '=' then) {
```

```

        token is 「!= (tNEQ)」
    }
    if (the next character is '~' then) {
        token is 「!~ (tNMATCH)」
    }
    if it is neither, push the read character back
    token is '!'

```

This case clause is short, but describes the important rule of the scanner. It is “the longest match rule”. The two characters “!=” can be interpreted in two ways: “!” and “=” or “!=”, but in this case “!=” must be selected. The longest match is essential for scanners of programming languages.

And, `lex_state` is the variable represents the state of the scanner. This will be discussed too much in the next chapter “Finite-State Scanner”, you can ignore it for now. `EXPR_BEG` indicates “it is clearly at the beginning”. This is because whichever it is ! of not or it is != or it is !~, its next symbol is the beginning of an expression.

'<'

Next, we'll try to look at '<' as an example of using `yyval` (the value of a symbol).

▼ `yylex->;`

```

3296      case '>':
3297          switch (lex_state) {
3298              case EXPR_FNAME: case EXPR_DOT:
3299                  lex_state = EXPR_ARG; break;
3300              default:

```

```
3301             lex_state = EXPR_BEG; break;
3302         }
3303         if ((c = nextc()) == '=') {
3304             return tGEQ;
3305         }
3306         if (c == '>') {
3307             if ((c = nextc()) == '=') {
3308                 yylval.id = tRSHFT;
3309                 lex_state = EXPR_BEG;
3310                 return tOP_ASgn;
3311             }
3312             pushback(c);
3313             return tRSHFT;
3314         }
3315         pushback(c);
3316         return '>';
```

(parse.y)

The places except for `yylval` can be ignored. Concentrating only one point when reading a program is essential.

At this point, for the symbol `tOP_ASgn` of `>=`, it set its value `tRSHIFT`. Since the used union member is `id`, its type is `ID`. `tOP_ASgn` is the symbol of self assignment, it represents all of the things like `+=` and `-=` and `*=`. In order to distinguish them later, it passes the type of the self assignment as a value.

The reason why the self assignments are bundled is, it makes the rule shorter. Bundling things that can be bundled at the scanner as much as possible makes the rule more concise. Then, why are the binary arithmetic operators not bundled? It is because they differs in their precedences.

':'

If scanning is completely independent from parsing, this talk would be simple. But in reality, it is not that simple. The Ruby grammar is particularly complex, it has a somewhat different meaning when there's a space in front of it, the way to split tokens is changed depending on the situation around. The code of `':'` shown below is an example that a space changes the behavior.

▼ `yylex-':'`

```
3761     case ':':
3762         c = nextc();
3763         if (c == ':') {
3764             if (lex_state == EXPR_BEG || lex_state == EXPR_
3765                 (IS_ARG() && space_seen)) {
3766                 lex_state = EXPR_BEG;
3767                 return tCOLON3;
3768             }
3769             lex_state = EXPR_DOT;
3770             return tCOLON2;
3771         }
3772         pushback(c);
3773         if (lex_state == EXPR_END ||
3774             lex_state == EXPR_ENDARG ||
3775             ISSPACE(c)) {
3776             lex_state = EXPR_BEG;
3777             return ':';
3778         }
3779         lex_state = EXPR_FNAME;
3780         return tSYMBEG;

(parse.y)
```

Again, ignoring things relating to `lex_state`, I'd like you focus on around `space_seen`.

`space_seen` is the variable that becomes true when there's a space before a token. If it is met, meaning there's a space in front of `'::'`, it becomes `tCOLON3`, if there's not, it seems to become `tCOLON2`. This is as I explained at `primary` in the previous section.

# Identifier

Until now, since there were only symbols, it was just a character or 2 characters. This time, we'll look at a little longer things. It is the scanning pattern of identifiers.

First, the outline of `yylex` was as follows:

```
yylex(...)  
{  
    switch (c = nextc()) {  
        case xxxx:  
            ....  
        case xxxx:  
            ....  
        default:  
    }  
    the scanning code of identifiers  
}
```

The next code is an extract from the end of the huge `switch`. This is relatively long, so I'll show it with comments.

## ▼ yylex – identifiers

```
4081     case '@':                  /* an instance variable or a
4082         c = nextc();
```

```
4083     newtok();
4084     tokadd('@');
4085     if (c == '@') {             /* @@, meaning a class varia
4086         tokadd('@');
4087         c = nextc();
4088     }
4089     if (ISDIGIT(c)) {          /* @1 and such */
4090         if (tokidx == 1) {
4091             rb_compile_error("`@%c' is not a valid instance variable n
4092         }
4093         else {
4094             rb_compile_error("`@@%c' is not a valid class variable nam
4095         }
4096     }
4097     if (!is_identchar(c)) { /* a strange character appea
4098         pushback(c);
4099         return '@';
4100     }
4101     break;
4102
4103     default:
4104     if (!is_identchar(c) || ISDIGIT(c)) {
4105         rb_compile_error("Invalid char `\\%03o' in expre
4106         goto retry;
4107     }
4108
4109     newtok();
4110     break;
4111 }
4112
4113     while (is_identchar(c)) { /* between characters that c
4114         tokadd(c);
4115         if (ismbchar(c)) { /* if it is the head byte of
4116             int i, len = mbclen(c)-1;
4117
4118             for (i = 0; i < len; i++) {
4119                 c = nextc();
4120                 tokadd(c);
4121             }
4122         }
4123         c = nextc();
4124     }
4125     if ((c == '!' || c == '?') &&
```

```
4126     is_identchar(tok()[0]) &&
4127     !peek('=')) { /* the end character of name! or
4128         tokadd(c);
4129     }
4130     pushback(c);
4131     tokfix();
```

(parse.y)

Finally, I'd like you focus on the condition at the place where adding ! or ?. This part is to interpret in the next way.

obj.m=1	# obj.m = 1	(not obj.m=)
obj.m!=1	# obj.m != 1	(not obj.m!)

((errata: this code is not relating to that condition))

This is “not” longest-match. The “longest-match” is a principle but not a constraint. Sometimes, you can refuse it.

## The reserved words

After scanning the identifiers, there are about 100 lines of the code further to determine the actual symbols. In the previous code, instance variables, class variables and local variables, they are scanned all at once, but they are categorized here.

This is OK but, inside it there's a little strange part. It is the part to filter the reserved words. Since the reserved words are not different from local variables in its character type, scanning in a bundle and

categorizing later is more efficient.

Then, assume there's `str` that is a `char*` string, how can we determine whether it is a reserved word? First, of course, there's a way of comparing a lot by `if` statements and `strcmp()`. However, this is completely not smart. It is not flexible. Its speed will also linearly increase. Usually, only the data would be separated to a list or a hash in order to keep the code short.

```
/* convert the code to data */
struct entry {char *name; int symbol;};
struct entry *table[] = {
    {"if",      kIF},
    {"unless",  kUNLESS},
    {"while",   kWHILE},
    /* ..... omission ..... */
};

{
    ...
    return lookup_symbol(table, tok());
}
```

Then, how ruby is doing is that, it uses a hash table. Furthermore, it is a perfect hash. As I said when talking about `st_table`, if you knew the set of the possible keys beforehand, sometimes you could create a hash function that never conflicts. As for the reserved words, “the set of the possible keys is known beforehand”, so it is likely that we can create a perfect hash function.

But, “being able to create” and actually creating are different. Creating manually is too much cumbersome. Since the reserved words can increase or decrease, this kind of process must be

automated.

Therefore, gperf comes in. gperf is one of GNU products, it generates a perfect function from a set of values. In order to know the usage of gperf itself in detail, I recommend to do `man gperf`. Here, I'll only describe how to use the generated result.

In ruby the input file for gperf is `keywords` and the output is `lex.c`. `parse.y` directly `#include` it. Basically, doing `#include` C files is not good, but performing non-essential file separation for just one function is worse. Particularly, in ruby, there's the possibility that `extern+` functions are used by extension libraries without being noticed, thus the function that does not want to keep its compatibility should be `static`.

Then, in the `lex.c`, a function named `rb_reserved_word()` is defined. By calling it with the `char*` of a reserved word as key, you can look up. The return value is `NULL` if not found, `struct kwtable*` if found (in other words, if the argument is a reserved word). The definition of `struct kwtable` is as follows:

### ▼ `kwtable`

```
1 struct kwtable {char *name; int id[2]; enum lex_state state;  
(keywords)
```

`name` is the name of the reserved word, `id[0]` is its symbol, `id[1]` is its symbol as a modification (`KIF_MOD` and such). `lex_state` is “the

lex\_state should be moved to after reading this reserved word". lex\_state will be explained in the next chapter.

This is the place where actually looking up.

### ▼ yylex() — identifier — call rb\_reserved\_word()

```
4173             struct kwtable *kw;
4174
4175             /* See if it is a reserved word. */
4176             kw = rb_reserved_word(tok(), toklen());
4177             if (kw) {
4178
4179             (parse.y)
```

## ■ Strings

The double quote ("") part of yylex() is this.

### ▼ yylex - ""

```
3318         case '':
3319             lex_strterm = NEW_STRTERM(str_dquote, "", 0);
3320             return tSTRING_BEG;
3321
3322             (parse.y)
```

Surprisingly it finishes after scanning only the first character. Then, this time, when taking a look at the rule, tSTRING\_BEG is found in the following part:

### ▼ rules for strings

```

string1           : tSTRING_BEG string_contents tSTRING_END

string_contents  :
                  | string_contents string_content

string_content   : tSTRING_CONTENT
                  | tSTRING_DVAR string_dvar
                  | tSTRING_DBEG term_push compstmt '}'

string_dvar      : tGVAR
                  | tIVAR
                  | tCVAR
                  | backref

term_push        :

```

These rules are the part introduced to deal with embedded expressions inside of strings. `tSTRING_CONTENT` is literal part, `tSTRING_DBEG` is "#{}". `tSTRING_DVAR` represents “# that in front of a variable”. For example,

“.....#\$gvar.....”

this kind of syntax. I have not explained but when the embedded expression is only a variable, { and } can be left out. But this is often not recommended. D of DVAR, DBEG seems the abbreviation of dynamic.

And, `backref` represents the special variables relating to regular expressions, such as `$1` `$2` or `$&` `$'`.

`term_push` is “a rule defined for its action”.

Now, we'll go back to `yylex()` here. If it simply returns the parser, since its context is the “interior” of a string, it would be a problem if a variable and `if` and others are suddenly scanned in the next `yylex()`. What plays an important role there is ...

```
case '":  
    lex_strterm = NEW_STRTERM(str_dquote, "", 0);  
    return tSTRING_BEG;
```

... `lex_strterm`. Let's go back to the beginning of `yylex()`.

## ▼ the beginning of `yylex()`

```
3106 static int  
3107 yylex()  
3108 {  
3109     static ID last_id = 0;  
3110     register int c;  
3111     int space_seen = 0;  
3112     int cmd_state;  
3113  
3114     if (lex_strterm) {  
3115         /* scanning string */  
3116         return token;  
3117     }  
3118     cmd_state = command_start;  
3119     command_start = Qfalse;  
3120     retry:  
3121         switch (c = nextc()) {  
  
(parse.y)
```

If `lex_strterm` exists, it enters the string mode without asking. It means, conversely speaking, if there's `lex_strterm`, it is while scanning string, and when parsing the embedded expressions

inside strings, you have to set `lex_strterm` to 0. And, when the embedded expression ends, you have to set it back. This is done in the following part:

## ▼ `string_content`

```
1916  string_content  : ....
1917          | tSTRING_DBEG term_push
1918          {
1919              $<num>1 = lex_strnest;
1920              $<node>$ = lex_strterm;
1921              lex_strterm = 0;
1922              lex_state = EXPR_BEG;
1923          }
1924          compstmt '}'
1925          {
1926              lex_strnest = $<num>1;
1927              quoted_term = $2;
1928              lex_strterm = $<node>3;
1929              if (($$ = $4) && nd_type($$) == NODE)
1930                  $$ = $$->nd_next;
1931                  rb_gc_force_recycle((VALUE)$4);
1932              }
1933              $$ = NEW_EVSTR($$);
1934          }
```

(`parse.y`)

In the embedded action, `lex_stream` is saved as the value of `tSTRING_DBEG` (virtually, this is a stack push), it recovers in the ordinary action (pop). This is a fairly smart way.

But why is it doing this tedious thing? Can't it be done by, after scanning normally, calling `yyparse()` recursively at the point when it finds #{}? There's actually a problem. `yyparse()` can't be called

recursively. This is the well known limit of yacc. Since the `yyval` that is used to receive or pass a value is a global variable, careless recursive calls can destroy the value. With `bison` (yacc of GNU), recursive calls are possible by using `%pure_parser` directive, but the current ruby decided not to assume `bison`. In reality, `byacc` (Berkely yacc) is often used in BSD-derived OS and Windows and such, if `bison` is assumed, it causes a little cumbersome.

## lex\_strterm

As we've seen, when you consider `lex_stream` as a boolean value, it represents whether or not the scanner is in the string mode. But its contents also has a meaning. First, let's look at its type.

### ▼ lex\_strterm

```
72 static NODE *lex_strterm;  
(parse.y)
```

This definition shows its type is `NODE*`. This is the type used for syntax tree and will be discussed in detail in Chapter 12: Syntax tree construction. For the time being, it is a structure which has three elements, since it is `VALUE` you don't have to `free()` it, you should remember only these two points.

### ▼ NEW\_STRTERM()

```
2865 #define NEW_STRTERM(func, term, paren) \  
2866         rb_node_newnode(NODE_STRTERM, (func), (term), (paren)
```

This is a macro to create a node to be stored in `lex_stream`. First, `term` is the terminal character of the string. For example, if it is a " string, it is ", and if it is a ' string, it is '.

`paren` is used to store the corresponding parenthesis when it is a % string. For example,

```
%Q(.....)
```

in this case, `paren` stores '('. And, `term` stores the closing parenthesis ')'. If it is not a % string, `paren` is 0.

At last, `func`, this indicates the type of a string. The available types are decided as follows:

### ▼ func

```

2775 #define STR_FUNC_ESCAPE 0x01 /* backslash notations such as
2776 #define STR_FUNC_EXPAND 0x02 /* embedded expressions are in
2777 #define STR_FUNC_REGEXP 0x04 /* it is a regular expression
2778 #define STR_FUNC_QWORDS 0x08 /* %w(....) or %W(....) */
2779 #define STR_FUNC_INDENT 0x20 /* <<-EOS(the finishing symbol
2780
2781 enum string_type {
2782     str_squote = (0),
2783     str_dquote = (STR_FUNC_EXPAND),
2784     str_xquote = (STR_FUNC_ESCAPE|STR_FUNC_EXPAND),
2785     str_regexp = (STR_FUNC_REGEXP|STR_FUNC_ESCAPE|STR_FUNC_E
2786     str_sword  = (STR_FUNC_QWORDS),
2787     str_dword  = (STR_FUNC_QWORDS|STR_FUNC_EXPAND),
2788 };

```

Each meaning of enum string\_type is as follows:

str\_squote ' string / %q

str\_dquote " string / %Q

str\_xquote command string (not be explained in this book)

str\_regex regular expression

str\_sword %w

str\_dword %W

## String scan function

The rest is reading `yylex()` in the string mode, in other words, the if at the beginning.

### ▼ `yylex - string`

```

3114      if (lex_strterm) {
3115          int token;
3116          if (nd_type(lex_strterm) == NODE_HEREDOC) {
3117              token = here_document(lex_strterm);
3118              if (token == tSTRING_END) {
3119                  lex_strterm = 0;
3120                  lex_state = EXPR_END;
3121              }
3122          }
3123      else {
3124          token = parse_string(lex_strterm);
3125          if (token == tSTRING_END || token == tREGEXP_END)
3126              rb_gc_force_recycle((VALUE)lex_strterm);
3127          lex_strterm = 0;
3128          lex_state = EXPR_END;
3129      }

```

```
3130 }  
3131     return token;  
3132 }  
  
(parse.y)
```

It is divided into the two major groups: here document and others. But this time, we won't read `parse_string()`. As I previously described, there are a lot of conditions, it is tremendously being a spaghetti code. If I tried to explain it, odds are high that readers would complain that “it is as the code is written!”. Furthermore, although it requires a lot of efforts, it is not interesting.

But, not explaining at all is also not a good thing to do, The modified version that functions are separately defined for each target to be scanned is contained in the attached CD-ROM (`doc/parse_string.html`). I'd like readers who are interested in to try to look over it.

## Here Document

In comparison to the ordinary strings, here documents are fairly interesting. That may be because, unlike the other elements, it deal with a line at a time. Moreover, it is terrific that the starting symbol can exist in the middle of a program. First, I'll show the code of `yylex()` to scan the starting symbol of a here document.

▼ `yylex-&lt;`

```
3260     case '<':
```

```
3261     c = nextc();
3262     if (c == '<' &&
3263         lex_state != EXPR_END &&
3264         lex_state != EXPR_DOT &&
3265         lex_state != EXPR_ENDARG &&
3266         lex_state != EXPR_CLASS &&
3267         (!IS_ARG() || space_seen)) {
3268         int token = heredoc_identifier();
3269         if (token) return token;
```

(parse.y)

As usual, we'll ignore the herd of `lex_state`. Then, we can see that it reads only “<<” here and the rest is scanned at `heredoc_identifier()`. Therefore, here is `heredoc_identifier()`.

## ▼ `heredoc_identifier()`

```
2926 static int
2927 heredoc_identifier()
2928 {
2929     /* ... omission ... reading the starting symbol */
2930     tokfix();
2931     len = lex_p - lex_pbeg; /*(A)*/
2932     lex_p = lex_pend; /*(B)*/
2933     lex_strterm = rb_node_newnode(NODE_HEREDOC,
2934                                     rb_str_new(tok(), toklen()), /* nd_
2935                                     len, /* nd_
2936                                     /*(C)*/
2937                                     lex_lastline); /* nd_
2938
2939     return term == ``' ? tXSTRING_BEG : tSTRING_BEG;
2940 }
```

(parse.y)

The part which reads the starting symbol (<<EOS) is not important, so it is totally left out. Until now, the input buffer probably has

become as depicted as Figure 10. Let's recall that the input buffer reads a line at a time.

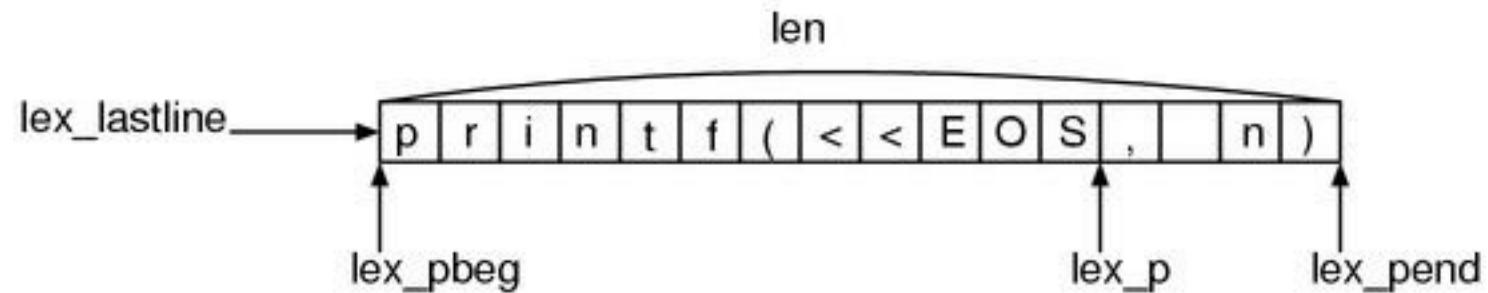


Figure 10: scanning "printf(<<EOS,n)"

What `heredoc_identifier()` is doing is as follows:

- (A) `len` is the number of read bytes in the current line.
- (B) and, suddenly move `lex_p` to the end of the line.

It means that in the read line, the part after the starting symbol is read but not parsed. When is that rest part parsed? For this mystery, a hint is that at (C) the `lex_lastline` (the currently read line) and `len` (the length that has already read) are saved.

Then, the dynamic call graph before and after `heredoc_identifier` is simply shown below:

```
yyparse
    yylex(case '<')
        heredoc_identifier(lex_strterm = ....)
    yylex(the beginning if)
        here_document
```

And, this `here_document()` is doing the scan of the body of the here document. Omitting invalid cases and adding some comments, `heredoc_identifier()` is shown below. Notice that `lex_strterm`

remains unchanged after it was set at `heredoc_identifier()`.

## ▼ `here_document()`(simplified)

```
here_document(NODE *here)
{
    VALUE line;                      /* the line currently being s
    VALUE str = rb_str_new("", 0);    /* a string to store the resu
    /* ... handling invalid conditions, omitted ... */

    if (embeded expressions not in effect) {
        do {
            line = lex_lastline;      /*(A)*/
            rb_str_cat(str, RSTRING(line)->ptr, RSTRING(line)->len
            lex_p = lex_pend;        /*(B)*/
            if (nextc() == -1) {     /*(C)*/
                goto error;
            }
        } while (the currently read line is not equal to the finis
    }
    else {
        /* the embeded expressions are available ... omitted */
    }
    heredoc_restore(lex_strterm);
    lex_strterm = NEW_STRTERM(-1, 0, 0);
    yylval.node = NEW_STR(str);
    return tSTRING_CONTENT;
}
```

`rb_str_cat()` is the function to connect a `char*` at the end of a Ruby string. It means that the currently being read line `lex_lastline` is connected to `str` at (A). After it is connected, there's no use of the current line. At (B), suddenly moving `lex_p` to the end of line. And (C) is a problem, in this place, it looks like doing the check whether it is finished, but actually the next “line” is read. I'd like you to

recall that `nextc()` automatically reads the next line when the current line has finished to be read. So, since the current line is forcibly finished at (B), `lex_p` moves to the next line at (C).

And finally, leaving the `do ~ while` loop, it is `heredoc_restore()`.

### ▼ `heredoc_restore()`

```
2990 static void
2991 heredoc_restore(here)
2992     NODE *here;
2993 {
2994     VALUE line = here->nd_orig;
2995     lex_lastline = line;
2996     lex_pbeg = RSTRING(line)->ptr;
2997     lex_pend = lex_pbeg + RSTRING(line)->len;
2998     lex_p = lex_pbeg + here->nd_nth;
2999     heredoc_end = ruby_sourceline;
3000     ruby_sourceline = nd_line(here);
3001     rb_gc_force_recycle(here->nd_lit);
3002     rb_gc_force_recycle((VALUE)here);
3003 }
```

(`parse.y`)

`here->nd_orig` holds the line which contains the starting symbol.  
`here->nd_nth` holds the length already read in the line contains the starting symbol.

It means it can continue to scan from the just after the starting symbol as if there was nothing happened. (Figure 11)

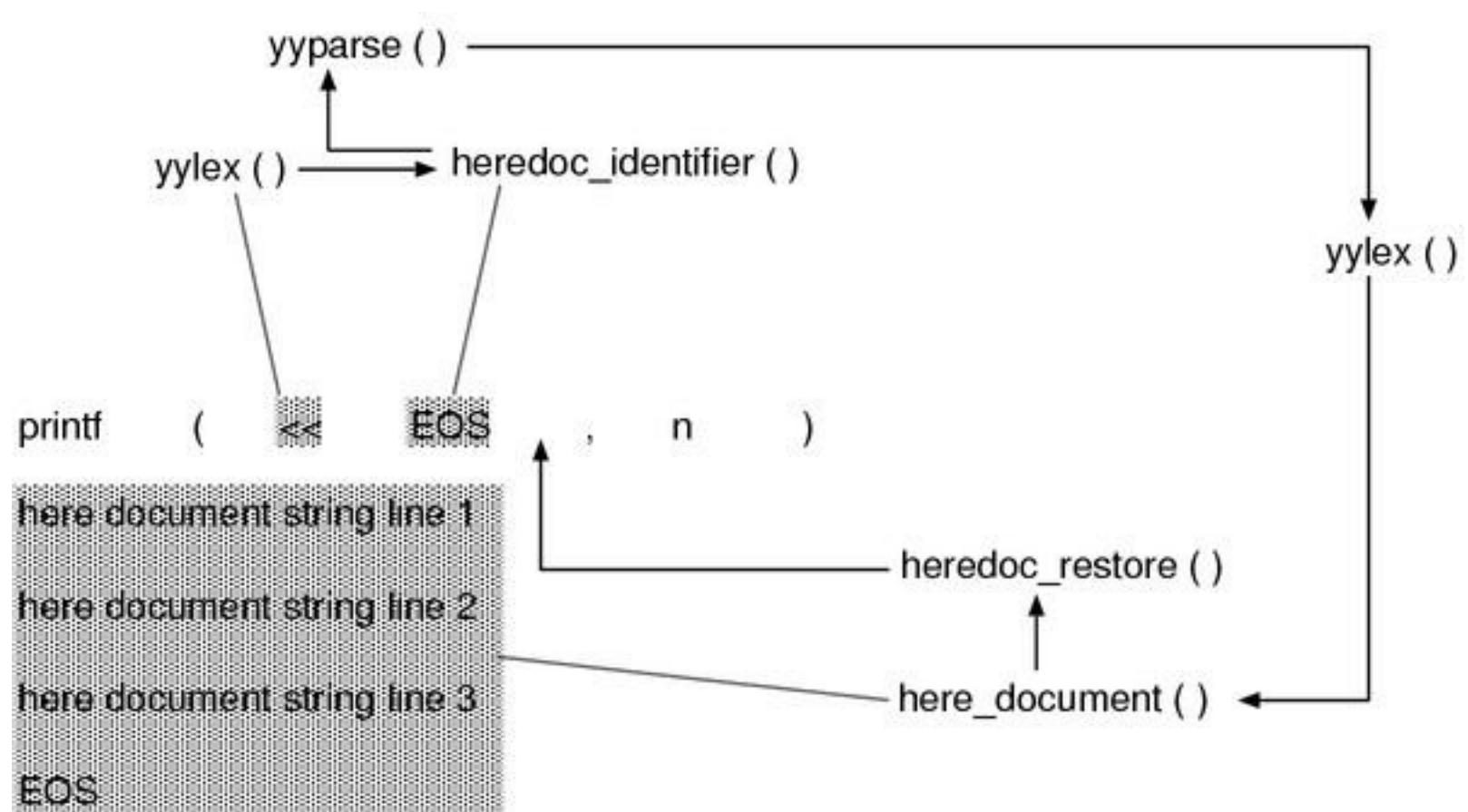


Figure 11: The picture of assignation of scanning Here Document

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

Translated by Peter Zotov

*I'm very grateful to my employer [Evil Martians](#) , who sponsored the work, and [Nikolay Konovalenko](#) , who put more effort in this translation than I could ever wish for. Without them, I would be still figuring out what `COND_LEXPOP()` actually does.*

# Chapter 11 Finite-state scanner

## Outline

---

In theory, the scanner and the parser are completely independent of each other – the scanner is supposed to recognize tokens, while the parser is supposed to process the resulting series of tokens. It would be nice if things were that simple, but in reality it rarely is. Depending on the context of the program it is often necessary to alter the way tokens are recognized or their symbols. In this chapter we will take a look at the way the scanner and the parser cooperate.

## Practical examples

In most programming languages, spaces don't have any specific meaning unless they are used to separate words. However, Ruby is not an ordinary language and meanings can change significantly depending on the presence of spaces. Here is an example

```
a[i] = 1      # a[i] = (1)
```

```
a [i]      # a([i])
```

The former is an example of assigning an index. The latter is an example of omitting the method call parentheses and passing a member of an array to a parameter.

Here is another example.

```
a + 1      # (a) + (1)
a +1       # a(+1)
```

This seems to be really disliked by some.

However, the above examples might give one the impression that only omitting the method call parentheses can be a source of trouble. Let's look at a different example.

```
`cvs diff parse.y`      # command call string
obj.`("cvs diff parse.y")` # normal method call
```

Here, the former is a method call using a literal. In contrast, the latter is a normal method call (with “`” being the method name). Depending on the context, they could be handled quite differently.

Below is another example where the functioning changes dramatically

```
print(<<EOS)  # here-document
.....
EOS

list = []
```

```
list << nil      # list.push(nil)
```

The former is a method call using a here-document. The latter is a method call using an operator.

As demonstrated, Ruby's grammar contains many parts which are difficult to implement in practice. I couldn't realistically give a thorough description of all in just one chapter, so in this one I will look at the basic principles and those parts which present the most difficulty.

## ■ lex\_state

There is a variable called “lex\_state”. “lex”, obviously, stands for “lexer”. Thus, it is a variable which shows the scanner's state.

What states are there? Let's look at the definitions.

### ▼ enum lex\_state

```
61 static enum lex_state {
62     EXPR_BEG,          /* ignore newline, +/- is a sign. */
63     EXPR_END,          /* newline significant, +/- is a operator */
64     EXPR_ARG,          /* newline significant, +/- is a operator */
65     EXPR_CMDARG,       /* newline significant, +/- is a operator */
66     EXPR_ENDARG,       /* newline significant, +/- is a operator */
67     EXPR_MID,          /* newline significant, +/- is a operator */
68     EXPR_FNAME,         /* ignore newline, no reserved words. */
69     EXPR_DOT,          /* right after `.' or `::', no reserved w */
70     EXPR_CLASS,         /* immediate after `class', no here docum */
71 } lex_state;
```

(parse.y)

The EXPR prefix stands for “expression”. EXPR\_BEG is “Beginning of expression” and EXPR\_DOT is “inside the expression, after the dot”.

To elaborate, EXPR\_BEG denotes “Located at the head of the expression”. EXPR\_END denotes “Located at the end of the expression”. EXPR\_ARG denotes “Before the method parameter”. EXPR\_FNAME denotes “Before the method name (such as def)”. The ones not covered here will be analyzed in detail below.

Incidentally, I am led to believe that `lex_state` actually denotes “after parentheses”, “head of statement”, so it shows the state of the parser rather than the scanner. However, it’s still conventionally referred to as the scanner’s state and here’s why.

The meaning of “state” here is actually subtly different from how it’s usually understood. The “state” of `lex_state` is “a state under which the scanner does x”. For example an accurate description of EXPR\_BEG would be “A state under which the scanner, if run, will react as if this is at the head of the expression”

Technically, this “state” can be described as the state of the scanner if we look at the scanner as a state machine. However, delving there would be veering off topic and too tedious. I would refer any interested readers to any textbook on data structures.

## ■ Understanding the finite-state scanner

The trick to reading a finite-state scanner is to not try to grasp

everything at once. Someone writing a parser would prefer not to use a finite-state scanner. That is to say, they would prefer not to make it the main part of the process. Scanner state management often ends up being an extra part attached to the main part. In other words, there is no such thing as a clean and concise diagram for state transitions.

What one should do is think toward specific goals: “This part is needed to solve this task” “This code is for overcoming this problem”. Basically, put out code in accordance with the task at hand. If you start thinking about the mutual relationship between tasks, you’ll invariably end up stuck. Like I said, there is simply no such thing.

However, there still needs to be an overreaching objective. When reading a finite-state scanner, that objective would undoubtedly be to understand every state. For example, what kind of state is EXPR\_BEG? It is a state where the parser is at the head of the expression.

## **The static approach**

So, how can we understand what a state does? There are three basic approaches

- Look at the name of the state

The simplest and most obvious approach. For example, the name EXPR\_BEG obviously refers to the head (beginning) of something.

- Observe what changes under this state

Look at the way token recognition changes under the state, then test it in comparison to previous examples.

- Look at the state from which it transitions

Look at which state it transitions from and which token causes it. For example, if '\n' is always followed by a transition to a HEAD state, it must denote the head of the line.

Let us take EXPR\_BEG as an example. In Ruby, all state transitions are expressed as assignments to `lex_state`, so first we need to grep EXPR\_BEG assignments to find them. Then we need to export their location, for example, such as '#' and '\*' and '!' of `yylex()`. Then we need to recall the state prior to the transition and consider which case suits best (see image 1)

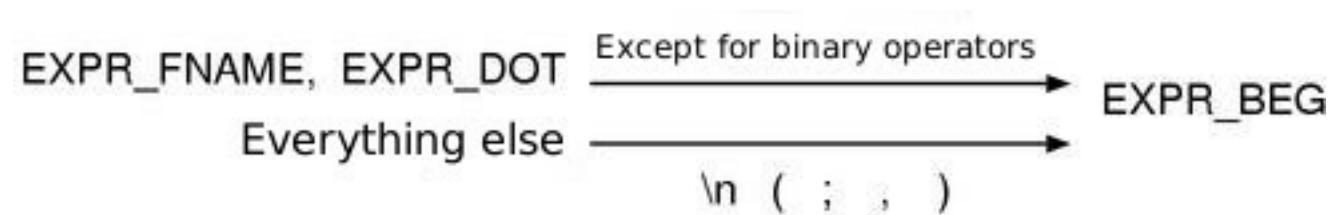


Figure 1: Transition to EXPR\_BEG

((errata:

1. Actually when the state is EXPR\_DOT, the state after reading a tIDENTIFIER would be either ARG or CMDARG. However, because the author wanted to roughly group them as FNAME/DOT and the others here, these two are shown together. Therefore, to be precise,

`EXPR_FNAME` and `EXPR_DOT` should have also been separated.

2. ‘)’ does not cause the transition from “everything else” to `EXPR_BEG`. ))

This does indeed look like the head of statement. Especially the ‘\n’ and the ‘;’ The open parentheses and the comma also suggest that it’s the head not just of the statement, but of the expression as well.

## The dynamic approach

There are other easy methods to observe the functioning. For example, you can use a debugger to “hook” the `yylex()` and look at the `lex_state`

Another way is to rewrite the source code to output state transitions. In the case of `lex_state` we only have a few patterns for assignment and comparison, so the solution would be to grasp them as text patterns and rewrite the code to output state transitions. The CD that comes with this book contains the `rubylex-analyser` tool. When necessary, I will refer to it in this text.

The overall process looks like this: use a debugger or the aforementioned tool to observe the functioning of the program. Then look at the source code to confirm the acquired data and use it.

## ■ Description of states

Here I will give simple descriptions of `lex_state` states.

- `EXPR_BEG`

Head of expression. Comes immediately after `\n` ( `{` `[` `!` `?` `:` , or the operator `op=` The most general state.

- `EXPR_MID`

Comes immediately after the reserved words `return` `break` `next` `rescue`. Invalidates binary operators such as `*` or `&` Generally similar in function to `EXPR_BEG`

- `EXPR_ARG`

Comes immediately after elements which are likely to be the method name in a method call. Also comes immediately after `'` `[` `''` Except for cases where `EXPR_CMDARG` is used.

- `EXPR_CMDARG`

Comes before the first parameter of a normal method call. For more information, see the section “The `do` conflict”

- `EXPR_END`

Used when there is a possibility that the statement is terminal. For example, after a literal or a closing parenthesis. Except for cases when `EXPR_ENDARG` is used

- EXPR\_ENDARG

Special iteration of EXPR\_END Comes immediately after the closing parenthesis corresponding to tLPAREN\_ARG Refer to the section “First parameter enclosed in parentheses”

- EXPR\_FNAME

Comes before the method name, usually after def, alias, undef or the symbol `:'. A single ``'' can be a name.

- EXPR\_DOT

Comes after the dot in a method call. Handled similarly to EXPR\_FNAME Various reserved words are treated as simple identifiers. A single ``'' can be a name.

- EXPR\_CLASS

Comes after the reserved word class This is a very limited state.

The following states can be grouped together

- BEG MID
- END ENDARG
- ARG CMDARG
- FNAME DOT

They all express similar conditions. EXPR\_CLASS is a little different,

but only appears in a limited number of places, not warranting any special attention.

# Line-break handling

---

## ■ The problem

In Ruby, a statement does not necessarily require a terminator. In C or Java a statement must always end with a semicolon, but Ruby has no such requirement. Statements usually take up only one line, and thus end at the end of the line.

On the other hand, when a statement is clearly continued, this happens automatically. Some conditions for “This statement is clearly continued” are as follows:

- After a comma
- After an infix operator
- Parentheses or brackets are not balanced
- Immediately after the reserved word `if`

Etc.

## ■ Implementation

So, what do we need to implement this grammar? Simply having

the scanner ignore line-breaks is not sufficient. In a grammar like Ruby's, where statements are delimited by reserved words on both ends, conflicts don't happen as frequently as in C languages, but when I tried a simple experiment, I couldn't get it to work until I got rid of `return` next `break` and returned the method call parentheses wherever they were omitted. To retain those features we need some kind of terminal symbol for statements' ends. It doesn't matter whether it's `\n` or `';'` but it is necessary.

Two solutions exist – parser-based and scanner-based. For the former, you can just optionally put `\n` in every place that allows it. For the latter, have the `\n` passed to the parser only when it has some meaning (ignoring it otherwise).

Which solution to use is up to your preferences, but usually the scanner-based one is used. That way produces a more compact code. Moreover, if the rules are overloaded with meaningless symbols, it defeats the purpose of the parser-generator.

To sum up, in Ruby, line-breaks are best handled using the scanner. When a line needs to be continued, the `\n` will be ignored, and when it needs to be terminated, the `\n` is passed as a token. In the `yylex()` this is found here:

▼ `yylex() - '\n'`

```
3155      case '\n':
3156          switch (lex_state) {
3157              case EXPR_BEG:
3158              case EXPR_FNAME:
```

```
3159         case EXPR_DOT:
3160         case EXPR_CLASS:
3161             goto retry;
3162         default:
3163             break;
3164     }
3165     command_start = Qtrue;
3166     lex_state = EXPR_BEG;
3167     return '\n';
```

(parse.y)

With EXPR\_BEG, EXPR\_FNAME, EXPR\_DOT, EXPR\_CLASS it will be goto retry. That is to say, it's meaningless and shall be ignored. The label retry is found in front of the large switch in the yylex()

In all other instances, line-breaks are meaningful and shall be passed to the parser, after which lex\_state is restored to EXPR\_BEG. Basically, whenever a line-break is meaningful, it will be the end of expr

I recommend leaving command\_start alone for the time being. To reiterate, trying to grasp too many things at once will only end in needless confusion.

Let us now take a look at some examples using the rubylex-analyser tool.

```
% rubylex-analyser -e '
m(a,
  b, c) unless i
'
+EXPR_BEG
EXPR_BEG      C      "\nm"  tIDENTIFIER
```

EXPR\_CMDARG

EXPR_CMDARG		"(" ' (	EXPR_BEG
EXPR_BEG	C	"a" tIDENTIFIER	0:cond push
EXPR_CMDARG		"," ','	0:cmd push
EXPR_BEG	S	"\n b" tIDENTIFIER	EXPR_CMDARG
EXPR_ARG		"," ','	EXPR_BEG
EXPR_BEG	S	"c" tIDENTIFIER	EXPR_ARG
EXPR_ARG		")" ')' '	EXPR_END
EXPR_END	S	"unless" kUNLESS_MOD	0:cond lexpop
EXPR_BEG	S	"i" tIDENTIFIER	0:cmd lexpop
EXPR_ARG		"\n" '\n	EXPR_BEG
EXPR_BEG	C	"\n" '	EXPR_BEG

As you can see, there is a lot of output here, but we only need the left and middle columns. The left column displays the `lex_state` before it enters the `yylex()` while the middle column displays the tokens and their symbols.

The first token `m` and the second parameter `b` are preceded by a line-break but a `\n` is appended in front of them and it is not treated as a terminal symbol. That is because the `lex_state` is `EXPR_BEG`.

However, in the second to last line `\n` is used as a terminal symbol. That is because the state is `EXPR_ARG`

And that is how it should be used. Let us have another example.

```
% rubylex-analyser -e 'class
C < Object
end'
+EXPR_BEG
EXPR_BEG C "class" kCLASS EXPR_CLASS
```

EXPR_CLASS		"\nC"	tCONSTANT	EXPR_END
EXPR_END	S	"<"	'<'	EXPR_BEG
+EXPR_BEG				
EXPR_BEG	S	"Object"	tCONSTANT	EXPR_ARG
EXPR_ARG		"\n"	\n	EXPR_BEG
EXPR_BEG	C	"end"	kEND	EXPR_END
EXPR_END		"\n"	\n	EXPR_BEG

The reserved word `class` is followed by EXPR\_CLASS so the line-break is ignored. However, the superclass `Object` is followed by EXPR\_ARG, so the `\n` appears.

% rubylex-analyser -e 'obj.				
class'				
+EXPR_BEG				
EXPR_BEG	C	"obj"	tIDENTIFIER	EXPR_CMDARG
EXPR_CMDARG		"."	'.'	EXPR_DOT
EXPR_DOT		"\nclass"	tIDENTIFIER	EXPR_ARG
EXPR_ARG		"\n"	\n	EXPR_BEG

'.' is followed by EXPR\_DOT so the `\n` is ignored.

Note that `class` becomes `tIDENTIFIER` despite being a reserved word. This is discussed in the next section.

## Reserved words and identical method names

### ■ The problem

In Ruby, reserved words can be used as method names. However, in actuality it's not as simple as "it can be used" – there exist three possible contexts:

- Method definition (`def xxxx`)
- Call (`obj.xxxx`)
- Symbol literal (`:xxxx`)

All three are possible in Ruby. Below we will take a closer look at each.

First, the method definition. It is preceded by the reserved word `def` so it should work.

In case of the method call, omitting the receiver can be a source of difficulty. However, the scope of use here is even more limited, and omitting the receiver is actually forbidden. That is, when the method name is a reserved word, the receiver absolutely cannot be omitted. Perhaps it would be more accurate to say that it is forbidden in order to guarantee that parsing is always possible.

Finally, in case of the symbol, it is preceded by the terminal symbol `'::'` so it also should work. However, regardless of reserved words, the `'::'` here conflicts with the colon in `a?b:c` If this is avoided, there should be no further trouble.

For each of these cases, similarly to before, a scanner-based solution and a parser-based solution exist. For the former use `tIDENTIFIER` (for example) as the reserved word that comes after `def`

or . or : For the latter, make that into a rule. Ruby allows for both solutions to be used in each of the three cases.

## Method definition

The name part of the method definition. This is handled by the parser.

### ▼ Method definition rule

```
| kDEF fname
| f_arglist
| bodystmt
| kEND
| kDEF singleton dot_or_colon fname
| f_arglist
| bodystmt
| kEND
```

There exist only two rules for method definition – one for normal methods and one for singleton methods. For both, the name part is `fname` and it is defined as follows.

### ▼ fname

```
fname      : tIDENTIFIER
| tCONSTANT
| tFID
| op
| reswords
```

`reswords` is a reserved word and `op` is a binary operator. Both rules consist of simply all terminal symbols lined up, so I won't go into

detail here. Finally, for `tFID` the end contains symbols similarly to `gsub!` and `include?`

## Method call

Method calls with names identical to reserved words are handled by the scanner. The scan code for reserved words is shown below.

```
Scanning the identifier
result = (tIDENTIFIER or tCONSTANT)

if (lex_state != EXPR_DOT) {
    struct kwtable *kw;

    /* See if it is a reserved word. */
    kw = rb_reserved_word(tok(), toklen());
    Reserved word is processed
}
```

`EXPR_DOT` expresses what comes after the method call dot. Under `EXPR_DOT` reserved words are universally not processed. The symbol for reserved words after the dot becomes either `tIDENTIFIER` or `tCONSTANT`.

## Symbols

Reserved word symbols are handled by both the scanner and the parser. First, the rule.

▼ symbol

```

symbol          : tSYMBEG sym
sym            : fname
| tIVAR
| tGVAR
| tCVAR
fname          : tIDENTIFIER
| tCONSTANT
| tFID
| op
| reswords

```

Reserved words (reswords) are explicitly passed through the parser. This is only possible because the special terminal symbol `tSYMBEG` is present at the start. If the symbol were, for example, `:::` it would conflict with the conditional operator `(a?b:c)` and stall. Thus, the trick is to recognize `tSYMBEG` on the scanner level.

But how to cause that recognition? Let's look at the implementation of the scanner.

▼ `yylex-:::`

```

3761      case '::':
3762          c = nextc();
3763          if (c == '::') {
3764              if (lex_state == EXPR_BEG || lex_state == EXPR_
3765                  (IS_ARG() && space_seen)) {
3766                  lex_state = EXPR_BEG;
3767                  return tCOLON3;
3768              }
3769              lex_state = EXPR_DOT;
3770              return tCOLON2;
3771          }
3772          pushback(c);

```

```
3773     if (lex_state == EXPR_END ||  
            lex_state == EXPR_ENDARG ||  
            ISSPACE(c)) {  
            lex_state = EXPR_BEG;  
            return ':';  
        }  
        lex_state = EXPR_FNAME;  
        return tSYMBEG;
```

(parse.y)

This is a situation when the `if` in the first half has two consecutive `'::'`. In this situation, the `'::'` is scanned in accordance with the leftmost longest match basic rule.

For the next `if`, the `'::'` is the aforementioned conditional operator. Both `EXPR_END` and `EXPR_ENDARG` come at the end of the expression, so a parameter does not appear. That is to say, since there can't be a symbol, the `'::'` is a conditional operator. Similarly, if the next letter is a space (`ISSPACE(c)`), a symbol is unlikely so it is again a conditional operator.

When none of the above applies, it's all symbols. In that case, a transition to `EXPR_FNAME` occurs to prepare for all method names. There is no particular danger to parsing here, but if this is forgotten, the scanner will not pass values to reserved words and value calculation will be disrupted.

## Modifiers

# The problem

For example, for `if` if there exists a normal notation and one for postfix modification.

```
# Normal notation
if cond then
    expr
end
```

```
# Postfix
expr if cond
```

This could cause a conflict. The reason can be guessed – again, it's because method parentheses have been omitted previously.

Observe this example

```
call if cond then a else b end
```

Reading this expression up to the `if` gives us two possible interpretations.

```
call((if ....))
call() if ....
```

When unsure, I recommend simply using trial and error and seeing if a conflict occurs. Let us try to handle it with `yacc` after changing `kIF_MOD` to `kIF` in the grammar.

```
% yacc parse.y
parse.y contains 4 shift/reduce conflicts and 13 reduce/reduce con
```

As expected, conflicts are aplenty. If you are interested, you add the option `-v` to `yacc` and build a log. The nature of the conflicts should be shown there in great detail.

## Implementation

So, what is there to do? In Ruby, on the symbol level (that is, on the scanner level) the normal `if` is distinguished from the postfix `if` by them being `kIF` and `kIF_MOD` respectively. This also applies to all other postfix operators. In all, there are five – `kUNLESS_MOD` `kUNTIL_MOD` `kWHILE_MOD` `kRESCUE_MOD` and `kIF_MOD`. The distinction is made here:

### ▼ yylex-Reserved word

```
4173         struct kwtable *kw;
4174
4175         /* See if it is a reserved word. */
4176         kw = rb_reserved_word(tok(), toklen());
4177         if (kw) {
4178             enum lex_state state = lex_state;
4179             lex_state = kw->state;
4180             if (state == EXPR_FNAME) {
4181                 yylval.id = rb_intern(kw->name);
4182             }
4183             if (kw->id[0] == kD0) {
4184                 if (COND_P()) return kD0_COND;
4185                 if (CMDARG_P() && state != EXPR_CMDARG)
4186                     return kD0_BLOCK;
4187                 if (state == EXPR_ENDARG)
4188                     return kD0_BLOCK;
4189                 return kD0;
4190             }
4191             if (state == EXPR_BEG)  /*** Here ***/
4192                 return kw->id[0];
```

```
4193         else {
4194             if (kw->id[0] != kw->id[1])
4195                 lex_state = EXPR_BEG;
4196             return kw->id[1];
4197         }
4198     }
```

(parse.y)

This is located at the end of `yylex` after the identifiers are scanned. The part that handles modifiers is the last (innermost) `if~else`. Whether the return value is altered can be determined by whether or not the state is `EXPR_BEG`. This is where a modifier is identified. Basically, the variable `kw` is the key and if you look far above you will find that it is `struct kwtable`

I've already described in the previous chapter how `struct kwtable` is a structure defined in `keywords` and the hash function `rb_reserved_word()` is created by `gperf`. I'll show the structure here again.

## ▼ `keywords` – `struct kwtable`

```
1 struct kwtable {char *name; int id[2]; enum lex_state state;
(keywords)
```

I've already explained about `name` and `id[0]` – they are the reserved word name and its symbol. Here I will speak about the remaining members.

First, `id[1]` is a symbol to deal with modifiers. For example, in case

of `if` that would be `KIF_MOD`. When a reserved word does not have a modifier equivalent, `id[0]` and `id[1]` contain the same things.

Because `state` is `enum lex_state` it is the state to which a transition should occur after the reserved word is read. Below is a list created in the `kwstat.rb` tool which I made. The tool can be found on the CD.

```
% kwstat.rb ruby/keywords
---- EXPR_ARG
defined? super yield

---- EXPR_BEG
and case else ensure if module or unless wh
begin do elsif for in not then until until wh

---- EXPR_CLASS
class

---- EXPR_END
BEGIN __FILE__ end nil retry true
END __LINE__ false redo self

---- EXPR_FNAME
alias def undef

---- EXPR_MID
break next rescue return

---- modifiers
if rescue unless until while
```

## The do conflict

# The problem

There are two iterator forms – `do~end` and `{~}` Their difference is in priority – `{~}` has a much higher priority. A higher priority means that as part of the grammar a unit is “small” which means it can be put into a smaller rule. For example, it can be put not into `stmt` but `expr` or `primary`. In the past `{~}` iterators were in `primary` while `do~end` iterators were in `stmt`

By the way, there has been a request for an expression like this:

```
m do .... end + m do .... end
```

To allow for this, put the `do~end` iterator in `arg` or `primary`. Incidentally, the condition for `while` is `expr`, meaning it contains `arg` and `primary`, so the `do` will cause a conflict here. Basically, it looks like this:

```
while m do
  ...
end
```

At first glance, the `do` looks like the `do` of `while`. However, a closer look reveals that it could be a `m do~end` bundling. Something that's not obvious even to a person will definitely cause `yacc` to conflict. Let's try it in practice.

```
/* do conflict experiment */
%token kWHILE kDO tIDENTIFIER kEND
%%
```

```
expr: kWILE expr kDO expr kEND
| tIDENTIFIER
| tIDENTIFIER kDO expr kEND
```

I simplified the example to only include `while`, variable referencing and iterators. This rule causes a shift/reduce conflict if the head of the conditional contains `tIDENTIFIER`. If `tIDENTIFIER` is used for variable referencing and `do` is appended to `while`, then it's reduction. If it's made an iterator `do`, then it's a shift.

Unfortunately, in a shift/reduce conflict the shift is prioritized, so if left unchecked, `do` will become an iterator `do`. That said, even if a reduction is forced through operator priorities or some other method, `do` won't shift at all, becoming unusable. Thus, to solve the problem without any contradictions, we need to either deal with on the scanner level or write a rule that allows to use operators without putting the `do~end` iterator into `expr`.

However, not putting `do~end` into `expr` is not a realistic goal. That would require all rules for `expr` (as well as for `arg` and `primary`) to be repeated. This leaves us only the scanner solution.

## Rule-level solution

Below is a simplified example of a relevant rule.

### ▼ do symbol

```
primary      : kWILE expr_value do compstmt kEND
```

```

do          : term
| kDO_COND

primary    : operation brace_block
| method_call brace_block

brace_block : '{' opt_block_var compstmt '}'
| kDO opt_block_var compstmt kEND

```

As you can see, the terminal symbols for the `do` of `while` and for the iterator `do` are different. For the former it's `kDO_COND` while for the latter it's `kDO`. Then it's simply a matter of pointing that distinction out to the scanner.

## Symbol-level solution

Below is a partial view of the `yylex` section that processes reserved words. It's the only part tasked with processing `do` so looking at this code should be enough to understand the criteria for making the distinction.

### ▼ `yylex-Identifier-Reserved word`

```

4183          if (kw->id[0] == kDO) {
4184              if (COND_P()) return kDO_COND;
4185              if (CMDARG_P() && state != EXPR_CMDARG)
4186                  return kDO_BLOCK;
4187              if (state == EXPR_ENDARG)
4188                  return kDO_BLOCK;
4189              return kDO;
4190          }

(parse.y)

```

It's a little messy, but you only need the part associated with `kDO_COND`. That is because only two comparisons are meaningful. The first is the comparison between `kDO_COND` and `kDO/kDO_BLOCK`. The second is the comparison between `kDO` and `kDO_BLOCK`. The rest are meaningless. Right now we only need to distinguish the conditional `do` – leave all the other conditions alone.

Basically, `COND_P()` is the key.

## COND\_P()

### cond\_stack

`COND_P()` is defined close to the head of `parse.y`

#### ▼ cond\_stack

```
75 #ifdef HAVE_LONG_LONG
76 typedef unsigned LONG_LONG stack_type;
77 #else
78 typedef unsigned long stack_type;
79 #endif
80
81 static stack_type cond_stack = 0;
82 #define COND_PUSH(n) (cond_stack = (cond_stack<<1)|((n)&1))
83 #define COND_POP() (cond_stack >>= 1)
84 #define COND_LEXPPOP() do {\
85     int last = COND_P();\
86     cond_stack >>= 1;\
87     if (last) cond_stack |= 1;\
88 } while (0)
89 #define COND_P() (cond_stack&1)
```

(`parse.y`)

The type `stack_type` is either `long` (over 32 bit) or `long long` (over 64 bit). `cond_stack` is initialized by `yycompile()` at the start of parsing and after that is handled only through macros. All you need, then, is to understand those macros.

If you look at `COND_PUSH/POP` you will see that these macros use integers as stacks consisting of bits.

MSB $\leftarrow$	$\rightarrow$ LSB	
...	0000000000	Initial value 0
...	0000000001	COND_PUSH(1)
...	0000000010	COND_PUSH(0)
...	0000000101	COND_PUSH(1)
...	0000000010	COND_POP()
...	0000000100	COND_PUSH(0)
...	0000000010	COND_POP()

As for `COND_P()`, since it determines whether or not the least significant bit (LSB) is a 1, it effectively determines whether the head of the stack is a 1.

The remaining `COND_LEXPOP()` is a little weird. It leaves `COND_P()` at the head of the stack and executes a right shift. Basically, it “crushes” the second bit from the bottom with the lowermost bit.

MSB $\leftarrow$	$\rightarrow$ LSB	
...	0000000000	Initial value 0
...	0000000001	COND_PUSH(1)
...	0000000010	COND_PUSH(0)
...	0000000101	COND_PUSH(1)
...	0000000011	COND_LEXPOP()
...	0000000100	COND_PUSH(0)
...	0000000010	COND_LEXPOP()

((errata:

It leaves COND\_P() only when it is 1. When COND\_P() is 0 and the second bottom bit is 1, it would become 1 after doing LEXPOL, thus COND\_P() is not left in this case. ))

Now I will explain what that means.

# Investigating the function

Let us investigate the function of this stack. To do that I will list up all the parts where `COND_PUSH()` `COND_POP()` are used.

```
| kWILE {COND_PUSH(1);} expr_value do {COND_POP();}  
| kUNTIL {COND_PUSH(1);} expr_value do {COND_POP();}  
| kFOR block_var kIN {COND_PUSH(1);} expr_value do {COND_POP();}  
  
case '(':  
    :  
    :  
    COND_PUSH(0);  
    CMDARG_PUSH(0);  
  
case '[':  
    :  
    :  
    COND_PUSH(0);  
    CMDARG_PUSH(0);  
  
case '{':  
    :  
    :  
    COND_PUSH(0);  
    CMDARG_PUSH(0);  
  
case ']':
```

```

case '}':
case ')':
COND_LEXPOP();
CMDARG_LEXPOP();

```

From this we can derive the following general rules

- At the start of a conditional expression `PUSH(1)`
- At opening parenthesis `PUSH(0)`
- At the end of a conditional expression `POP()`
- At closing parenthesis `LEXPOP()`

With this, you should see how to use it. If you think about it for a minute, the name `cond_stack` itself is clearly the name for a macro that determines whether or not it's on the same level as the conditional expression (see image 2)

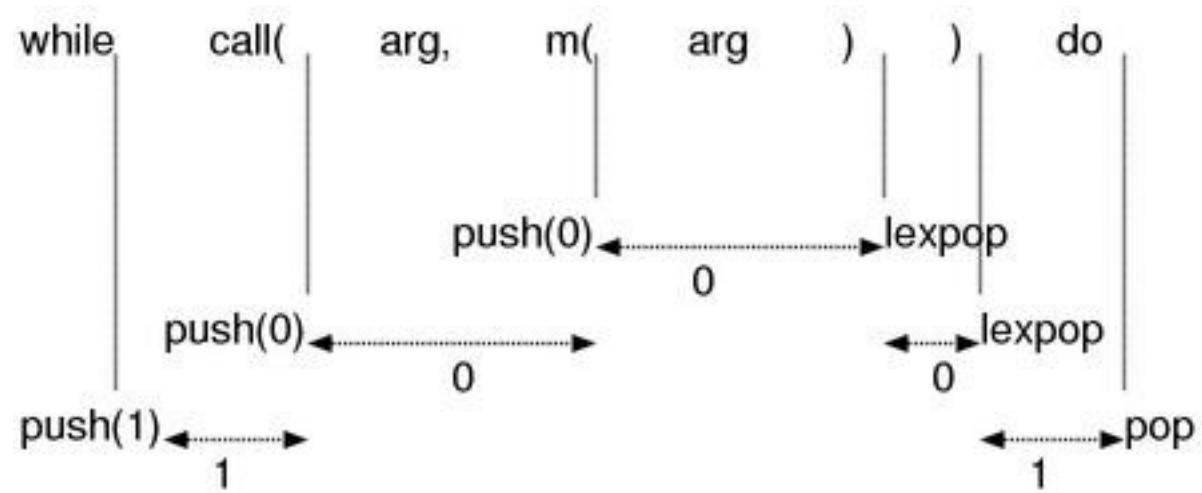


Figure 2: Changes of `COND_P()`

Using this trick should also make situations like the one shown below easy to deal with.

```
while (m do .... end)  # do is an iterator do(kD0)
```

end

....

This means that on a 32-bit machine in the absence of `long long` if conditional expressions or parentheses are nested at 32 levels, things could get strange. Of course, in reality you won't need to nest so deep so there's no actual risk.

Finally, the definition of `COND_LEXPOP()` looks a bit strange – that seems to be a way of dealing with lookahead. However, the rules now do not allow for lookahead to occur, so there's no purpose to make the distinction between `POP` and `LEXPOP`. Basically, at this time it would be correct to say that `COND_LEXPOP()` has no meaning.

## tLPAREN\_ARG(1)

### ■ The problem

This one is very complicated. It only became workable in Ruby 1.7 and only fairly recently. The core of the issue is interpreting this:

```
call (expr) + 1
```

As one of the following

```
(call(expr)) + 1
```

```
call((expr) + 1)
```

In the past, it was always interpreted as the former. That is, the parentheses were always treated as “Method parameter parentheses”. But since Ruby 1.7 it became possible to interpret it as the latter – basically, if a space is added, the parentheses become “Parentheses of expr”

I will also provide an example to explain why the interpretation changed. First, I wrote a statement as follows

```
p m() + 1
```

So far so good. But let's assume the value returned by `m` is a fraction and there are too many digits. Then we will have it displayed as an integer.

```
p m() + 1 .to_i    # ??
```

Uh-oh, we need parentheses.

```
p (m() + 1).to_i
```

How to interpret this? Up to 1.6 it will be this

```
(p(m() + 1)).to_i
```

The much-needed `to_i` is rendered meaningless, which is unacceptable. To counter that, adding a space between it and the

parentheses will cause the parentheses to be treated specially as expr parentheses.

For those eager to test this, this feature was implemented in `parse.y` revision 1.100(2001-05-31). Thus, it should be relatively prominent when looking at the differences between it and 1.99. This is the command to find the difference.

```
~/src/ruby % cvs diff -r1.99 -r1.100 parse.y
```

## Investigation

First let us look at how the set-up works in reality. Using the `ruby-lexer` tool{`ruby-lexer`: located in `tools/ruby-lexer.tar.gz` on the CD} we can look at the list of symbols corresponding to the program.

```
% ruby-lexer -e 'm(a)'  
tIDENTIFIER '(' tIDENTIFIER ')' '\n'
```

Similarly to Ruby, `-e` is the option to pass the program directly from the command line. With this we can try all kinds of things. Let's start with the problem at hand – the case where the first parameter is enclosed in parentheses.

```
% ruby-lexer -e 'm (a)'  
tIDENTIFIER tLPAREN_ARG tIDENTIFIER ')' '\n'
```

After adding a space, the symbol of the opening parenthesis became `tLPAREN_ARG`. Now let's look at normal expression

parentheses.

```
% ruby-lexer -e '(a)'  
tLPAREN tIDENTIFIER ')' '\n'
```

For normal expression parentheses it seems to be tLPAREN. To sum up:

## Input Symbol of opening parenthesis

m(a)	'('
m (a)	tLPAREN_ARG
(a)	tLPAREN

Thus the focus is distinguishing between the three. For now tLPAREN\_ARG is the most important.

## The case of one parameter

We'll start by looking at the yylex() section for '('

### ▼ yylex-'('

```
3841      case '(':  
3842          command_start = Qtrue;  
3843          if (lex_state == EXPR_BEG || lex_state == EXPR_MID)  
3844              c = tLPAREN;  
3845          }  
3846          else if (space_seen) {  
3847              if (lex_state == EXPR_CMDARG) {  
3848                  c = tLPAREN_ARG;  
3849              }  
3850              else if (lex_state == EXPR_ARG) {  
3851                  c = tLPAREN_ARG;  
3852                  yylval.id = last_id;
```

```
3853 }  
3854 }  
3855 COND_PUSH(0);  
3856 CMDARG_PUSH(0);  
3857 lex_state = EXPR_BEG;  
3858 return c;
```

(parse.y)

Since the first `if` is `tLPAREN` we're looking at a normal expression parenthesis. The distinguishing feature is that `lex_state` is either `BEG` or `MID` – that is, it's clearly at the beginning of the expression.

The following `space_seen` shows whether the parenthesis is preceded by a space. If there is a space and `lex_state` is either `ARG` or `CMDARG`, basically if it's before the first parameter, the symbol is not `'('` but `tLPAREN_ARG`. This way, for example, the following situation can be avoided

```
m(          # Parenthesis not preceded by a space. Method pa  
m arg, (  # Unless first parameter, expression parenthesis
```

When it is neither `tLPAREN` nor `tLPAREN_ARG`, the input character `c` is used as is and becomes `'('`. This will definitely be a method call parenthesis.

If such a clear distinction is made on the symbol level, no conflict should occur even if rules are written as usual. Simplified, it becomes something like this:

```
stmt      : command_call
```

```

method_call  : tIDENTIFIER '(' args ')'      /* Normal method */

command_call : tIDENTIFIER command_args      /* Method with parent

command_args : args

args          : arg
               : args ',' arg

arg           : primary

primary       : tLPAREN compstmt ')'
               | tLPAREN_ARG expr ')'
               | method_call

```

Now I need you to focus on `method_call` and `command_call`. If you leave the `'('` without introducing `tLPAREN_ARG`, then `command_args` will produce `args`, `args` will produce `arg`, `arg` will produce `primary`. Then, `'('` will appear from `tLPAREN_ARG` and conflict with `method_call` (see image 3)

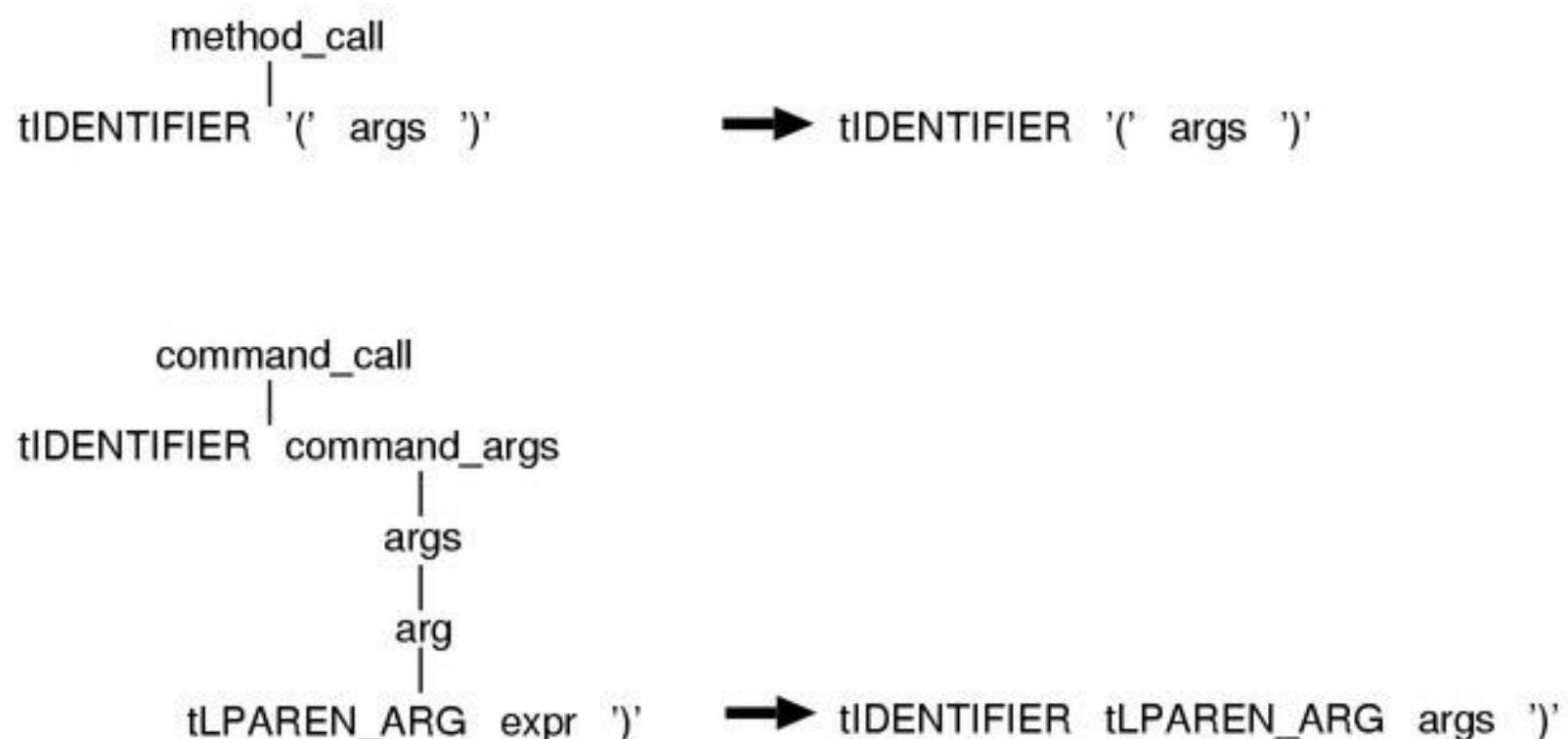


Figure 3: `method_call` and `command_call`

# The case of two parameters and more

One might think that if the parenthesis becomes `tLPAREN_ARG` all will be well. That is not so. For example, consider the following

```
m (a, a, a)
```

Before now, expressions like this one were treated as method calls and did not produce errors. However, if `tLPAREN_ARG` is introduced, the opening parenthesis becomes an `expr` parenthesis, and if two or more parameters are present, that will cause a parse error. This needs to be resolved for the sake of compatibility.

Unfortunately, rushing ahead and just adding a rule like

```
command_args : tLPAREN_ARG args ')'
```

will just cause a conflict. Let's look at the bigger picture and think carefully.

```
stmt      : command_call
           | expr

expr      : arg

command_call : tIDENTIFIER command_args

command_args : args
              | tLPAREN_ARG args ')'

args      : arg
           : args ',' arg
```

```

arg          : primary
primary      : tLPAREN compstmt ')'
| tLPAREN_ARG expr ')'
| method_call
method_call  : tIDENTIFIER '(' args ')'

```

Look at the first rule of `command_args`. Here, `args` produces `arg`. Then `arg` produces `primary` and out of there comes the `tLPAREN_ARG` rule. And since `expr` contains `arg` and as it is expanded, it becomes like this:

```

command_args : tLPAREN_ARG arg ')'
| tLPAREN_ARG arg ')'

```

This is a reduce/reduce conflict, which is very bad.

So, how can we deal with only 2+ parameters without causing a conflict? We'll have to write to accommodate for that situation specifically. In practice, it's solved like this:

## ▼ `command_args`

```

command_args      : open_args
open_args         : call_args
| tLPAREN_ARG    ')'
| tLPAREN_ARG call_args2 ')'
call_args         : command
| args opt_block_arg
| args ',' tSTAR arg_value opt_block_arg
| assocs opt_block_arg

```

```

| assocs ',' tSTAR arg_value opt_block_arg
| args ',' assocs opt_block_arg
| args ',' assocs ',' tSTAR arg opt_block_arg
| tSTAR arg_value opt_block_arg
| block_arg

call_args2 : arg_value ',' args opt_block_arg
| arg_value ',' block_arg
| arg_value ',' tSTAR arg_value opt_block_arg
| arg_value ',' args ',' tSTAR arg_value opt_block
| assocs opt_block_arg
| assocs ',' tSTAR arg_value opt_block_arg
| arg_value ',' assocs opt_block_arg
| arg_value ',' args ',' assocs opt_block_arg
| arg_value ',' assocs ',' tSTAR arg_value opt_blo
| arg_value ',' args ',' assocs ',' '
| | tSTAR arg_value opt_block_arg
| tSTAR arg_value opt_block_arg
| block_arg

primary : literal
| strings
| xstring
| :
| tLPAREN_ARG expr ')'

```

Here `command_args` is followed by another level – `open_args` which may not be reflected in the rules without consequence. The key is the second and third rules of this `open_args`. This form is similar to the recent example, but is actually subtly different. The difference is that `call_args2` has been introduced. The defining characteristic of this `call_args2` is that the number of parameters is always two or more. This is evidenced by the fact that most rules contain `','`. The only exception is `assocs`, but since `assocs` does not come out of `expr` it cannot conflict anyway.

That wasn't a very good explanation. To put it simply, in a grammar where this:

```
command_args      : call_args
```

doesn't work, and only in such a grammar, the next rule is used to make an addition. Thus, the best way to think here is "In what kind of grammar would this rule not work?" Furthermore, since a conflict only occurs when the `primary` of `tLPAREN_ARG` appears at the head of `call_args`, the scope can be limited further and the best way to think is "In what kind of grammar does this rule not work when a `tIDENTIFIER tLPAREN_ARG` line appears?" Below are a few examples.

```
m (a, a)
```

This is a situation when the `tLPAREN_ARG` list contains two or more items.

```
m ()
```

Conversely, this is a situation when the `tLPAREN_ARG` list is empty.

```
m (*args)
m (&block)
m (k => v)
```

This is a situation when the `tLPAREN_ARG` list contains a special expression (one not present in `expr`).

This should be sufficient for most cases. Now let's compare the above with a practical implementation.

## ▼ open\_args(1)

```
open_args      : call_args
| tLPAREN_ARG  ')'
```

First, the rule deals with empty lists

## ▼ open\_args(2)

```
                                | tLPAREN_ARG call_args2  ')'
call_args2      : arg_value ',' args opt_block_arg
| arg_value ',' block_arg
| arg_value ',' tSTAR arg_value opt_block_arg
| arg_value ',' args ',' tSTAR arg_value opt_block
| assocs opt_block_arg
| assocs ',' tSTAR arg_value opt_block_arg
| arg_value ',' assocs opt_block_arg
| arg_value ',' args ',' assocs opt_block_arg
| arg_value ',' assocs ',' tSTAR arg_value opt_blo
| arg_value ',' args ',' assocs ',' ,
|                                tSTAR arg_value opt_block_arg
| tSTAR arg_value opt_block_arg
| block_arg
```

And `call_args2` deals with elements containing special types such as `assocs`, passing of arrays or passing of blocks. With this, the scope is now sufficiently broad.

## ■ The problem

In the previous section I said that the examples provided should be sufficient for “most” special method call expressions. I said “most” because iterators are still not covered. For example, the below statement will not work:

```
m (a) {....}  
m (a) do .... end
```

In this section we will once again look at the previously introduced parts with solving this problem in mind.

## ■ Rule-level solution

Let us start with the rules. The first part here is all familiar rules, so focus on the `do_block` part

### ▼ command\_call

```
command_call      : command  
                  | block_command  
  
command          : operation command_args  
  
command_args     : open_args  
  
open_args         : call_args  
                  | tLPAREN_ARG ')'  
                  | tLPAREN_ARG call_args2 ')'
```

```
block_command      : block_call
block_call        : command do_block
do_block          : kDO_BLOCK opt_block_var compstmt '}'
| tLBRACE_ARG opt_block_var compstmt '}'
```

Both do and { are completely new symbols kDO\_BLOCK and tLBRACE\_ARG. Why isn't it kDO or '{' you ask? In this kind of situation the best answer is an experiment, so we will try replacing kDO\_BLOCK with kDO and tLBRACE\_ARG with '{' and processing that with yacc

```
% yacc parse.y
conflicts:  2 shift/reduce, 6 reduce/reduce
```

It conflicts badly. A further investigation reveals that this statement is the cause.

```
m (a), b {....}
```

That is because this kind of statement is already supposed to work. b{....} becomes primary. And now a rule has been added that concatenates the block with m. That results in two possible interpretations:

```
m((a), b) {....}
m((a), (b {....}))
```

This is the cause of the conflict – namely, a 2 shift/reduce conflict.

The other conflict has to do with do~end

```
m((a)) do .... end      # Add do~end using block_call
m((a)) do .... end      # Add do~end using primary
```

These two conflict. This is 6 reduce/reduce conflict.

## ▀ {~} iterator

This is the important part. As shown previously, you can avoid a conflict by changing the do and '{' symbols.

### ▼ yylex-'{'

```
3884      case '{':
3885          if (IS_ARG() || lex_state == EXPR_END)
3886              c = '{';           /* block (primary) */
3887          else if (lex_state == EXPR_ENDARG)
3888              c = tLBRACE_ARG; /* block (expr) */
3889          else
3890              c = tLBRACE;      /* hash */
3891          COND_PUSH(0);
3892          CMDARG_PUSH(0);
3893          lex_state = EXPR_BEG;
3894          return c;
```

(parse.y)

IS\_ARG() is defined as

### ▼ IS\_ARG

```
3104 #define IS_ARG() (lex_state == EXPR_ARG || lex_state == EXPR
```

(parse.y)

Thus, when the state is EXPR\_ENDARG it will always be false. In other words, when `lex_state` is EXPR\_ENDARG, it will always become tLBRACE\_ARG, so the key to everything is the transition to EXPR\_ENDARG.

## EXPR\_ENDARG

Now we need to know how to set EXPR\_ENDARG I used grep to find where it is assigned.

### ▼ Transition toEXPR\_ENDARG

```
open_args      : call_args
| tLPAREN_ARG {lex_state = EXPR_ENDARG;} ')'
| tLPAREN_ARG call_args2 {lex_state = EXPR_ENDARG; }

primary       : tLPAREN_ARG expr {lex_state = EXPR_ENDARG;} ')'
```

That's strange. One would expect the transition to EXPR\_ENDARG to occur after the closing parenthesis corresponding to tLPAREN\_ARG, but it's actually assigned before ')'. I ran grep a few more times thinking there might be other parts setting the EXPR\_ENDARG but found nothing.

Maybe there's some mistake. Maybe `lex_state` is being changed some other way. Let's use rubylex-analyser to visualize the `lex_state` transition.

```
% rubylex-analyser -e 'm (a) { nil }'
+EXPR_BEG
EXPR_BEG      C          "m"    tIDENTIFIER          EXPR_CMDARG
EXPR_CMDARG  S          "("    tLPAREN_ARG          EXPR_BEG
```

EXPR_BEG	C	"a" tIDENTIFIER	0:cond push
EXPR_CMDARG		")" ')	0:cmd push- EXPR_CMDARG EXPR_END
+EXPR_ENDARG			0:cond lexpop 1:cmd lexpop
EXPR_ENDARG	S	"{" tLBRACE_ARG	EXPR_BEG
			0:cond push
			10:cmd push
			0:cmd resume
EXPR_BEG	S	"nil" kNIL	EXPR_END
EXPR_END	S	"}" '}'	EXPR_END
			0:cond lexpop
			0:cmd lexpop
EXPR_END		"\n" \n	EXPR_BEG

The three big branching lines show the state transition caused by `yylex()`. On the left is the state before `yylex()` The middle two are the word text and its symbols. Finally, on the right is the `lex_state` after `yylex()`

The problem here are parts of single lines that come out as `+EXPR_ENDARG`. This indicates a transition occurring during parser action. According to this, for some reason an action is executed after reading the '))' a transition to `EXPR_ENDARG` occurs and '{' is nicely changed into `tLBRACE_ARG` This is actually a pretty high-level technique – generously (ab)using the LALR up to the (1).

## Abusing the lookahead

`ruby -y` can bring up a detailed display of the `yacc` parser engine. This time we will use it to more closely trace the parser.

```
% ruby -yce 'm (a) {nil}' 2>&1 | egrep '^Reading|Reducing'  
Reducing via rule 1 (line 303), -> @1  
Reading a token: Next token is 304 (tIDENTIFIER)  
Reading a token: Next token is 340 (tLPAREN_ARG)  
Reducing via rule 446 (line 2234), tIDENTIFIER -> operation  
Reducing via rule 233 (line 1222), -> @6  
Reading a token: Next token is 304 (tIDENTIFIER)  
Reading a token: Next token is 41 (')')  
Reducing via rule 392 (line 1993), tIDENTIFIER -> variable  
Reducing via rule 403 (line 2006), variable -> var_ref  
Reducing via rule 256 (line 1305), var_ref -> primary  
Reducing via rule 198 (line 1062), primary -> arg  
Reducing via rule 42 (line 593), arg -> expr  
Reducing via rule 260 (line 1317), -> @9  
Reducing via rule 261 (line 1317), tLPAREN_ARG expr @9 ')' -> pri  
Reading a token: Next token is 344 (tLBRACE_ARG)  
:  
:
```

Here we're using the option `-c` which stops the process at just compiling and `-e` which allows to give a program from the command line. And we're using `grep` to single out token read and reduction reports.

Start by looking at the middle of the list. `')'` is read. Now look at the end – the reduction (execution) of embedding action (@9) finally happens. Indeed, this would allow `EXPR_ENDARG` to be set after the `')'` before the `'{'`. But is this always the case? Let's take another look at the part where it's set.

```
Rule 1      tLPAREN_ARG  {lex_state = EXPR_ENDARG;} ')'  
Rule 2      tLPAREN_ARG  call_args2 {lex_state = EXPR_ENDARG;} ')'  
Rule 3      tLPAREN_ARG  expr  {lex_state = EXPR_ENDARG;} ')'
```

The embedding action can be substituted with an empty rule. For example, we can rewrite this using rule 1 with no change in meaning whatsoever.

```
target  : tLPAREN_ARG tmp ')'
tmp    :
{
    lex_state = EXPR_ENDARG;
}
```

Assuming that this is before `tmp`, it's possible that one terminal symbol will be read by lookahead. Thus we can skip the (empty) `tmp` and read the next. And if we are certain that lookahead will occur, the assignment to `lex_state` is guaranteed to change to `EXPR_ENDARG` after `)'`. But is `)'` certain to be read by lookahead in this rule?

## Ascertaining lookahead

This is actually pretty clear. Think about the following input.

```
m () { nil }      # A
m (a) { nil }     # B
m (a,b,c) { nil } # C
```

I also took the opportunity to rewrite the rule to make it easier to understand (with no actual changes).

```
rule1: tLPAREN_ARG          e1  ')'
rule2: tLPAREN_ARG  one_arg   e2  ')'
rule3: tLPAREN_ARG  more_args e3  ')'

e1: /* empty */
```

```
e2: /* empty */  
e3: /* empty */
```

First, the case of input A. Reading up to

```
m ( # ... tLPAREN_ARG
```

we arrive before the e1. If e1 is reduced here, another rule cannot be chosen anymore. Thus, a lookahead occurs to confirm whether to reduce e1 and continue with rule1 to the bitter end or to choose a different rule. Accordingly, if the input matches rule1 it is certain that ')' will be read by lookahead.

On to input B. First, reading up to here

```
m ( # ... tLPAREN_ARG
```

Here a lookahead occurs for the same reason as described above. Further reading up to here

```
m (a # ... tLPAREN_ARG '()' tIDENTIFIER
```

Another lookahead occurs. It occurs because depending on whether what follows is a ',', ' or a ')' a decision is made between rule2 and rule3. If what follows is a ',', ' then it can only be a comma to separate parameters, thus rule3 the rule for two or more parameters, is chosen. This is also true if the input is not a simple a but something like an if or literal. When the input is complete, a lookahead occurs to choose between rule2 and rule3 - the rules for

one parameter and two or more parameters respectively.

The presence of a separate embedding action is present before '))' in every rule. There's no going back after an action is executed, so the parser will try to postpone executing an action until it is as certain as possible. For that reason, situations when this certainty cannot be gained with a single lookahead should be excluded when building a parser as it is a conflict.

Proceeding to input C.

```
m (a, b, c
```

At this point anything other than `rule3` is unlikely so we're not expecting a lookahead. And yet, that is wrong. If the following is '()' then it's a method call, but if the following is ',', ' or ')' it needs to be a variable reference. Basically, this time a lookahead is needed to confirm parameter elements instead of embedding action reduction.

But what about the other inputs? For example, what if the third parameter is a method call?

```
m (a, b, c(....)    # ... ',', method_call
```

Once again a lookahead is necessary because a choice needs to be made between shift and reduction depending on whether what follows is ',', ' or ')'. Thus, in this rule in all instances the ')' is read before the embedding action is executed. This is quite

complicated and more than a little impressive.

But would it be possible to set `lex_state` using a normal action instead of an embedding action? For example, like this:

```
| tLPAREN_ARG ')' { lex_state = EXPR_ENDARG; }
```

This won't do because another lookahead is likely to occur before the action is reduced. This time the lookahead works to our disadvantage. With this it should be clear that abusing the lookahead of a LALR parser is pretty tricky and not something a novice should be doing.

## do~end iterator

So far we've dealt with the `{~}` iterator, but we still have `do~end` left. Since they're both iterators, one would expect the same solutions to work, but it isn't so. The priorities are different. For example,

```
m a, b {....}          # m(a, (b{....}))  
m a, b do .... end    # m(a, b) do....end
```

Thus it's only appropriate to deal with them differently.

That said, in some situations the same solutions do apply. The example below is one such situation

```
m (a) {....}  
m (a) do .... end
```

In the end, our only option is to look at the real thing. Since we're dealing with `do` here, we should look in the part of `yylex()` that handles reserved words.

## ▼ yylex-Identifiers-Reserved words-do

```
4183         if (kw->id[0] == kDO) {
4184             if (COND_P()) return kDO_COND;
4185             if (CMDARG_P() && state != EXPR_CMDARG)
4186                 return kDO_BLOCK;
4187             if (state == EXPR_ENDARG)
4188                 return kDO_BLOCK;
4189             return kDO;
4190         }
```

(`parse.y`)

This time we only need the part that distinguishes between `kDO_BLOCK` and `kDO`. Ignore `kDO_COND`. Only look at what's always relevant in a finite-state scanner.

The decision-making part using `EXPR_ENDARG` is the same as `TLBRACE_ARG` so priorities shouldn't be an issue here. Similarly to '`{`' the right course of action is probably to make it `kDO_BLOCK`

((errata:

In the following case, priorities should have an influence. (But it does not in the actual code. It means this is a bug.)

```
m m (a) { ... } # This should be interpreted as m(m(a) {...}),
                  # but is interpreted as m(m(a)) {...}
m m (a) do ... end # as the same as this: m(m(a)) do ... end
```

))

The problem lies with `CMDARG_P()` and `EXPR_CMDARG`. Let's look at both.

## CMDARG\_P()

### ▼ cmdarg\_stack

```
91 static stack_type cmdarg_stack = 0;
92 #define CMDARG_PUSH(n) (cmdarg_stack = (cmdarg_stack<<1) | ((n
93 #define CMDARG_POP() (cmdarg_stack >>= 1)
94 #define CMDARG_LEXPOP() do {\ \
95     int last = CMDARG_P(); \
96     cmdarg_stack >>= 1; \
97     if (last) cmdarg_stack |= 1; \
98 } while (0)
99 #define CMDARG_P() (cmdarg_stack&1)
```

(`parse.y`)

The structure and interface (macro) of `cmdarg_stack` is completely identical to `cond_stack`. It's a stack of bits. Since it's the same, we can use the same means to investigate it. Let's list up the places which use it. First, during the action we have this:

```
command_args : {
    $<num>$ = cmdarg_stack;
    CMDARG_PUSH(1);
}
open_args {
    /* CMDARG_POP() */
    cmdarg_stack = $<num>1;
    $$ = $2;
}
```

`$<num>$` represents the left value with a forced casting. In this case it comes out as the value of the embedding action itself, so it can be produced in the next action with `$<num>1`. Basically, it's a structure where `cmdarg_stack` is hidden in `$$` before `open_args` and then restored in the next action.

But why use a hide-restore system instead of a simple push-pop? That will be explained at the end of this section.

Searching `yylex()` for more CMDARG relations, I found this.

<b>Token</b>	<b>Relation</b>
--------------	-----------------

<code>'( ' ' [ ' ' { '</code>	<code>CMDARG_PUSH(0)</code>
<code>') ' ' ] ' ' }</code>	<code>CMDARG_LEXPOP()</code>

Basically, as long as it is enclosed in parentheses, `CMDARG_P()` is false.

Consider both, and it can be said that when `command_args` , a parameter for a method call with parentheses omitted, is not enclosed in parentheses `CMDARG_P()` is true.

## EXPR\_CMDARG

Now let's take a look at one more condition – `EXPR_CMDARG` Like before, let us look for place where a transition to `EXPR_CMDARG` occurs.

### ▼ yylex-Identifiers-State Transitions

```
4201         if (lex_state == EXPR_BEG ||  
4202             lex_state == EXPR_MID ||  
4203             lex_state == EXPR_DOT ||  
4204             lex_state == EXPR_ARG ||  
4205             lex_state == EXPR_CMDARG) {  
4206             if (cmd_state)  
4207                 lex_state = EXPR_CMDARG;  
4208             else  
4209                 lex_state = EXPR_ARG;  
4210         }  
4211         else {  
4212             lex_state = EXPR_END;  
4213         }  
4214     }  
4215     else {  
4216         lex_state = EXPR_ARG;  
4217     }  
4218 }
```

(parse.y)

This is code that handles identifiers inside `yylex()` Leaving aside that there are a bunch of `lex_state` tests in here, let's look first at `cmd_state` And what is this?

▼ `cmd_state`

```
3106 static int  
3107 yylex()  
3108 {  
3109     static ID last_id = 0;  
3110     register int c;  
3111     int space_seen = 0;  
3112     int cmd_state;  
3113  
3114     if (lex_strterm) {  
3115         /* .....omitted..... */  
3116     }  
3117     cmd_state = command_start;  
3118     command_start = Qfalse;  
3119 }
```

(parse.y)

Turns out it's an `yylex` local variable. Furthermore, an investigation using `grep` revealed that here is the only place where its value is altered. This means it's just a temporary variable for storing `command_start` during a single run of `yylex`

When does `command_start` become true, then?

▼ `command_start`

```
2327 static int command_start = 0true;
2334 static NODE*
2335 yycompile(f, line)
2336     char *f;
2337     int line;
2338 {
2339     :
2380     command_start = 1;
2381
2382     static int
2383     yylex()
2384     {
2385         :
2386         case '\n':
2387             /* .....omitted..... */
2388             command_start = 0true;
2389             lex_state = EXPR_BEG;
2390             return '\n';
2391
2392         case ';':
2393             command_start = 0true;
2394
2395         case '(':
2396             command_start = 0true;
2397
2398     (parse.y)
```

From this we understand that `command_start` becomes true when one of the `parse.y` static variables `\n` ; ( is scanned.

Summing up what we've covered up to now, first, when `\n` ; ( is read, `command_start` becomes true and during the next `yylex()` run `cmd_state` becomes true.

And here is the code in `yylex()` that uses `cmd_state`

## ▼ `yylex`-Identifiers-State transitions

```
4201         if (lex_state == EXPR_BEG ||  
4202             lex_state == EXPR_MID ||  
4203             lex_state == EXPR_DOT ||  
4204             lex_state == EXPR_ARG ||  
4205             lex_state == EXPR_CMDARG) {  
4206             if (cmd_state)  
4207                 lex_state = EXPR_CMDARG;  
4208             else  
4209                 lex_state = EXPR_ARG;  
4210         }  
4211         else {  
4212             lex_state = EXPR_END;  
4213         }  
  
(parse.y)
```

From this we understand the following: when after `\n` ; ( the state is `EXPR_BEG` `MID` `DOT` `ARG` `CMDARG` and an identifier is read, a transition to `EXPR_CMDARG` occurs. However, `lex_state` can only become `EXPR_BEG` following a `\n` ; ( so when a transition occurs to `EXPR_CMDARG` the `lex_state` loses its meaning. The `lex_state` restriction is only important to transitions dealing with `EXPR_ARG`

Based on the above we can now think of a situation where the state is EXPR\_CMDARG. For example, see the one below. The underscore is the current position.

m \_  
m(m \_  
m m \_

((errata:

The third one “m m \_” is not EXPR\_CMDARG. (It is EXPR\_ARG.) ))

## Conclusion

Let us now return to the `do` decision code.

### ▼ yylex-Identifiers-Reserved words-kD0-kD0\_BLOCK

```
4185 if (CMDARG_P() && state != EXPR_CMDARG)
4186     return kD0_BLOCK;
(parse.y)
```

Inside the parameter of a method call with parentheses omitted but not before the first parameter. That means from the second parameter of `command_call` onward. Basically, like this:

m arg, arg do .... end  
m (arg), arg do .... end

Why is the case of EXPR\_CMDARG excluded? This example should clear It up

```
m do .... end
```

This pattern can already be handled using the `do~end` iterator which uses `kDO` and is defined in `primary`. Thus, including that case would cause another conflict.

## Reality and truth

Did you think we're done? Not yet. Certainly, the theory is now complete, but only if everything that has been written is correct. As a matter of fact, there is one falsehood in this section. Well, more accurately, it isn't a falsehood but an inexact statement. It's in the part about `CMDARG_P()`

Actually, `CMDARG_P()` becomes true when inside `command_args`, that is to say, inside the parameter of a method call with parentheses omitted.

But where exactly is “inside the parameter of a method call with parentheses omitted”? Once again, let us use `rubylex-analyser` to inspect in detail.

```
% rubylex-analyser -e 'm a,a,a,a;'  
+EXPR_BEG  
EXPR_BEG C      "m"  tIDENTIFIER      EXPR_CMDARG  
EXPR_CMDARG S      "a"  tIDENTIFIER      EXPR_ARG  
EXPR_ARG          ","   ','           1:cmd push-  
EXPR_BEG          "a"  tIDENTIFIER      EXPR_BEG  
EXPR_ARG          ","   ','           EXPR_ARG  
EXPR_BEG          "a"  tIDENTIFIER      EXPR_ARG
```

EXPR_ARG		,	,		EXPR_BEG
EXPR_BEG		"a"	tIDENTIFIER		EXPR_ARG
EXPR_ARG		";"	;"		EXPR_BEG
				0:cmd resume	
EXPR_BEG	C	"\n"	'		EXPR_BEG

The 1:cmd push- in the right column is the push to cmd\_stack. When the rightmost digit in that line is 1 CMDARG\_P() become true. To sum up, the period of CMDARG\_P() can be described as:

From immediately after the first parameter of a method call with parentheses omitted To the terminal symbol following the final parameter

But, very strictly speaking, even this is still not entirely accurate.

% rubylex-analyser -e	'm	a()	,a,a;	'	
+EXPR_BEG					
EXPR_BEG	C	"m"	tIDENTIFIER		EXPR_CMDARG
EXPR_CMDARG	S	"a"	tIDENTIFIER		EXPR_ARG
		"("	'('	1:cmd push-	
EXPR_ARG		)"	')'	EXPR_BEG	
				0:cond push	
EXPR_BEG	C	)"	')'	10:cmd push	
				EXPR_END	
EXPR_END		,	,	0:cond lexpop	
EXPR_BEG		"a"	tIDENTIFIER	1:cmd lexpop	
EXPR_ARG		,	,	EXPR_BEG	
EXPR_BEG		"a"	tIDENTIFIER	EXPR_ARG	
EXPR_ARG		";"	;"	EXPR_BEG	
				0:cmd resume	
EXPR_BEG	C	"\n"	'	EXPR_BEG	

When the first terminal symbol of the first parameter has been

read, `CMDARG_P()` is true. Therefore, the complete answer would be:

From the first terminal symbol of the first parameter of a method call with parentheses omitted To the terminal symbol following the final parameter

What repercussions does this fact have? Recall the code that uses `CMDARG_P()`

## ▼ yylex-Identifiers-Reserved words-kD0-kD0\_BLOCK

```
4185 if (CMDARG_P() && state != EXPR_CMDARG
4186         return kD0_BLOCK;
(parse.y)
```

`EXPR_CMDARG` stands for “Before the first parameter of `command_call`” and is excluded. But wait, this meaning is also included in `CMDARG_P()`. Thus, the final conclusion of this section:

`EXPR_CMDARG` is completely useless

Truth be told, when I realized this, I almost broke down crying. I was sure it had to mean SOMETHING and spent enormous effort analyzing the source, but couldn’t understand anything. Finally, I ran all kind of tests on the code using `rubylex-analyser` and arrived at the conclusion that it has no meaning whatsoever.

I didn’t spend so much time doing something meaningless just to fill up more pages. It was an attempt to simulate a situation likely

to happen in reality. No program is perfect, all programs contain their own mistakes. Complicated situations like the one discussed here are where mistakes occur most easily, and when they do, reading the source material with the assumption that it's flawless can really backfire. In the end, when reading the source code, you can only trust the what actually happens.

Hopefully, this will teach you the importance of dynamic analysis. When investigating something, focus on what really happens. The source code will not tell you everything. It can't tell anything other than what the reader infers.

And with this very useful sermon, I close the chapter.

((errata:

This confidently written conclusion was wrong. Without EXPR\_CMDARG, for instance, this program “`m (m do end)`” cannot be parsed. This is an example of the fact that correctness is not proved even if dynamic analyses are done so many times. ))

## Still not the end

Another thing I forgot. I can't end the chapter without explaining why `CMDARG_P()` takes that value. Here's the problematic part:

▼ `command_args`

```
1209  command_args      :  {  
1210                      $<num>$ = cmdarg_stack;  
1211                      CMDARG_PUSH(1);
```

```

1212         }
1213         open_args
1214         {
1215             /* CMDARG_POP() */
1216             cmdarg_stack = $<num>1;
1217             $$ = $2;
1218         }
1221     open_args      : call_args
(parse.y)

```

All things considered, this looks like another influence from lookahead. `command_args` is always in the following context:

`tIDENTIFIER _`

Thus, this looks like a variable reference or a method call. If it's a variable reference, it needs to be reduced to `variable` and if it's a method call it needs to be reduced to `operation`. We cannot decide how to proceed without employing lookahead. Thus a lookahead always occurs at the head of `command_args` and after the first terminal symbol of the first parameter is read, `CMDARG_PUSH()` is executed.

The reason why `POP` and `LEXP0P` exist separately in `cmdarg_stack` is also here. Observe the following example:

```

% rubylex-analyser -e 'm m (a), a'
-e:1: warning: parenthesize argument(s) for future version
+EXPR_BEG
EXPR_BEG      C          "m"  tIDENTIFIER          EXPR_CMDARG
EXPR_CMDARG  S          "m"  tIDENTIFIER          EXPR_ARG
                                         1:cmd push-

```

EXPR_ARG	S	"(" tLPAREN_ARG	EXPR_BEG 0:cond push 10:cmd push 101:cmd push-
EXPR_BEG	C	"a" tIDENTIFIER	EXPR_CMDARG
EXPR_CMDARG		")" ')	EXPR_END 0:cond lexpop 11:cmd lexpop
+EXPR_ENDARG			
EXPR_ENDARG		"," ','	EXPR_BEG
EXPR_BEG	S	"a" tIDENTIFIER	EXPR_ARG
EXPR_ARG		"\n" \n	EXPR_BEG 10:cmd resume 0:cmd resume

Looking only at the parts related to cmd and how they correspond to each other...

1:cmd push-	parserpush(1)
10:cmd push	scannerpush
101:cmd push-	parserpush(2)
11:cmd lexpop	scannerpop
10:cmd resume	parserpop(2)
0:cmd resume	parserpop(1)

The cmd push- with a minus sign at the end is a parser push. Basically, push and pop do not correspond. Originally there were supposed to be two consecutive push- and the stack would become 110, but due to the lookahead the stack became 101 instead. CMDARG\_LEXPOP() is a last-resort measure to deal with this. The scanner always pushes 0 so normally what it pops should also always be 0. When it isn't 0, we can only assume that it's 1 due to the parser push being late. Thus, the value is left.

Conversely, at the time of the parser pop the stack is supposed to be

back in normal state and usually pop shouldn't cause any trouble. When it doesn't do that, the reason is basically that it should work right. Whether popping or hiding in \$\$ and restoring, the process is the same. When you consider all the following alterations, it's really impossible to tell how lookahead's behavior will change. Moreover, this problem appears in a grammar that's going to be forbidden in the future (that's why there is a warning). To make something like this work, the trick is to consider numerous possible situations and respond them. And that is why I think this kind of implementation is right for Ruby. Therein lies the real solution.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](#)

# Ruby Hacking Guide

# Chapter 12: Syntax tree construction

## Node

---

### ▀ NODE

As I've already described, a Ruby program is first converted to a syntax tree. To be more precise, a syntax tree is a tree structure made of structs called “nodes”. In `ruby`, all nodes are of type `NODE`.

### ▼ NODE

```
128  typedef struct RNode {
129      unsigned long flags;
130      char *nd_file;
131      union {
132          struct RNode *node;
133          ID id;
134          VALUE value;
135          VALUE (*cfunc)(ANYARGS);
136          ID *tbl;
137      } u1;
138      union {
139          struct RNode *node;
140          ID id;
141          int argc;
```

```
142     VALUE value;
143 } u2;
144 union {
145     struct RNode *node;
146     ID id;
147     long state;
148     struct global_entry *entry;
149     long cnt;
150     VALUE value;
151 } u3;
152 } NODE;
```

(node.h)

Although you might be able to infer from the struct name `RNode`, nodes are Ruby objects. This means the creation and release of nodes are taken care of by the ruby's garbage collector.

Therefore, `flags` naturally has the same role as `basic.flags` of the object struct. It means that `T_NODE` which is the type of a struct and flags such as `FL_FREEZE` are stored in it. As for `NODE`, in addition to these, its node type is stored in `flags`.

What does it mean? Since a program could contain various elements such as `if` and `while` and `def` and so on, there are also various corresponding node types. The three available union are complicated, but how these unions are used is decided to only one specific way for each node. For example, the below table shows the case when it is `NODE_IF` that is the node of `if`.

<b>member</b>	<b>union member</b>	<b>role</b>
u1	<code>u1.node</code>	the condition expression
u2	<code>u2.node</code>	the body of true

And, in `node.h`, the macros to access each union member are available.

## ▼ the macros to access NODE

```
166 #define nd_head    u1.node
167 #define nd_alen   u2.argv
168 #define nd_next    u3.node
169
170 #define nd_cond    u1.node
171 #define nd_body    u2.node
172 #define nd_else    u3.node
173
174 #define nd_orig    u3.value
:
:
:
```

(`node.h`)

For example, these are used as follows:

```
NODE *head, *tail;
head->nd_next = tail; /* head->u3.node = tail */
```

In the source code, it's almost certain that these macros are used. A very few exceptions are only the two places where creating NODE in `parse.y` and where marking NODE in `gc.c`.

By the way, what is the reason why such macros are used? For one thing, it might be because it's cumbersome to remember numbers like `u1` that are not meaningful by just themselves. But what is

more important than that is, there should be no problem if the corresponding number is changed and it's possible that it will actually be changed. For example, since a condition clause of `if` does not have to be stored in `u1`, someone might want to change it to `u2` for some reason. But if `u1` is directly used, he needs to modify a lot of places all over the source codes, it is inconvenient. Since nodes are all declared as `NODE`, it's hard to find nodes that represent `if`. By preparing the macros to access, this kind of trouble can be avoided and conversely we can determine the node types from the macros.

## ■ Node Type

I said that in the `flags` of a `NODE` struct its node type is stored. We'll look at in what form this information is stored. A node type can be set by `nd_set_type()` and obtained by `nd_type()`.

### ▼ `nd_type` `nd_set_type`

```
156 #define nd_type(n) (((RNODE(n))->flags>>FL_USHIFT)&0xff)
157 #define nd_set_type(n,t) \
158     RNODE(n)->flags = ((RNODE(n)->flags & ~FL_UMASK) \
                           | (((t)<<FL_USHIFT) & FL_UMASK))

(node.h)
```

### ▼ `FL_USHIFT` `FL_UMASK`

```
418 #define FL_USHIFT    11
429 #define FL_UMASK   (0xff<<FL_USHIFT)
```

It won't be so much trouble if we'll keep focus on around `nd_type`. Fig.1 shows how it seems like.

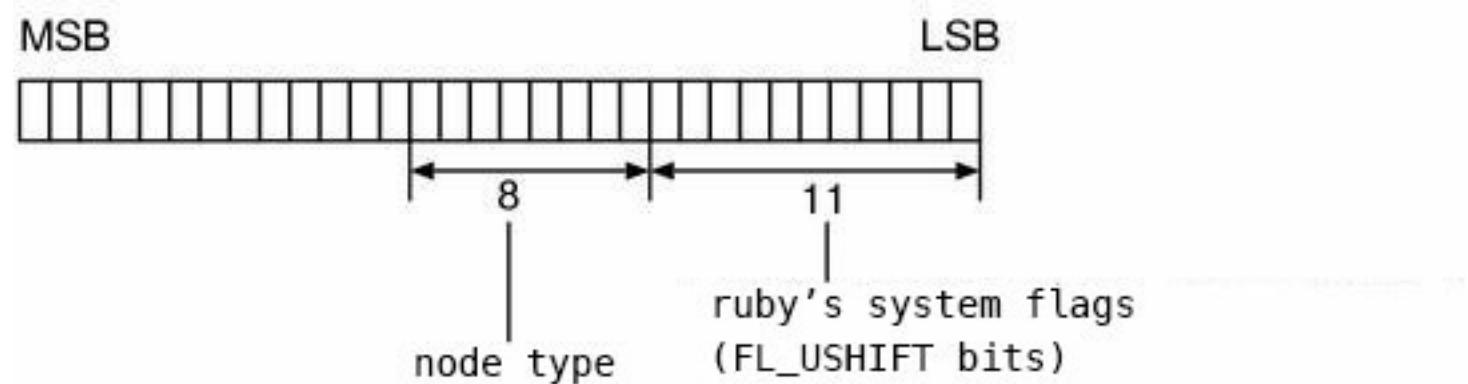


Fig.1: The usage of RNode.flags

And, since macros cannot be used from debuggers, the `nodetype()` function is also available.

### ▼ nodetype

```

4247 static enum node_type
4248 nodetype(node) /* for debug */
4249     NODE *node;
4250 {
4251     return (enum node_type)nd_type(node);
4252 }
```

(parse.y)

## File Name and Line Number

The `nd_file` of a `NODE` holds (the pointer to) the name of the file where the text that corresponds to this node exists. Since there's

the file name, we naturally expect that there's also the line number, but the corresponding member could not be found around here. Actually, the line number is being embedded to `flags` by the following macro:

### ▼ `nd_line` `nd_set_line`

```
160 #define NODE_LSHIFT (FL_USHIFT+8)
161 #define NODE_LMASK (((long)1<<(sizeof(NODE*)*CHAR_BIT-NODE_
162 #define nd_line(n) \
    ((unsigned int)((RNODE(n)->flags >> NODE_LSHIFT) & NODE_
163 #define nd_set_line(n,l) \
164     RNODE(n)->flags = ((RNODE(n)->flags & ~(-1 << NODE_LSHIF
                           | (((l)&NODE_LMASK) << NODE_LSHIFT))
```

(node.h)

`nd_set_line()` is fairly spectacular. However, as the names suggest, it is certain that `nd_set_line()` and `nd_line` works symmetrically. Thus, if we first examine the simpler `nd_line()` and grasp the relationship between the parameters, there's no need to analyze `nd_set_line()` in the first place.

The first thing is `NODE_LSHIFT`, as you can guess from the description of the node types of the previous section, it is the number of used bits in `flags`. `FL_USHIFT` is reserved by system of ruby (11 bits, `ruby.h`), 8 bits are for its node type.

The next thing is `NODE_LMASK`.

`sizeof(NODE*) * CHAR_BIT - NODE_LSHIFT`

This is the number of the rest of the bits. Let's assume it is `restbits`. This makes the code a lot simpler.

```
#define NODE_LMASK (((long)1 << restbits) - 1)
```

Fig.2 shows what the above code seems to be doing. Note that a borrow occurs when subtracting 1. We can eventually understand that `NODE_LMASK` is a sequence filled with 1 whose size is the number of the bits that are still available.

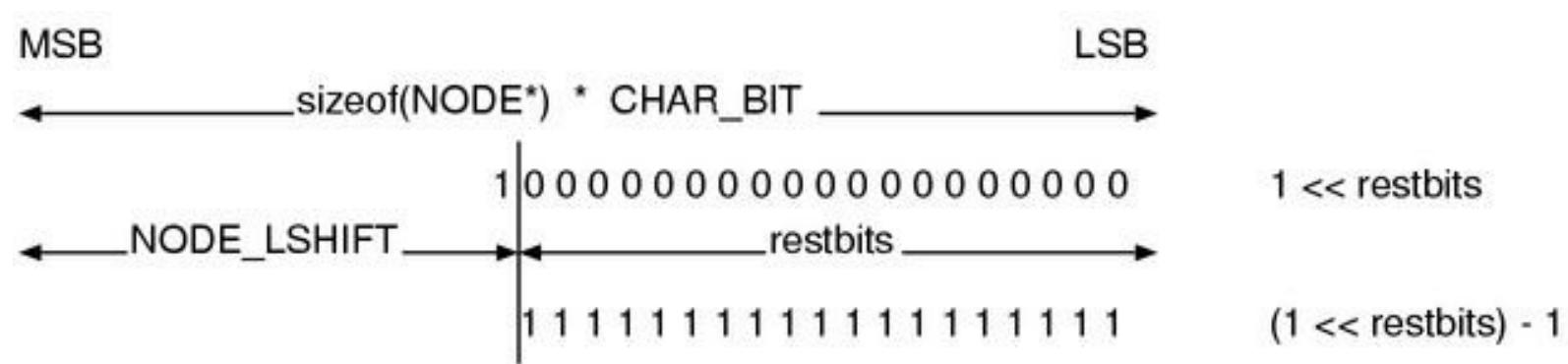


Fig.2: NODE\_LMASK

Now, let's look at `nd_line()` again.

```
(RNODE(n)->flags >> NODE_LSHIFT) & NODE_LMASK
```

By the right shift, the unused space is shifted to the LSB. The bitwise AND leaves only the unused space. Fig.3 shows how `flags` is used. Since `FL_USHIFT` is 11, in 32-bit machine  $32 - (11 + 8) = 13$  bits are available for the line number.

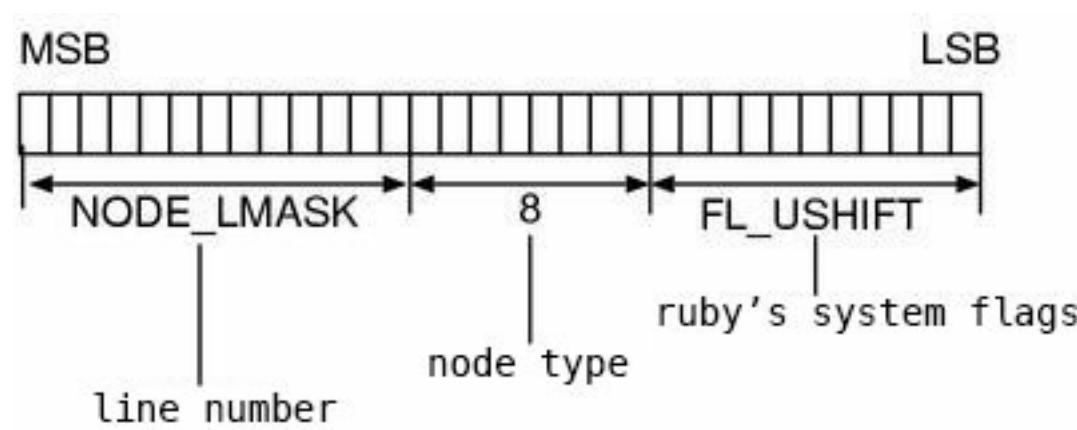


Fig.3: How flags are used at NODE

... This means, if the line numbers becomes beyond  $2^{13}=8192$ , the line numbers should wrongly be displayed. Let's try.

```
File.open('overflow.rb', 'w') {|f|
  10000.times { f.puts }
  f.puts 'raise'
}
```

With my 686 machine, ruby overflow.rb properly displayed 1809 as a line number. I've succeeded. However, if you use 64-bit machine, you need to create a little bigger file in order to successfully fail.

## rb\_node\_newnode()

Lastly let's look at the function rb\_node\_newnode() that creates a node.

### ▼ rb\_node\_newnode()

```
4228 NODE*
4229 rb_node_newnode(type, a0, a1, a2)
4230     enum node_type type;
4231     NODE *a0, *a1, *a2;
```

```
4232  {
4233      NODE *n = (NODE*) rb_newobj();
4234
4235      n->flags |= T_NODE;
4236      nd_set_type(n, type);
4237      nd_set_line(n, ruby_sourceline);
4238      n->nd_file = ruby_sourcefile;
4239
4240      n->u1.node = a0;
4241      n->u2.node = a1;
4242      n->u3.node = a2;
4243
4244      return n;
4245 }
```

(parse.y)

We've seen `rb_newobj()` in the Chapter 5: Garbage collection. It is the function to get a vacant RVALUE. By attaching the `T_NODE` struct-type flag to it, the initialization as a VALUE will complete. Of course, it's possible that some values that are not of type `NODE*` are passed for `u1 u2 u3`, but received as `NODE*` for the time being. Since the syntax trees of `ruby` does not contain `double` and such, if the values are received as pointers, it will never be too small in size.

For the rest part, you can forget about the details you've learned so far, and assume `NODE` is

- `flags`
- `nodetype`
- `nd_line`
- `nd_file`
- `u1`

- u2
- u3

a struct type that has the above seven members.

## Syntax Tree Construction

---

The role of the parser is to convert the source code that is a byte sequence to a syntax tree. Although the grammar passed, it does not finish even half of the task, so we have to assemble nodes and create a tree. In this section, we'll look at the construction process of that syntax tree.

### ■ YYSTYPE

Essentially this chapter is about actions, thus YYSTYPE which is the type of \$\$ or \$1 becomes important. Let's look at the %union of ruby first.

### ▼ %union declaration

```
170 %union {  
171     NODE *node;  
172     ID id;  
173     int num;  
174     struct RVarmap *vars;  
175 }
```

(parse.y)

struct RVarmap is a struct used by the evaluator and holds a block local variable. You can tell the rest. The most used one is of course node.

## Landscape with Syntax Trees

I mentioned that looking at the fact first is a theory of code reading. Since what we want to know this time is how the generated syntax tree is, we should start with looking at the answer (the syntax tree).

It's also nice using debuggers to observe every time, but you can visualize the syntax tree more handily by using the tool nodedump contained in the attached CD-ROM, This tool is originally the NodeDump made by [Pragmatic Programmers](#) and remodeled for this book. The original version shows quite explanatory output, but this remodeled version deeply and directly displays the appearance of the syntax tree.

For example, in order to dump the simple expression `m(a)`, you can do as follows:

```
% ruby -rnodedump -e 'm(a)'  
NODE_NEWLINE  
nd_file = "-e"  
nd_nth = 1  
nd_next:  
  NODE_FCALL  
  nd_mid = 9617 (m)  
  nd_args:
```

```
NODE_ARRAY
nd_alen = 1
nd_head:
    NODE_VCALL
        nd_mid = 9625 (a)
nd_next = (null)
```

The `-r` option is used to specify the library to be load, and the `-e` is used to pass a program. Then, the syntax tree expression of the program will be dumped.

I'll briefly explain about how to see the content. `NODE_NEWLINE` and `NODE_FCALL` and such are the node types. What are written at the same indent level of each node are the contents of its node members. For example, the root is `NODE_NEWLINE`, and it has the three members: `nd_file` `nd_nth` `nd_next`. `nd_file` points to the "`-e`" string of C, and `nd_nth` points to the `1` integer of C, and `nd_next` holds the next node `NODE_CALL`. But since these explanation in text are probably not intuitive, I recommend you to also check Fig.4 at the same time.

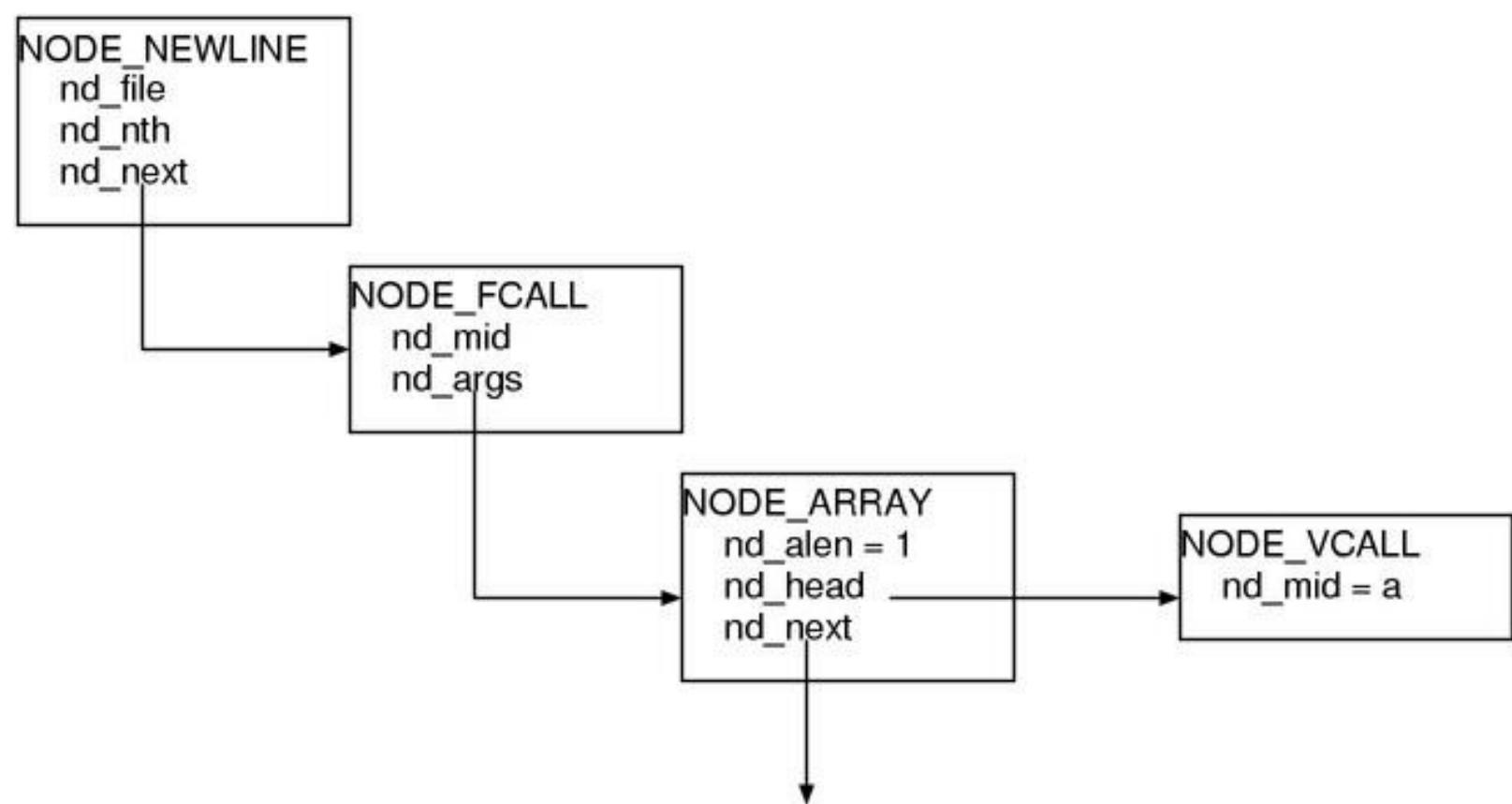


Fig.4: Syntax Tree

I'll explain the meaning of each node. **NODE\_CALL** is a Function CALL. **NODE\_ARRAY** is as its name suggests the node of array, and here it expresses the list of arguments. **NODE\_VCALL** is a Variable or CALL, a reference to undefined local variable will become this.

Then, what is **NODE\_NEWLINE**? This is the node to join the name of the currently executed file and the line number at runtime and is set for each `stmt`. Therefore, when only thinking about the meaning of the execution, this node can be ignored. When you require `nodedump-short` instead of `nodedump`, distractions like **NODE\_NEWLINE** are left out in the first place. Since it is easier to see if it is simple, `nodedump-short` will be used later on except for when particularly written.

Now, we'll look at the three type of composing elements in order to grasp how the whole syntax tree is. The first one is the leaves of a syntax tree. Next, we'll look at expressions that are combinations of that leaves, this means they are branches of a syntax tree. The last one is the list to list up the statements that is the trunk of a syntax tree in other words.

## Leaf

First, let's start with the edges that are the leaves of the syntax tree. Literals and variable references and so on, among the rules, they are what belong to `primary` and are particularly simple even among the `primary` rules.

```
% ruby -rnodedump-short -e '1'  
NODE_LIT  
nd_lit = 1:Fixnum
```

1 as a numeric value. There's not any twist. However, notice that what is stored in the node is not 1 of C but 1 of Ruby (1 of `Fixnum`). This is because ...

```
% ruby -rnodedump-short -e ':sym'  
NODE_LIT  
nd_lit = 9617:Symbol
```

This way, `Symbol` is represented by the same `NODE_LIT` when it becomes a syntax tree. As the above example, `VALUE` is always stored in `nd_lit` so it can be handled completely in the same way whether

it is a `Symbol` or a `Fixnum` when executing. In this way, all we need to do when dealing with it are retrieving the value in `nd_lit` and returning it. Since we create a syntax tree in order to execute it, designing it so that it becomes convenient when executing is the right thing to do.

```
% ruby -rnodedump-short -e '"a"'
NODE_STR
nd_lit = "a":String
```

A string. This is also a Ruby string. String literals are copied when actually used.

```
% ruby -rnodedump -e '[0,1]'
NODE_NEWLINE
nd_file = "-e"
nd_nth  = 1
nd_next:
  NODE_ARRAY
  nd_alen = 2
  nd_head:
    NODE_LIT
    nd_lit = 0:Fixnum
  nd_next:
    NODE_ARRAY
    nd_alen = 1
    nd_head:
      NODE_LIT
      nd_lit = 1:Fixnum
    nd_next = (null)
```

Array. I can't say this is a leaf, but let's allow this to be here because it's also a literal. It seems like a list of `NODE_ARRAY` hung with each element node. The reason why only in this case I didn't use `nodedump-short` is ... you will understand after finishing to read

this section.

## Branch

Next, we'll focus on “combinations” that are branches. `if` will be taken as an example.

`if`

I feel like `if` is always used as an example, that's because its structure is simple and there's not any reader who don't know about `if`, so it is convenient for writers.

Anyway, this is an example of `if`. For example, let's convert this code to a syntax tree.

### ▼ The Source Program

```
if true
  'true expr'
else
  'false expr'
end
```

### ▼ Its syntax tree expression

```
NODE_IF
  nd_cond:
    NODE_TRUE
  nd_body:
    NODE_STR
    nd_lit = "true expr":String
```

```
nd_else:  
  NODE_STR  
  nd_lit = "false expr":String
```

Here, the previously described `nodedump-short` is used, so `NODE_NEWLINE` disappeared. `nd_cond` is the condition, `nd_body` is the body of the true case, `nd_else` is the body of the false case.

Then, let's look at the code to build this.

## ▼ if rule

```
1373          | kIF expr_value then  
1374          compstmt  
1375          if_tail  
1376          kEND  
1377          {  
1378          $$ = NEW_IF(cond($2), $4, $5);  
1379          fixpos($$, $2);  
1380          }  
  
(parse.y)
```

It seems that `NEW_IF()` is the macro to create `NODE_IF`. Among the values of the symbols, `$2 $4 $5` are used, thus the correspondences between the symbols of the rule and `$n` are:

kIF	expr_value	then	compstmt	if_tail	kEND
\$1	\$2	\$3	\$4	\$5	\$6
NEW_IF(expr_value,			compstmt,	if_tail)	

this way. In other words, `expr_value` is the condition expression, `compstmt ($4)` is the case of true, `if_tail` is the case of false.

On the other hand, the macros to create nodes are all named `NEW_xxxx`, and they are defined `node.h`. Let's look at `NEW_IF()`.

### ▼ `NEW_IF()`

```
243 #define NEW_IF(c,t,e) rb_node_newnode(NODE_IF,c,t,e)  
(node.h)
```

As for the parameters, it seems that `c` represents condition, `t` represents then, and `e` represents else respectively. As described at the previous section, the order of members of a node is not so meaningful, so you don't need to be careful about parameter names in this kind of place.

And, the `code()` which processes the node of the condition expression in the action is a semantic analysis function. This will be described later.

Additionally, `fixpos()` corrects the line number. `NODE` is initialized with the file name and the line number of the time when it is “created”. However, for instance, the code of `if` should already be parsed by `end` by the time when creating `NODE_IF`. Thus, the line number would go wrong if it remains untouched. Therefore, it needs to be corrected by `fixpos()`.

```
fixpos(dest, src)
```

This way, the line number of the node `dest` is set to the one of the

node src. As for `if`, the line number of the condition expression becomes the line number of the whole `if` expression.

## elseif

Subsequently, let's look at the rule of `if_tail`.

### ▼ if\_tail

```
1543  if_tail          : opt_else
1544          | kELSIF expr_value then
1545          | compstmt
1546          | if_
1547          {
1548          |         $$ = NEW_IF(cond($2), $4, $5);
1549          |         fixpos($$, $2);
1550          }
1553  opt_else          : none
1554          | kELSE compstmt
1555          {
1556          |         $$ = $2;
1557          }
```

(`parse.y`)

First, this rule expresses “a list ends with `opt_else` after zero or more number of `elseif` clauses”. That's because, `if_tail` appears again and again while `elseif` continues, it disappears when `opt_else` comes in. We can understand this by extracting arbitrary times.

```
if_tail: kELSIF .... if_tail
if_tail: kELSIF .... kELSIF .... if_tail
if_tail: kELSIF .... kELSIF .... kELSIF .... if_tail
if_tail: kELSIF .... kELSIF .... kELSIF .... opt_else
```

```
if_tail: kELSIF .... kELSIF .... kELSIF .... kELSE compstmt
```

Next, let's focus on the actions, surprisingly, `elsif` uses the same `NEW_IF()` as `if`. It means, the below two programs will lose the difference after they become syntax trees.

```
if cond1
  body1
elsif cond2
  body2
elsif cond3
  body3
else
  body4
end
```

```
if cond1
  body1
else
  if cond2
    body2
  else
    if cond3
      body3
    else
      body4
    end
  end
end
```

Come to think of it, in C language and such, there's no distinction between the two also at the syntax level. Thus this might be a matter of course. Alternatively, the conditional operator (`a?b:c`) becomes indistinguishable from `if` statement after they become syntax trees.

The precedences was very meaningful when it was in the context of grammar, but they become unnecessary any more because the structure of a syntax tree contains that information. And, the difference in appearance such as `if` and the conditional operator become completely meaningless, its meaning (its behavior) only matters. Therefore, there's perfectly no problem if `if` and the

conditional operator are the same in its syntax tree expression.

I'll introduce a few more examples. `add` and `&&` become the same. `or` and `||` are also equal to each other. `not` and `!`, `if` and modifier `if`, and so on. These pairs also become equal to each other.

## Left Recursive and Right Recursive

By the way, the symbol of a list was always written at the left side when expressing a list in Chapter 9: yacc crash course. However, have you noticed it becomes opposite in `if_tail`? I'll show only the crucial part again.

```
if_tail: opt_else
| kELSIF ... if_tail
```

Surely, it is opposite of the previous examples. `if_tail` which is the symbol of a list is at the right side.

In fact, there's another established way of expressing lists,

```
list: END_ITEM
| ITEM list
```

when you write in this way, it becomes the list that contains continuous zero or more number of `ITEM` and ends with `END_ITEM`.

As an expression of a list, whichever is used it does not create a so much difference, but the way that the actions are executed is fatally different. With the form that `list` is written at the right, the actions

are sequentially executed from the last ITEM. We've already learned about the behavior of the stack of when list is at the left, so let's try the case that list is at the right. The input is 4 ITEMs and END\_ITEM.

ITEM	empty at first
ITEM ITEM	shift ITEM
ITEM ITEM ITEM	shift ITEM
ITEM ITEM ITEM ITEM	shift ITEM
ITEM ITEM ITEM ITEM END_ITEM	shift END_ITEM
ITEM ITEM ITEM ITEM list	reduce END_ITEM to list
ITEM ITEM ITEM list	reduce ITEM list to list
ITEM ITEM list	reduce ITEM list to list
ITEM list	reduce ITEM list to list
list	accept.

When list was at the left, shifts and reductions were done in turns. This time, as you see, there are continuous shifts and continuous reductions.

The reason why if\_tail places “list at the right” is to create a syntax tree from the bottom up. When creating from the bottom up, the node of if will be left in hand in the end. But if defining if\_tail by placing “list at the left”, in order to eventually leave the node of if in hand, it needs to traverse all links of the elseif and every time elseif is found add it to the end. This is cumbersome.

And, slow. Thus, `if_tail` is constructed in the “list at the right” manner.

Finally, the meaning of the headline is, in grammar terms, “the left is list” is called left-recursive, “the right is list” is called right-recursive. These terms are used mainly when reading papers about processing grammars or writing a book of yacc.

## Trunk

Leaf, branch, and finally, it’s trunk. Let’s look at how the list of statements are joined.

### ▼ The Source Program

```
7
8
9
```

The dump of the corresponding syntax tree is shown below. This is not `nodedump-short` but in the perfect form.

### ▼ Its Syntax Tree

```
NODE_BLOCK
nd_head:
  NODE_NEWLINE
  nd_file = "multistmt"
  nd_nth  = 1
  nd_next:
    NODE_LIT
    nd_lit = 7:Fixnum
nd_next:
```

```
NODE_BLOCK
nd_head:
  NODE_NEWLINE
  nd_file = "multistmt"
  nd_nth  = 2
  nd_next:
    NODE_LIT
    nd_lit = 8:Fixnum
nd_next:
  NODE_BLOCK
  nd_head:
    NODE_NEWLINE
    nd_file = "multistmt"
    nd_nth  = 3
    nd_next:
      NODE_LIT
      nd_lit = 9:Fixnum
nd_next = (null)
```

We can see the list of NODE\_BLOCK is created and NODE\_NEWLINE are attached as headers. (Fig.5)

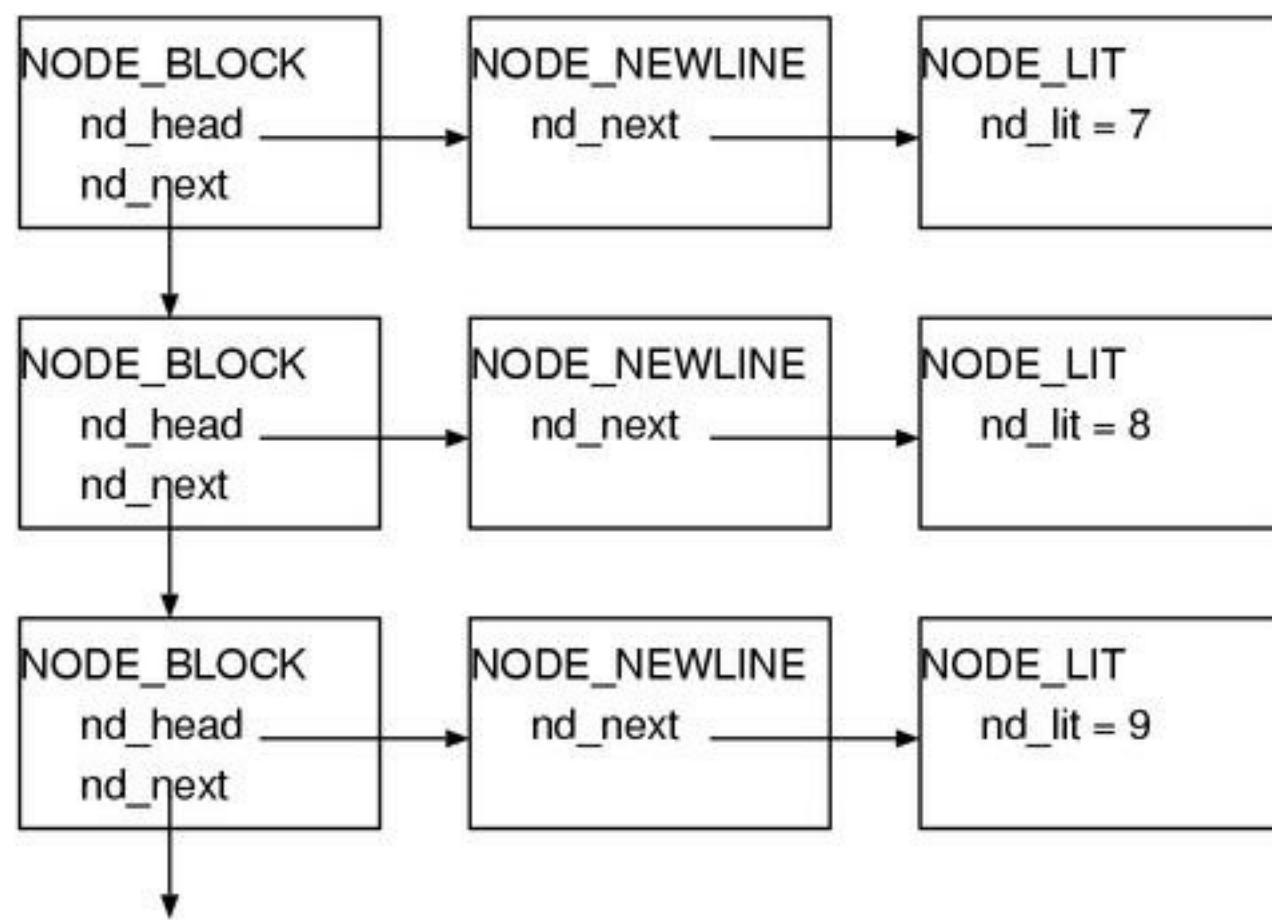


Fig.5: NODE\_BLOCK and NODE\_NEWLINE

It means, for each statement (`stmt`) `NODE_NEWLINE` is attached, and when they are multiple, it will be a list of `NODE_BLOCK`. Let's also see the code.

## ▼ stmts

```
354     stmts      : none
355             | stmt
356             {
357                 $$ = newline_node($1);
358             }
359             | stmts terms stmt
360             {
361                 $$ = block_append($1, newline_node($2));
362             }
```

`newline_node()` caps `NODE_NEWLINE`, `block_append()` appends it to the list. It's straightforward. Let's look at the content only of the `block_append()`.

## block\_append()

In this function, the error checks are in the very middle and obstructive. Thus I'll show the code without that part.

### ▼ `block_append()` (omitted)

```
4285 static NODE*
4286 block_append(head, tail)
4287     NODE *head, *tail;
4288 {
4289     NODE *end;
4290
4291     if (tail == 0) return head;
4292     if (head == 0) return tail;
4293
4294     if (nd_type(head) != NODE_BLOCK) {
4295         end = NEW_BLOCK(head);
4296         end->nd_end = end; /* (A-1) */
4297         fixpos(end, head);
4298         head = end;
4299     }
4300     else {
4301         end = head->nd_end; /* (A-2) */
4302     }
4303
4304     /* .....omitted..... */
4305
4306     if (nd_type(tail) != NODE_BLOCK) {
4307         tail = NEW_BLOCK(tail);
4308         tail->nd_end = tail;
4309     }
4310     end->nd_next = tail;
```

```

4330     head->nd_end = tail->nd_end; /* (A-3) */
4331     return head;
4332 }

(parse.y)

```

According to the previous syntax tree dump, NEW\_BLOCK was a linked list uses `nd_next`. Being aware of it while reading, it can be read “if either `head` or `tail` is not `NODE_BLOCK`, wrap it with `NODE_BLOCK` and join the lists each other.”

Additionally, on (A-1~3), the `nd_end` of the `NODE_BLOCK` of the head of the list always points to the `NODE_BLOCK` of the tail of the list. This is probably because in this way we don’t have to traverse all elements when adding an element to the tail (Fig.6). Conversely speaking, when you need to add elements later, `NODE_BLOCK` is suitable.

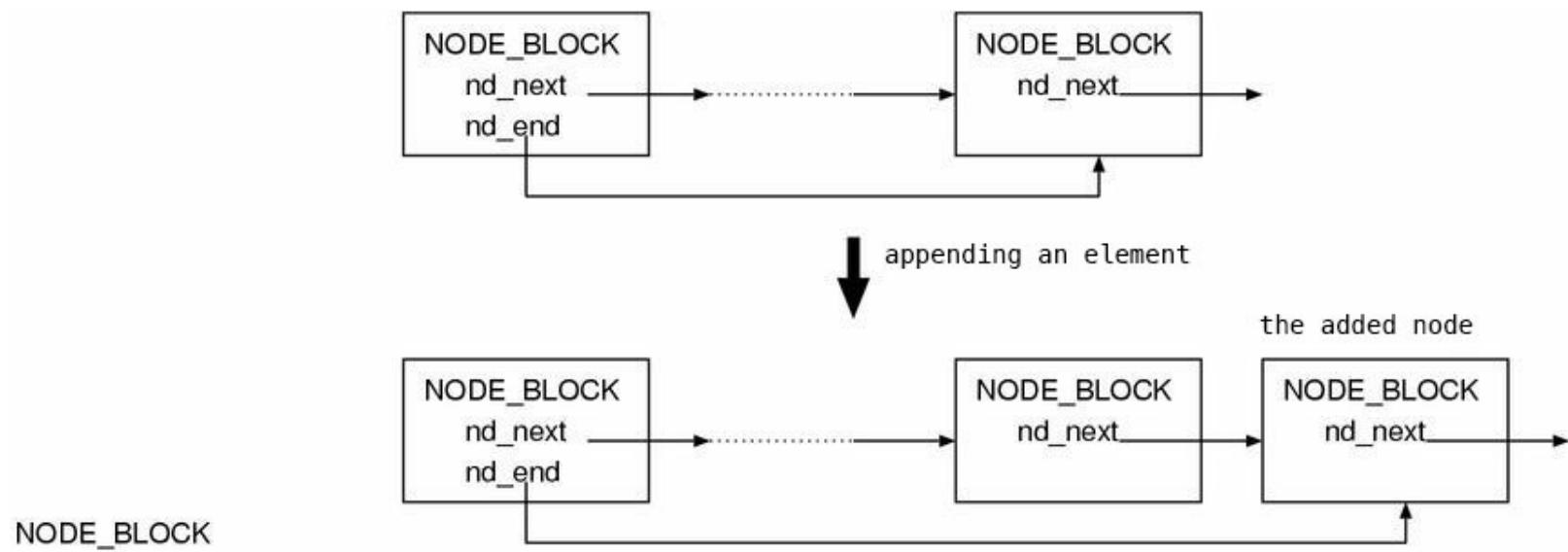


Fig.6: Appending is easy.

## The two types of lists

Now, I've explained the outline so far. Because the structure of syntax tree will also appear in Part 3 in large amounts, we won't go further as long as we are in Part 2. But before ending, there's one more thing I'd like to talk about. It is about the two general-purpose lists.

The two general-purpose lists mean `BLOCK` and `LIST`. `BLOCK` is, as previously described, a linked list of `NODE_BLOCK` to join the statements. `LIST` is, although it is called `LIST`, a list of `NODE_ARRAY`. This is what is used for array literals. `LIST` is used to store the arguments of a method or the list of multiple assignments.

As for the difference between the two lists, looking at the usage of the nodes is helpful to understand.

`NODE_BLOCK` `nd_head` holding an element

`nd_end` pointing to the `NODE_BLOCK` of the end of the list  
    `nd_next` pointing to the next `NODE_BLOCK`

`NODE_ARRAY` `nd_head` holding an element

`nd_alen` the length of the list that follows this node  
    `nd_next` pointing to the next `NODE_ARRAY`

The usage differs only in the second elements that are `nd_end` and `nd_alen`. And this is exactly the significance of the existence of each type of the two nodes. Since its size can be stored in `NODE_ARRAY`, we use an `ARRAY` list when the size of the list will frequently be required. Otherwise, we use a `BLOCK` list that is very fast to join. I don't describe this topic in details because the codes that use them

is necessary to understand the significance but not shown here, but when the codes appear in Part 3, I'd like you to recall this and think "Oh, this uses the length".

## Semantic Analysis

---

As I briefly mentioned at the beginning of Part 2, there are two types of analysis that are appearance analysis and semantic analysis. The appearance analysis is mostly done by yacc, the rest is doing the semantic analysis inside actions.

### ■ Errors inside actions

What does the semantic analysis precisely mean? For example, there are type checks in a language that has types. Alternatively, check if variables with the same name are not defined multiple times, and check if variables are not used before their definitions, and check if the procedure being used is defined, and check if `return` is not used outside of procedures, and so on. These are part of the semantic analysis.

What kind of semantic analysis is done in the current `ruby`? Since the error checks occupies almost all of semantic analysis in `ruby`, searching the places where generating errors seems a good way. In a parser of yacc, `yyerror()` is supposed to be called when an error

occurs. Conversely speaking, there's an error where `yyerror()` exists. So, I made a list of the places where calling `yyerror()` inside the actions.

- an expression not having its value (void value expression) at a place where a value is required
- an alias of `$n`
- `BEGIN` inside of a method
- `END` inside of a method
- `return` outside of methods
- a local variable at a place where constant is required
- a `class` statement inside of a method
- an invalid parameter variable (`$gvar` and `CONST` and such)
- parameters with the same name appear twice
- an invalid receiver of a singleton method (`def () .method` and such)
- a singleton method definition on literals
- an odd number of a list for hash literals
- an assignment to `self/nil/true/false/_FILE__/_LINE__`
- a constant assignment inside of a method
- a multiple assignment inside of a conditional expression

These checks can roughly be categorized by each purpose as follows:

- for the better error message
- in order not to make the rule too complex
- the others (pure semantic analysis)

For example, “return outside of a method” is a check in order not to make the rule too complex. Since this error is a problem of the structure, it can be dealt with by grammar. For example, it’s possible by defining the rules separately for both inside and outside of methods and making the list of all what are allowed and what are not allowed respectively. But this is in any way cumbersome and rejecting it in an action is far more concise.

And, “an assignment to self” seems a check for the better error message. In comparison to “return outside of methods”, rejecting it by grammar is much easier, but if it is rejected by the parser, the output would be just “parse error”. Comparing to it, the current

```
% ruby -e 'self = 1'  
-e:1: Can't change the value of self  
self = 1  
^
```

this error is much more friendly.

Of course, we can not always say that an arbitrary rule is exactly “for this purpose”. For example, as for “return outside of methods”, this can also be considered that this is a check “for the better error message”. The purposes are overlapping each other.

Now, the problem is “a pure semantic analysis”, in Ruby there are few things belong to this category. In the case of a typed language, the type analysis is a big event, but because variables are not typed in Ruby, it is meaningless. What is standing out instead is the

cheek of an expression that has its value.

To put “having its value” precisely, it is “you can obtain a value as a result of evaluating it”. `return` and `break` do not have values by themselves. Of course, a value is passed to the place where `return` to, but not any values are left at the place where `return` is written. Therefore, for example, the next expression is odd,

```
i = return(1)
```

Since this kind of expressions are clearly due to misunderstanding or simple mistakes, it’s better to reject when compiling. Next, we’ll look at `value_expr` which is one of the functions to check if it takes a value.

## ▀ `value_expr()`

`value_expr()` is the function to check if it is an `expr` that has a value.

### ▼ `value_expr()`

```
4754 static int
4755 value_expr(node)
4756     NODE *node;
4757 {
4758     while (node) {
4759         switch (nd_type(node)) {
4760             case NODE_CLASS:
4761             case NODE_MODULE:
4762             case NODE_DEFN:
4763             case NODE_DEFS:
4764                 rb_warning("void value expression");
4765                 return Qfalse;
```

```
4766
4767         case NODE_RETURN:
4768         case NODE_BREAK:
4769         case NODE_NEXT:
4770         case NODE_REDO:
4771         case NODE_RETRY:
4772             yyerror("void value expression");
4773             /* or "control never reach"? */
4774             return Qfalse;
4775
4776         case NODE_BLOCK:
4777             while (node->nd_next) {
4778                 node = node->nd_next;
4779             }
4780             node = node->nd_head;
4781             break;
4782
4783         case NODE_BEGIN:
4784             node = node->nd_body;
4785             break;
4786
4787         case NODE_IF:
4788             if (!value_expr(node->nd_body)) return Qfalse;
4789             node = node->nd_else;
4790             break;
4791
4792         case NODE_AND:
4793         case NODE_OR:
4794             node = node->nd_2nd;
4795             break;
4796
4797         case NODE_NEWLINE:
4798             node = node->nd_next;
4799             break;
4800
4801         default:
4802             return Qtrue;
4803         }
4804     }
4805
4806     return Qtrue;
4807 }
```

# Algorithm

Summary: It sequentially checks the nodes of the tree, if it hits “an expression certainly not having its value”, it means the tree does not have any value. Then it warns about that by using `rb_warning()` and return `Qfalse`. If it finishes to traverse the entire tree without hitting any “an expression not having its value”, it means the tree does have a value. Thus it returns `Qtrue`.

Here, notice that it does not always need to check the whole tree. For example, let’s assume `value_expr()` is called on the argument of a method. Here:

▼ check the value of `arg` by using `value_expr()`

```

1055 arg_value      : arg
1056           {
1057           value_expr($1);
1058           $$ = $1;
1059       }

(parse.y)

```

Inside of this argument `$1`, there can also be other nesting method calls again. But, the argument of the inside method must have been already checked with `value_expr()`, so you don’t have to check it again.

Let’s think more generally. Assume an arbitrary grammar element

A exists, and assume `value_expr()` is called against its all composing elements, the necessity to check the element A again would disappear.

Then, for example, how is `if`? Is it possible to be handled as if `value_expr()` has already called for all elements? If I put only the bottom line, it isn't. That is because, since `if` is a statement (which does not use a value), the main body should not have to return a value. For example, in the next case:

```
def method
  if true
    return 1
  else
    return 2
  end
  5
end
```

This `if` statement does not need a value.

But in the next case, its value is necessary.

```
def method( arg )
  tmp = if arg
    then 3
    else 98
  end
  tmp * tmp / 3.5
end
```

So, in this case, the `if` statement must be checked when checking the entire assignment expression. This kind of things are laid out in the `switch` statement of `value_expr()`.

# Removing Tail Recursion

By the way, when looking over the whole `value_expr`, we can see that there's the following pattern appears frequently:

```
while (node) {
    switch (nd_type(node)) {
        case NODE_XXXX:
            node = node->nd_xxxx;
            break;
            :
            :
    }
}
```

This expression will also carry the same meaning after being modified to the below:

```
return value_expr(node->nd_xxxx)
```

A code like this which does a recursive call just before `return` is called a tail recursion. It is known that this can generally be converted to `goto`. This method is often used when optimizing. As for Scheme, it is defined in specifications that tail recursions must be removed by language processors. This is because recursions are often used instead of loops in Lisp-like languages.

However, be careful that tail recursions are only when “calling just before `return`”. For example, take a look at the `NODE_IF` of `value_expr()`,

```
if (!value_expr(node->nd_body)) return Qfalse;  
node = node->nd_else;  
break;
```

As shown above, the first time is a recursive call. Rewriting this to the form of using `return`,

```
return value_expr(node->nd_body) && value_expr(node->nd_else);
```

If the left `value_expr()` is false, the right `value_expr()` is also executed. In this case, the left `value_expr()` is not “just before” `return`. Therefore, it is not a tail recursion. Hence, it can’t be extracted to `goto`.

## ▀ The whole picture of the value check

As for value checks, we won’t read the functions further. You might think it’s too early, but all of the other functions are, as the same as `value_expr()`, step-by-step one-by-one only traversing and checking nodes, so they are completely not interesting. However, I’d like to cover the whole picture at least, so I finish this section by just showing the call graph of the relevant functions (Fig.7).

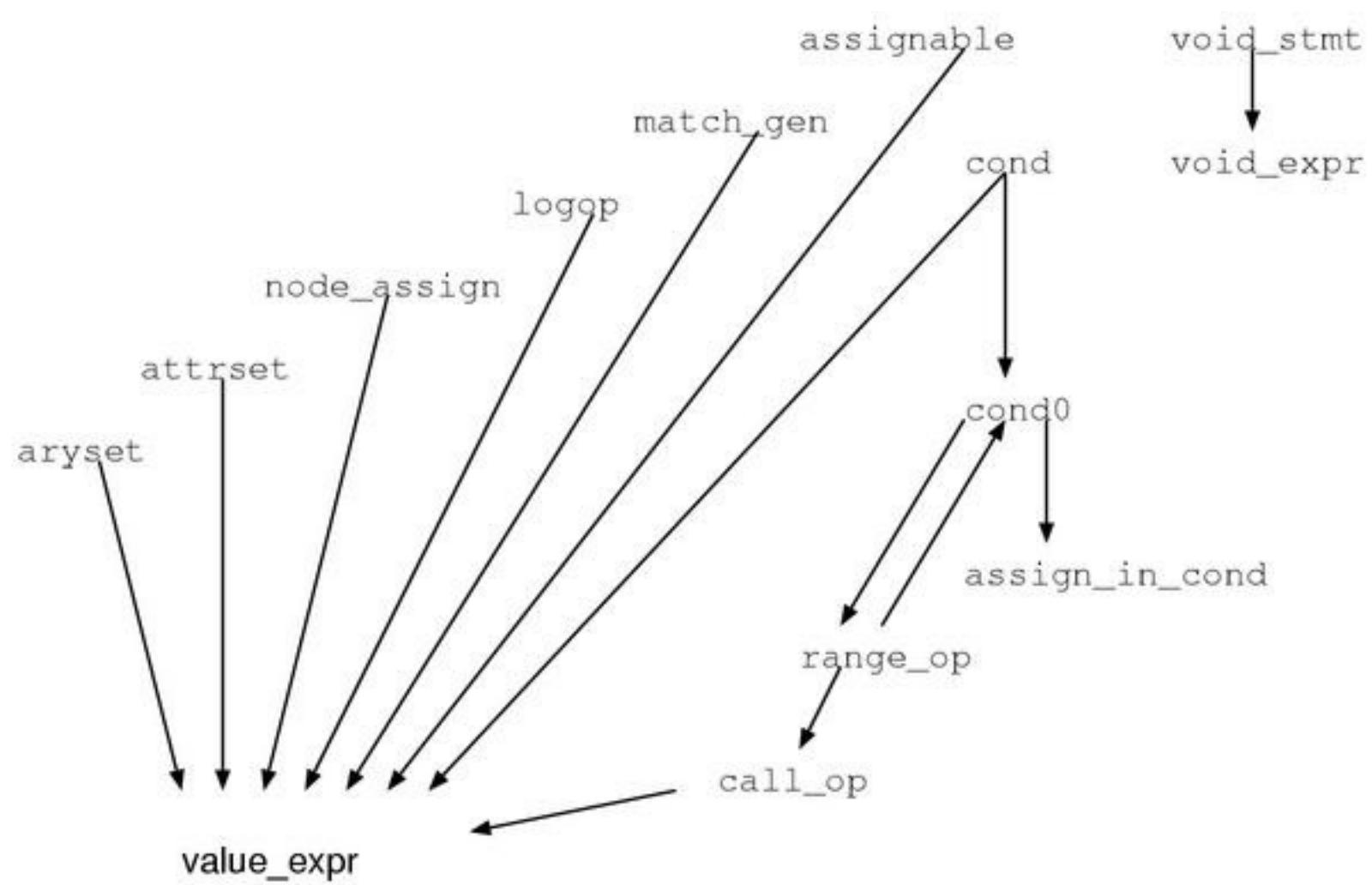


Fig.7: the call graph of the value check functions

## Local Variables

### Local Variable Definitions

The variable definitions in Ruby are really various. As for constants and class variables, these are defined on the first assignment. As for instance variables and global variables, as all names can be considered that they are already defined, you can refer them without assigning beforehand (although it produces warnings).

The definitions of local variables are again completely different from the above all. A local variable is defined when its assignment appears on the program. For example, as follows:

```
lvar = nil
p lvar      # being defined
```

In this case, as the assignment to `lvar` is written at the first line, in this moment `lvar` is defined. When it is undefined, it ends up with a runtime exception `NameError` as follows:

```
% ruby lvar.rb
lvar.rb:1: undefined local variable or method `lvar'
for #<Object:0x40163a9c> (NameError)
```

Why does it say "local variable or method"? As for methods, the parentheses of the arguments can be omitted when calling, so when there's not any arguments, it can't be distinguished from local variables. To resolve this situation, ruby tries to call it as a method when it finds an undefined local variable. Then if the corresponding method is not found, it generates an error such as the above one.

By the way, it is defined when "it appears", this means it is defined even though it was not assigned. The initial value of a defined variable is `nil`.

```
if false
  lvar = "this assigment will never be executed"
end
p lvar  # shows nil
```

Moreover, since it is defined “when” it “appears”, the definition has to be before the reference in a symbol sequence. For example, in the next case, it is not defined.

```
p lvar      # not defined !
lvar = nil  # although appearing here ...
```

Be careful about the point of “in the symbol sequence”. It has completely nothing to do with the order of evaluations. For example, for the next code, naturally the condition expression is evaluated first, but in the symbol sequence, at the moment when `p` appears the assignment to `lvar` has not appeared yet. Therefore, this produces `NameError`.

```
p(lvar) if lvar = true
```

What we’ve learned by now is that the local variables are extremely influenced by the appearances. When a symbol sequence that expresses an assignment appears, it will be defined in the appearance order. Based on this information, we can infer that `ruby` seems to define local variables while parsing because the order of the symbol sequence does not exist after leaving the parser. And in fact, it is true. In `ruby`, the parser defines local variables.

## Block Local Variables

The local variables newly defined in an iterator block are called

block local variables or dynamic variables. Block local variables are, in language specifications, identical to local variables. However, these two differ in their implementations. We'll look at how is the difference from now on.

## The data structure

We'll start with the local variable table `struct local_vars`.

### ▼ `struct local_vars`

```
5174 static struct local_vars {  
5175     ID *tbl;                      /* the table of local variab  
5176     int nofree;                   /* whether it is used from o  
5177     int cnt;                      /* the size of the tbl array  
5178     int dlev;                     /* the nesting level of dyna  
5179     struct RVarmap* dyna_vars;    /* block local variable name  
5180     struct local_vars *prev;  
5181 } *lvtbl;  
  
(parse.y)
```

The member name `prev` indicates that the `struct local_vars` is a opposite-direction linked list. ... Based on this, we can expect a stack. The simultaneously declared global variable `lvtbl` points to `local_vars` that is the top of that stack.

And, `struct RVarmap` is defined in `env.h`, and is available to other files and is also used by the evaluator. This is used to store the block local variables.

### ▼ `struct RVarmap`

```
52 struct RVarmap {  
53     struct RBasic super;  
54     ID id;           /* the variable name */  
55     VALUE val;       /* its value */  
56     struct RVarmap *next;  
57 };
```

(env.h)

Since there's `struct RBasic` at the top, this is a Ruby object. It means it is managed by the garbage collector. And since it is joined by the `next` member, it is probably a linked list.

Based on the observation we've done and the information that will be explained, Fig.8 illustrates the image of both structs while executing the parser.

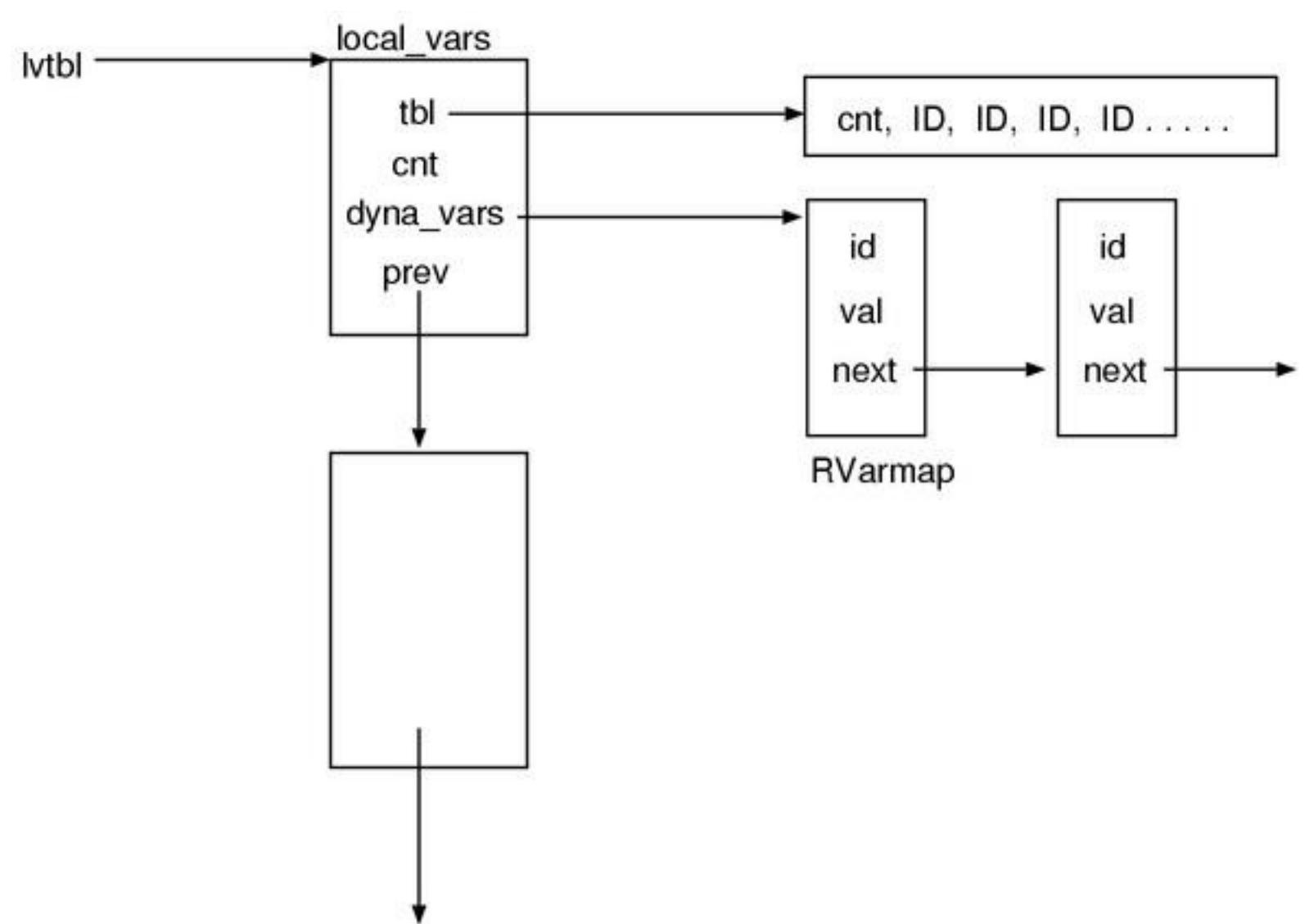


Fig.8: The image of local variable tables at runtime

## Local Variable Scope

When looking over the list of function names of `parse.y`, we can find functions such as `local_push()` `local_pop()` `local_cnt()` are laid out. In whatever way of thinking, they appear to be relating to a local variable. Moreover, because the names are `push` `pop`, it is clearly a stack. So first, let's find out the places where using these functions.

▼ `local_push()` `local_pop()` used examples

```

1475 | kDEF fname
1476 {
1477     $<id>$ = cur_mid;
1478     cur_mid = $2;
1479     in_def++;
1480     local_push(0);
1481 }
1482 f_arglist
1483 bodystmt
1484 kEND
1485 {
1486     /* NOEX_PRIVATE for toplevel */
1487     $$ = NEW_DEFN($2, $4, $5,
1488                 class_nest?NOEX_PUBLIC:NOEX_PRIV
1489                 if (is_attrset_id($2))
1490                     $$->nd_noex = NOEX_PUBLIC;
1491                     fixpos($$, $4);
1492                     local_pop();
1493                     in_def--;
1494                     cur_mid = $<id>3;
1495     }

```

(parse.y)

At `def`, I could find the place where it is used. It can also be found in class definitions and singleton class definitions, and module definitions. In other words, it is the place where the scope of local variables is cut. Moreover, as for how they are used, it does `push` where the method definition starts and does `pop` when the definition ends. This means, as we expected, it is almost certain that the functions start with `local_` are relating to local variables. And it is also revealed that the part between `push` and `pop` is probably a local variable scope.

Moreover, I also searched `local_cnt()`.

## ▼ NEW\_LASGN()

```
269 #define NEW_LASGN(v, val) rb_node_newnode(NODE_LASGN, v, val, lo  
(node.h)
```

This is found in `node.h`. Even though there are also the places where using in `parse.y`, I found it in the other file. Thus, probably I'm in desperation.

This `NEW_LASGN` is “new local assignment”. This should mean the node of an assignment to a local variable. And also considering the place where using it, the parameter `v` is apparently the local variable name. `val` is probably (a syntax tree that represents) the right-hand side value

Based on the above observations, `local_push()` is at the beginning of the local variable, `local_cnt()` is used to add a local variable if there's a local variable assignment in the halfway, `local_pop()` is used when ending the scope. This perfect scenario comes out. (Fig.9)

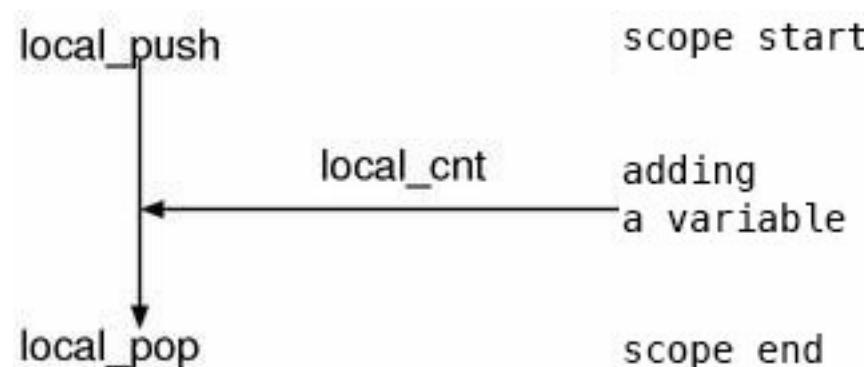


Fig.9: the flow of the local variable management

Then, let's look at the content of the function.

## push and pop

### ▼ local\_push()

```
5183 static void
5184 local_push(top)
5185     int top;
5186 {
5187     struct local_vars *local;
5188
5189     local = ALLOC(struct local_vars);
5190     local->prev = lvtbl;
5191     local->nofree = 0;
5192     local->cnt = 0;
5193     local->tbl = 0;
5194     local->dlev = 0;
5195     local->dyna_vars = ruby_dyna_vars;
5196     lvtbl = local;
5197     if (!top) {
5198         /* preserve the variable table of the previous scope
5199         rb_dvar_push(0, (VALUE)ruby_dyna_vars);
5200         ruby_dyna_vars->next = 0;
5201     }
5202 }
```

(parse.y)

As we expected, it seems that `struct local_vars` is used as a stack. Also, we can see `lvtbl` is pointing to the top of the stack. The lines relates to `rb_dvar_push()` will be read later, so it is left untouched for now.

Subsequently, we'll look at `local_pop()` and `local_tbl()` at the same

time.

▼ local\_tbl local\_pop

```
5218 static ID*
5219 local_tbl()
5220 {
5221     lvtbl->nofree = 1;
5222     return lvtbl->tbl;
5223 }

5204 static void
5205 local_pop()
5206 {
5207     struct local_vars *local = lvtbl->prev;
5208
5209     if (lvtbl->tbl) {
5210         if (!lvtbl->nofree) free(lvtbl->tbl);
5211         else lvtbl->tbl[0] = lvtbl->cnt;
5212     }
5213     ruby_dyna_vars = lvtbl->dyna_vars;
5214     free(lvtbl);
5215     lvtbl = local;
5216 }
```

(parse.y)

I'd like you to look at `local_tbl()`. This is the function to obtain the current local variable table (`lvtbl->tbl`). By calling this, the `nofree` of the current table becomes true. The meaning of `nofree` seems naturally “Don't free()”. In other words, this is like reference counting, “this table will be used, so please don't free()”. Conversely speaking, when `local_tbl()` was not called with a table even once, that table will be freed at the moment when being popped and be discarded. For example, this situation probably

happens when a method without any local variables.

However, the “necessary table” here means `lvtbl->tbl`. As you can see, `lvtbl` itself will be freed at the same moment when being popped. It means only the generated `lvtbl->tbl` is used in the evaluator. Then, the structure of `lvtbl->tbl` is becoming important. Let’s look at the function `local_cnt()` (which seems) to add variables which is probably helpful to understand how the structure is.

And before that, I’d like you to remember that `lvtbl->cnt` is stored at the index 0 of the `lvtbl->tbl`.

## ■ Adding variables

The function (which seems) to add a local variable is `local_cnt()`.

### ▼ `local_cnt()`

```
5246 static int
5247 local_cnt(id)
5248     ID id;
5249 {
5250     int cnt, max;
5251
5252     if (id == 0) return lvtbl->cnt;
5253
5254     for (cnt=1, max=lvtbl->cnt+1; cnt<max; cnt++) {
5255         if (lvtbl->tbl[cnt] == id) return cnt-1;
5256     }
5257     return local_append(id);
5258 }
```

This scans `lvtbl->tbl` and searches what is equals to `id`. If the searched one is found, it straightforwardly returns `cnt-1`. If nothing is found, it does `local_append()`. `local_append()` must be, as it is called `append`, the procedure to append. In other words, `local_cnt()` checks if the variable was already registered, if it was not, adds it by using `local_append()` and returns it.

What is the meaning of the return value of this function? `lvtbl->tbl` seems an array of the variables, so there're one-to-one correspondences between the variable names and “their index – 1 (`cnt-1`)”. (Fig.10)

0	1	2	3	4	return values of local_cnt()
??	id	id	id	id	
0	1	2	3	4	array indexes

Fig.10: The correspondences between the variable names and the return values

Moreover, this return value is calculated so that the start point becomes 0, the local variable space is probably an array. And, this returns the index to access that array. If it is not, like the instance variables or constants, (the ID of) the variable name could have been used as a key in the first place.

You might want to know why it is avoiding index 0 (the loop start from `cnt=1`) for some reasons, it is probably to store a value at

local\_pop().

Based on the knowledge we've learned, we can understand the role of `local_append()` without actually looking at the content. It registers a local variable and returns “(the index of the variable in `lvtbl->tbl`) – 1”. It is shown below, let's make sure.

### ▼ local\_append()

```
5225 static int
5226 local_append(id)
5227     ID id;
5228 {
5229     if (lvtbl->tbl == 0) {
5230         lvtbl->tbl = ALLOC_N(ID, 4);
5231         lvtbl->tbl[0] = 0;
5232         lvtbl->tbl[1] = '_';
5233         lvtbl->tbl[2] = '~';
5234         lvtbl->cnt = 2;
5235         if (id == '_') return 0;
5236         if (id == '~') return 1;
5237     }
5238     else {
5239         REALLOC_N(lvtbl->tbl, ID, lvtbl->cnt+2);
5240     }
5241
5242     lvtbl->tbl[lvtbl->cnt+1] = id;
5243     return lvtbl->cnt++;
5244 }
```

(parse.y)

It seems definitely true. `lvtbl->tbl` is an array of the local variable names, and its index – 1 is the return value (local variable ID).

Note that it increases `lvtbl->cnt`. Since the code to increase `lvtbl-`

`>cnt` only exists here, from only this code its meaning can be decided. Then, what is the meaning? It is, since “`lvtbl->cnt` increases by 1 when a new variable is added”, “`lvtbl->cnt` holds the number of local variables in this scope”.

Finally, I'll explain about `tbl[1]` and `tbl[2]`. These '`_`' and '`~`' are, as you can guess if you are familiar with Ruby, the special variables named `$_` and `$~`. Though their appearances are identical to global variables, they are actually local variables. Even If you didn't explicitly use it, when the methods such as `Kernel#gets` are called, these variables are implicitly assigned, thus it's necessary that the spaces are always allocated.

## ■ Summary of local variables

Since the description of local variables were complex in various ways, let's summarize it.

First, It seems the local variables are different from the other variables because they are not managed with `st_table`. Then, where are they stored in? It seems the answer is an array. Moreover, it is stored in a different array for each scope.

The array is `lvtbl->tbl`, and the index `0` holds the `lvtbl->cnt` which is set at `local_pop()`. In other words, it holds the number of the local variables. The index `1` or more hold the local variable names defined in the scope. Fig.11 shows the final appearance we expect.

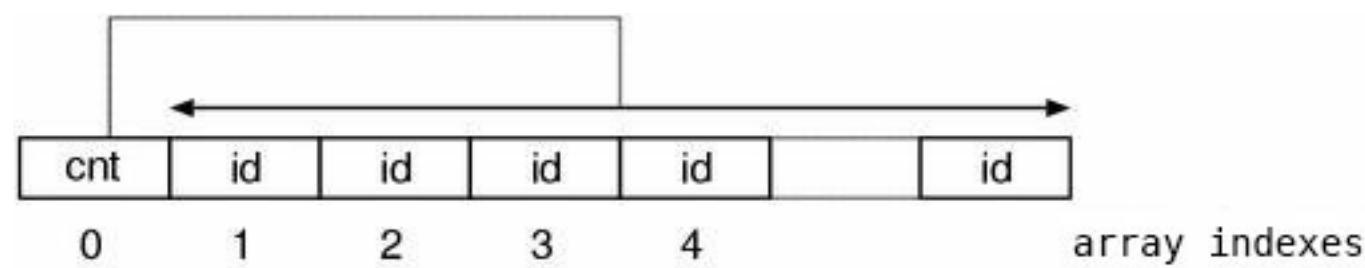


Fig.11: correspondences between local variable names and the return values

## Block Local Variables

The rest is `dyna_vars` which is a member of `struct local_vars`. In other words, this is about the block local variables. I thought that there must be the functions to do something with this, looked over the list of the function names, and found them as expected. There are the suspicious functions named `dyna_push()` `dyna_pop()` `dyna_in_block()`. Moreover, here is the place where these are used.

▼ an example using `dyna_push` `dyna_pop`

```

1651 brace_block      : '{'
1652           {
1653           $<vars>$ = dyna_push();
1654           }
1655           opt_block_var
1656           compstmt '}'
1657           {
1658           $$ = NEW_ITER($3, 0, $4);
1659           fixpos($$, $4);
1660           dyna_pop($<vars>2);
1661           }

```

(parse.y)

push at the beginning of an iterator block, pop at the end. This must

be the process of block local variables.

Now, we are going to look at the functions.

### ▼ dyna\_push()

```
5331 static struct RVarmap*
5332 dyna_push()
5333 {
5334     struct RVarmap* vars = ruby_dyna_vars;
5335
5336     rb_dvar_push(0, 0);
5337     lvtbl->dlev++;
5338     return vars;
5339 }
```

(parse.y)

Increasing `lvtbl->dlev` seems the mark indicates the existence of the block local variable scope. Meanwhile, `rb_dvar_push()` is ...

### ▼ rb\_dvar\_push()

```
691 void
692 rb_dvar_push(id, value)
693     ID id;
694     VALUE value;
695 {
696     ruby_dyna_vars = new_dvar(id, value, ruby_dyna_vars);
697 }
```

(eval.c)

It creates a `struct RVarmap` that has the variable name `id` and the value `val` as its members, adds it to the top of the global variable

`ruby_dyna_vars`. This is again and again the form of cons. In `dyna_push()`, `ruby_dyan_vars` is not set aside, it seems it adds directly to the `ruby_dyna_vars` of the previous scope.

Moreover, the value of the `id` member of the `RVarmap` to be added here is 0. Although it was not seriously discussed in this book, the `ID` of `ruby` will never be 0 while it is normally created by `rb_intern()`. Thus, we can infer that this `RVarmap`, as it is like `NUL` or `NULL`, probably has a role as sentinel. If we think based on this assumption, we can describe the reason why the holder of a variable (`RVarmap`) is added even though not any variables are added.

Next, `dyna_pop()`.

### ▼ `dyna_pop()`

```
5341 static void
5342 dyna_pop(vars)
5343     struct RVarmap* vars;
5344 {
5345     lvtbl->dlev--;
5346     ruby_dyna_vars = vars;
5347 }
```

(`parse.y`)

By reducing `lvtbl->dlev`, it writes down the fact that the block local variable scope ended. It seems that something is done by using the argument, let's see this later at once.

The place to add a block local variable has not appeared yet.

Something like `local_cnt()` of local variables is missing. So, I did plenty of grep with `dvar` and `dyna`, and this code was found.

## ▼ `assignable()` (partial)

```
4599 static NODE*
4600 assignable(id, val)
4601     ID id;
4602     NODE *val;
4603 {
4604     :
4634         rb_dvar_push(id, Qnil);
4635     return NEW_DASGN_CURR(id, val);

```

(`parse.y`)

`assignable()` is the function to create a node relates to assignments, this citation is the fragment of that function only contains the part to deal with block local variables. It seems that it adds a new variable (to `ruby_dyna_vars`) by using `rb_dvar_push()` that we've just seen.

## █ `ruby_dyna_vars` in the parser

Now, taking the above all into considerations, let's imagine the appearance of `ruby_dyna_vars` at the moment when a local variable scope is finished to be parsed.

First, as I said previously, the `RVarmap` of `id=0` which is added at the beginning of a block scope is a sentinel which represents a break between two block scopes. We'll call this “the header of

`ruby_dyna_vars`".

Next, among the previously shown actions of the rule of the iterator block, I'd like you to focus on this part:

```
$<vars>$ = dyna_push();      /* what assigned into $<vars>$ is ...
:
:
dyna_pop($<vars>2);          /* ..... appears at $<vars>2 */
```

`dyna_push()` returns the `ruby_dyna_vars` at the moment. `dyna_pop()` put the argument into `ruby_dyna_vars`. This means `ruby_dyna_vars` would be saved and restored for each the block local variable scope. Therefore, when parsing the following program,

```
iter {
  a = nil
  iter {
    b = nil
    iter {
      c = nil
      # nesting level 3
    }
    bb = nil
    # nesting level 2
    iter {
      e = nil
    }
  }
  # nesting level 1
}
```

Fig.12 shows the `ruby_dyna_vars` in this situation.

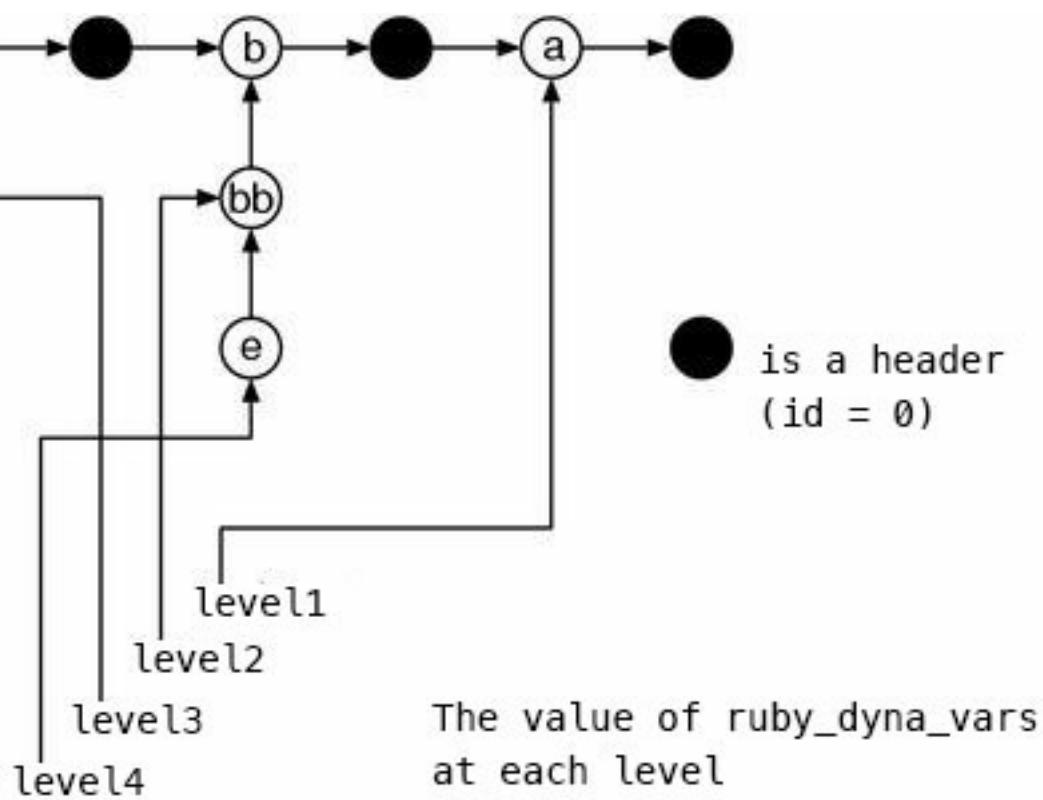


Fig.12: ruby\_dyna\_vars when all scopes are finished to be parsed

This structure is fairly smart. That's because the variables of the higher levels can naturally be accessed by traversing over all of the list even if the nesting level is deep. This way has the simpler searching process than creating a different table for each level.

Plus, in the figure, it looks like bb is hung at a strange place, but this is correct. When a variable is found at the nest level which is decreased after increased once, it is attached to the subsequent of the list of the original level. Moreover, in this way, the specification of local variable that “only the variables which already exist in the symbol sequence are defined” is expressed in a natural form.

And finally, at each cut of local variable scopes (this is not of block local variable scopes), this link is entirely saved or restored to `lvtbl->dyna_vars`. I'd like you to go back a little and check

`local_push()` and `local_pop()`.

By the way, although creating the `ruby_dyna_vars` list was a huge task, it is by itself not used at the evaluator. This list is used only to check the existence of the variables and will be garbage collected at the same moment when parsing is finished. And after entering the evaluator, another chain is created again. There's a quite deep reason for this, ... we'll see around this once again in Part 3.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

# Chapter 13: Structure of the evaluator

## Outline

---

### ■ Interface

We are not familiar with the word “Hyo-ka-ki” (evaluator). Literally, it must be a “-ki” (device) to “hyo-ka” (evaluating). Then, what is “hyo-ka”?

“Hyo-ka” is the definitive translation of “evaluate”. However, if the premise is describing about programming languages, it can be considered as an error in translation. It’s hard to avoid that the word “hyo-ka” gives the impression of “whether it is good or bad”.

“Evaluate” in the context of programming languages has nothing to do with “good or bad”, and its meaning is more close to “speculating” or “executing”. The origin of “evaluate” is a Latin word “ex+value+ate”. If I translate it directly, it is “turn it into a value”. This may be the simplest way to understand: to determine the value from an expression expressed in text.

Very frankly speaking, the bottom line is that evaluating is executing a written expression and getting the result of it. Then why is it not called just “execute”? It’s because evaluating is not only executing.

For example, in an ordinary programming language, when we write “3”, it will be dealt with as an integer 3. This situation is sometimes described as “the result of evaluating “3” is 3”. It’s hard to say an expression of a constant is executed, but it is certainly an evaluation. It’s all right if there exist a programming language in which the letter “3”, when it is evaluated, will be dealt with (evaluated) as an integer 6.

I’ll introduce another example. When an expression consists of multiple constants, sometimes the constants are calculated during the compiling process (constant folding). We usually don’t call it “executing” because executing indicates the process that the created binary is working. However, no matter when it is calculated you’ll get the same result from the same program.

In other words, “evaluating” is usually equals to “executing”, but essentially “evaluating” is different from “executing”. For now, only this point is what I’d like you to remember.

## **The characteristics of ruby's evaluator.**

The biggest characteristic of ruby’s evaluator is that, as this is also of the whole ruby’s interpreter, the difference in expressions

between the C-level code (extension libraries) and the Ruby-level code is small. In ordinary programming languages, the amount of the features of its interpreter we can use from extension libraries is usually very limited, but there are awfully few limits in ruby. Defining classes, defining methods and calling a method without limitation, these can be taken for granted. We can also use exception handling, iterators. Furthermore, threads.

But we have to compensate for the conveniences somewhere. Some codes are weirdly hard to implement, some codes have a lot overhead, and there are a lot of places implementing the almost same thing twice both for C and Ruby.

Additionally, ruby is a dynamic language, it means that you can construct and evaluate a string at runtime. That is `eval` which is a function-like method. As you expected, it is named after “evaluate”. By using it, you can even do something like this:

```
lvar = 1
answer = eval("lvar + lvar")      # the answer is 2
```

There are also `Module#module_eval` and `Object#instance_eval`, each method behaves slightly differently. I'll describe about them in detail in Chapter 17: Dynamic evaluation.

## eval.c

The evaluator is implemented in `eval.c`. However, this `eval.c` is a really huge file: it has 9000 lines, its size is 200K bytes, and the

number of the functions in it is 309. It is hard to fight against. When the size becomes this amount, it's impossible to figure out its structure by just looking over it.

So how can we do? First, the bigger the file, the less possibility of its content not separated at all. In other words, the inside of it must be modularized into small portions. Then, how can we find the modules? I'll list up some ways.

The first way is to print the list of the defined functions and look at the prefixes of them. `rb_dvar_`, `rb_mod_`, `rb_thread` — there are plenty of functions with these prefixes. Each prefix clearly indicate a group of the same type of functions.

Alternatively, as we can tell when looking at the code of the class libraries, `Init_xxxx()` is always put at the end of a block in `ruby`. Therefore, `Init_xxxx()` also indicates a break between modules.

Additionally, the names are obviously important, too. Since `eval()` and `rb_eval()` and `eval_node()` appear close to each other, we naturally think there should be a deep relationship among them.

Finally, in the source code of `ruby`, the definitions of types or variables and the declarations of prototypes often indicate a break between modules.

Being aware of these points when looking, it seems that `eval.c` can be mainly divided into these modules listed below:

Safe Level	already explained in Chapter 7: Security
Method Entry Manipulations	finding or deleting syntax trees which are actual method bodies
Evaluator Core	the heart of the evaluator that <code>rb_eval()</code> is at its center.
Exception	generations of exceptions and creations of backtraces
Method	the implementation of method call
Iterator	the implementation of functions that are related to blocks
Load	loading and evaluating external files
Proc	the implementation of Proc
Thread	the implementation of Ruby threads

Among them, “Load” and “Thread” are the parts that essentially should not be in `eval.c`. They are in `eval.c` merely because of the restrictions of C language. To put it more precisely, they need the macros such as `PUSH_TAG` defined in `eval.c`. So, I decided to exclude the two topics from Part 3 and deal with them at Part 4. And, it’s probably all right if I don’t explain the safe level here because I’ve already done in Part 1.

Excluding the above three, the six items are left to be described. The below table shows the corresponding chapter of each of them:

Method Entry Manipulations	the next chapter: Context
Evaluator Core	the entire part of Part 3
Exception	this chapter
Method	Chapter 15: Methods
Iterator	Chapter 16: Blocks

## From main by way of ruby\_run to rb\_eval

### Call Graph

The true core of the evaluator is a function called `rb_eval()`. In this chapter, we will follow the path from `main()` to that `rb_eval()`. First of all, here is a rough call graph around `rb_eval`:

```
main                                ....main.c
  ruby_init                          ....eval.c
    ruby_prog_init                   ....ruby.c
  ruby_options                       ....eval.c
    ruby_process_options            ....ruby.c
  ruby_run                           ....eval.c
    eval_node
      rb_eval
      *
    ruby_stop
```

I put the file names on the right side when moving to another file. Gazing this carefully, the first thing we'll notice is that the functions of `eval.c` call the functions of `ruby.c` back.

I wrote it as “calling back” because `main.c` and `ruby.c` are relatively for the implementation of `ruby` command. `eval.c` is the implementation of the evaluator itself which keeps a little distance from `ruby` command. In other words, `eval.c` is supposed to be used by `ruby.c` and calling the functions of `ruby.c` from `eval.c` makes `eval.c` less independent.

Then, why is this in this way? It's mainly because of the restrictions of C language. Because the functions such as `ruby_prog_init()` and `ruby_process_options()` start to use the API of the ruby world, it's possible an exception occurs. However, in order to stop an exception of Ruby, it's necessary to use the macro named `PUSH_TAG()` which can only be used in `eval.c`. In other words, essentially, `ruby_init()` and `ruby_run()` should have been defined in `ruby.c`.

Then, why isn't `PUSH_TAG` an `extern` function or something which is available to other files? Actually, `PUSH_TAG` can only be used as a pair with `POP_TAG` as follows:

```
PUSH_TAG();  
/* do lots of things */  
POP_TAG();
```

Because of its implementation, the two macros should be put into the same function. It's possible to implement in a way to be able to divide them into different functions, but not in such way because it's slower.

The next thing we notice is, the fact that it sequentially calls the functions named `ruby_xxxx` from `main()` seems very meaningful. Since they are really obviously symmetric, it's odd if there's not any relationship.

Actually, these three functions have deep relationships. Simply speaking, all of these three are "built-in Ruby interfaces". That is,

they are used only when creating a command with built-in ruby interpreter and not when writing extension libraries. Since ruby command itself can be considered as one of programs with built-in Ruby in theory, to use these interfaces is natural.

What is the `ruby_` prefix ? So far, the all of `ruby` 's functions are prefixed with `rb_`. Why are there the two types: `rb_` and `ruby_`? I investigated but could not understand the difference, so I asked directly. The answer was, “`ruby_` is for the auxiliary functions of `ruby` command and `rb_` is for the official interfaces”

“Then, why are the variables like `ruby_scope` are `ruby_`?”, I asked further. It seems this is just a coincidence. The variables like `ruby_scope` are originally named as `the_xxxx`, but in the middle of the version 1.3 there's a change to add prefixes to all interfaces. At that time `ruby_` was added to the “may-be-internals-for-some-reasons” variables.

The bottom line is that `ruby_` is attached to things that support `ruby` command or the internal variables and `rb_` is attached to the official interfaces of ruby interpreter.

## █ `main()`

First, straightforwardly, I'll start with `main()`. It is nice that this is very short.

## ▼ `main()`

```
36 int
37 main(argc, argv, envp)
38     int argc;
39     char **argv, **envp;
40 {
41 #if defined(NT)
42     NtInitialize(&argc, &argv);
43 #endif
44 #if defined(__MACOS__) && defined(__MWERKS__)
45     argc = ccommand(&argv);
46 #endif
47
48     ruby_init();
49     ruby_options(argc, argv);
50     ruby_run();
51     return 0;
52 }
```

(main.c)

#if def NT is obviously the NT of Windows NT. But somehow NT is also defined in Win9x. So, it means Win32 environment. NtInitialize() initializes argc argv and the socket system (WinSock) for Win32. Because this function is only doing the initialization, it's not interesting and not related to the main topic. Thus, I omit this.

And, \_\_MACOS\_\_ is not “Ma-Ko-Su” but Mac OS. In this case, it means Mac OS 9 and before, and it does not include Mac OS X. Even though such #ifdef remains, as I wrote at the beginning of this book, the current version can not run on Mac OS 9 and before. It's just a legacy from when ruby was able to run on it. Therefore, I also omit this code.

By the way, as it is probably known by the readers who are familiar with C language, the identifiers starting with an under bar are reserved for the system libraries or OS. However, although they are called “reserved”, using it is almost never result in an error, but if using a little weird `cc` it could result in an error. For example, it is the `cc` of HP-US. HP-US is an UNIX which `HP` is creating. If there’s any opinion such as HP-UX is not weird, I would deny it out loud.

Anyway, conventionally, we don’t define such identifiers in user applications.

Now, I’ll start to briefly explain about the built-in Ruby interfaces.

## ■ `ruby_init()`

`ruby_init()` initializes the Ruby interpreter. Since only a single interpreter of the current Ruby can exist in a process, it does not need neither arguments or a return value. This point is generally considered as “lack of features”.

When there’s only a single interpreter, more than anything, things around the development environment should be especially troublesome. Namely, the applications such as `irb`, RubyWin, and RDE. Although loading a rewritten program, the classes which are supposed to be deleted would remain. To counter this with the reflection API is not impossible but requires a lot of efforts.

However, it seems that Mr. Matsumoto (Matz) purposefully limits the number of interpreters to one. “it’s impossible to initialize

completely” seems its reason. For instance, “the loaded extension libraries could not be removed” is taken as an example.

The code of `ruby_init()` is omitted because it’s unnecessary to read.

## ■ `ruby_options()`

What to parse command-line options for the Ruby interpreter is `ruby_options()`. Of course, depending on the command, we do not have to use this.

Inside this function, `-r` (load a library) and `-e` (pass a program from command-line) are processed. This is also where the file passed as a command-line argument is parsed as a Ruby program.

`ruby` command reads the main program from a file if it was given, otherwise from `stdin`. After that, using `rb_compile_string()` or `rb_compile_file()` introduced at Part 2, it compiles the text into a syntax tree. The result will be set into the global variable `ruby_eval_tree`.

I also omit the code of `ruby_options()` because it’s just doing necessary things one by one and not interesting.

## ■ `ruby_run()`

Finally, `ruby_run()` starts to evaluate the syntax tree which was set to `ruby_eval_tree`. We also don’t always need to call this function. Other than `ruby_run()`, for instance, we can evaluate a string by

using a function named `rb_eval_string()`.

## ▼ `ruby_run()`

```
1257 void
1258 ruby_run()
1259 {
1260     int state;
1261     static int ex;
1262     volatile NODE *tmp;
1263
1264     if (ruby_nerrs > 0) exit(ruby_nerrs);
1265
1266     Init_stack((void*)&tmp);
1267     PUSH_TAG(PROT_NONE);
1268     PUSH_ITER(ITER_NOT);
1269     if ((state = EXEC_TAG()) == 0) {
1270         eval_node(ruby_top_self, ruby_eval_tree);
1271     }
1272     POP_ITER();
1273     POP_TAG();
1274
1275     if (state && !ex) ex = state;
1276     ruby_stop(ex);
1277 }
```

`(eval.c)`

We can see the macros `PUSH_xxxx()`, but we can ignore them for now. I'll explain about around them later when the time comes. The important thing here is only `eval_node()`. Its content is:

## ▼ `eval_node()`

```
1112 static VALUE
1113 eval_node(self, node)
1114     VALUE self;
```

```
1115     NODE *node;
1116 {
1117     NODE *beg_tree = ruby_eval_tree_begin;
1118
1119     ruby_eval_tree_begin = 0;
1120     if (beg_tree) {
1121         rb_eval(self, beg_tree);
1122     }
1123
1124     if (!node) return Qnil;
1125     return rb_eval(self, node);
1126 }
```

(eval.c)

This calls `rb_eval()` on `ruby_eval_tree`. The `ruby_eval_tree_begin` is storing the statements registered by `BEGIN`. But, this is also not important.

And, `ruby_stop()` inside of `ruby_run()` terminates all threads and finalizes all objects and checks exceptions and, in the end, calls `exit()`. This is also not important, so we won't see this.

## rb\_eval()

### ■ Outline

Now, `rb_eval()`. This function is exactly the real core of `ruby`. One `rb_eval()` call processes a single `NODE`, and the whole syntax tree will be processed by calling recursively. (Fig.1)

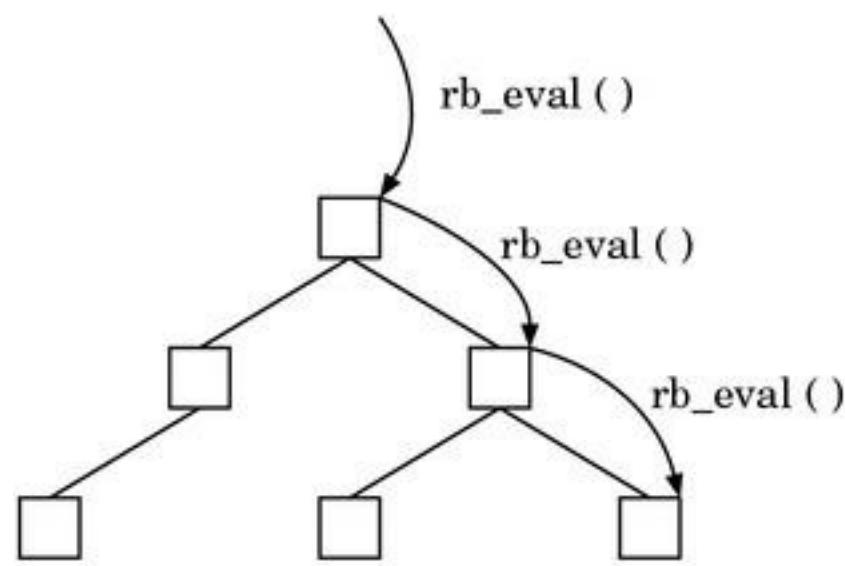


Fig.1: rb\_eval

rb\_eval is, as the same as yylex(), made of a huge switch statement and branching by each type of the nodes. First, let's look at the outline.

## ▼ rb\_eval() Outline

```

2221 static VALUE
2222 rb_eval(self, n)
2223     VALUE self;
2224     NODE *n;
2225 {
2226     NODE *nodesave = ruby_current_node;
2227     NODE * volatile node = n;
2228     int state;
2229     volatile VALUE result = Qnil;
2230
2231 #define RETURN(v) do { \
2232     result = (v);          \
2233     goto finish;          \
2234 } while (0)
2235
2236 again:
2237     if (!node) RETURN(Qnil);
2238
2239     ruby_last_node = ruby_current_node = node;
2240     switch (nd_type(node)) {
  
```

```
case NODE_BLOCK:  
    ....  
case NODE_POSTEXE:  
    ....  
case NODE_BEGIN:  
    :  
    (plenty of case statements)  
    :  
3415    default:  
3416        rb_bug("unknown node type %d", nd_type(node));  
3417    }  
3418    finish:  
3419        CHECK_INTS;  
3420        ruby_current_node = nodesave;  
3421        return result;  
3422 }
```

(eval.c)

In the omitted part, plenty of the codes to process all nodes are listed. By branching like this, it processes each node. When the code is only a few, it will be processed in `rb_eval()`. But when it becomes many, it will be a separated function. Most of functions in `eval.c` are created in this way.

When returning a value from `rb_eval()`, it uses the macro `RETURN()` instead of `return`, in order to always pass through `CHECK_INTS`. Since this macro is related to threads, you can ignore this until the chapter about it.

And finally, the local variables `result` and `node` are `volatile` for GC.

## ■ NODE\_IF

Now, taking the `if` statement as an example, let's look at the

process of the `rb_eval()` evaluation concretely. From here, in the description of `rb_eval()`,

- The source code (a Ruby program)
- Its corresponding syntax tree
- The partial code of `rb_eval()` to process the node.

these three will be listed at the beginning.

## ▼ source program

```
if true
  'true expr'
else
  'false expr'
end
```

## ▼ its corresponding syntax tree ( `nodedump` )

```
NODE_NEWLINE
nd_file = "if"
nd_nth  = 1
nd_next:
  NODE_IF
  nd_cond:
    NODE_TRUE
  nd_body:
    NODE_NEWLINE
    nd_file = "if"
    nd_nth  = 2
    nd_next:
      NODE_STR
      nd_lit = "true expr":String
  nd_else:
    NODE_NEWLINE
    nd_file = "if"
```

```
nd_nth  = 4
nd_next:
  NODE_STR
  nd_lit = "false expr":String
```

As we've seen in Part 2, `elsif` and `unless` can be, by contriving the ways to assemble, bundled to a single `NODE_IF` type, so we don't have to treat them specially.

### ▼ `rb_eval()` – `NODE_IF`

```
2324 case NODE_IF:
2325   if (trace_func) {
2326     call_trace_func("line", node, self,
2327                     ruby_frame->last_func,
2328                     ruby_frame->last_class);
2329   }
2330   if (RTEST(rb_eval(self, node->nd_cond))) {
2331     node = node->nd_body;
2332   }
2333   else {
2334     node = node->nd_else;
2335   }
2336   goto again;
```

`(eval.c)`

Only the last `if` statement is important. If rewriting it without any change in its meaning, it becomes this:

```
if (RTEST(rb_eval(self, node->nd_cond))) {          (A)
  RETURN(rb_eval(self, node->nd_body));            (B)
}
else {
  RETURN(rb_eval(self, node->nd_else));          (C)
}
```

First, at (A), evaluating (the node of) the Ruby's condition statement and testing its value with `RTEST()`. I've mentioned that `RTEST()` is a macro to test whether or not a `VALUE` is true of Ruby. If that was true, evaluating the `then` side clause at (B). If false, evaluating the `else` side clause at (C).

In addition, I've mentioned that `if` statement of Ruby also has its own value, so it's necessary to return a value. Since the value of an `if` is the value of either the `then` side or the `else` side which is the one executed, returning it by using the macro `RETURN()`.

In the original list, it does not call `rb_eval()` recursively but just does `goto`. This is the "conversion from tail recursion to `goto`" which has also appeared in the previous chapter "Syntax tree construction".

## ■ NODE\_NEWLINE

Since there was `NODE_NEWLINE` at the node for a `if` statement, let's look at the code for it.

### ▼ `rb_eval()` – `NODE_NEWLINE`

```
3404 case NODE_NEWLINE:  
3405   ruby_sourcefile = node->nd_file;  
3406   ruby_sourceline = node->nd_nth;  
3407   if (trace_func) {  
3408     call_trace_func("line", node, self,  
3409                     ruby_frame->last_func,  
3410                     ruby_frame->last_class);  
3411   }
```

```
3412     node = node->nd_next;  
3413     goto again;
```

(eval.c)

There's nothing particularly difficult.

`call_trace_func()` has already appeared at `NODE_IF`. Here is a simple explanation of what kind of thing it is. This is a feature to trace a Ruby program from Ruby level. The debugger ( `debug.rb` ) and the tracer ( `tracer.rb` ) and the profiler ( `profile.rb` ) and `irb` (interactive `ruby` command) and more are using this feature.

By using the function-like method `set_trace_func` you can register a `Proc` object to trace, and that `Proc` object is stored into `trace_func`. If `trace_func` is not `0`, it means not `QFalse`, it will be considered as a `Proc` object and executed (at `call_trace_func()` ).

This `call_trace_func()` has nothing to do with the main topic and not so interesting as well. Therefore in this book, from now on, I'll completely ignore it. If you are interested in it, I'd like you to challenge after finishing the Chapter 16: Blocks.

## ■ Pseudo-local Variables

`NODE_IF` and such are interior nodes in a syntax tree. Let's look at the leaves, too.

▼ `rb_eval()` Ppseudo-Local Variable Nodes

```
2312 case NODE_SELF:  
2313     RETURN(self);  
2314  
2315 case NODE_NIL:  
2316     RETURN(Qnil);  
2317  
2318 case NODE_TRUE:  
2319     RETURN(Qtrue);  
2320  
2321 case NODE_FALSE:  
2322     RETURN(Qfalse);
```

(eval.c)

We've seen `self` as the argument of `rb_eval()`. I'd like you to make sure it by going back a little. The others are probably not needed to be explained.

## Jump Tag

Next, I'd like to explain `NODE WHILE` which is corresponding to `while`, but to implement `break` or `next` only with recursive calls of a function is difficult. Since `ruby` enables these syntaxes by using what named “jump tag”, I'll start with describing it first.

Simply put, “jump tag” is a wrapper of `setjmp()` and `longjmp()` which are library functions of C language. Do you know about `setjmp()`? This function has already appeared at `gc.c`, but it is used in very abnormal way there. `setjmp()` is usually used to jump over functions. I'll explain by taking the below code as an example. The entry point is `parent()`.

## ▼ `setjmp()` and `longjmp()`

```
jmp_buf buf;

void child2(void) {
    longjmp(buf, 34); /* go back straight to parent
                        the return value of setjmp becomes 34 */
    puts("This message will never be printed.");
}

void child1(void) {
    child2();
    puts("This message will never be printed.");
}

void parent(void) {
    int result;
    if ((result = setjmp(buf)) == 0) {
        /* normally returned from setjmp */
        child1();
    } else {
        /* returned from child2 via longjmp */
        printf("%d\n", result); /* shows 34 */
    }
}
```

First, when `setjmp()` is called at `parent()`, the executing state at the time is saved to the argument `buf`. To put it a little more directly, the address of the top of the machine stack and the CPU registers are saved. If the return value of `setjmp()` was 0, it means it normally returned from `setjmp()`, thus you can write the subsequent code as usual. This is the `if` side. Here, it calls `child1()`.

Next, the control moves to `child2()` and calls `longjmp`, then it can go back straight to the place where the argument `buf` was `setjmp` ed.

So in this case, it goes back to the `setjmp` at `parent()`. When coming back via `longjmp`, the return value of `setjmp` becomes the value of the second argument of `longjmp`, so the `else` side is executed. And, even if we pass 0 to `longjmp`, it will be forced to be another value. Thus it's fruitless.

Fig.2 shows the state of the machine stack. The ordinary functions return only once for each call. However, it's possible `setjmp()` returns twice. Is it helpful to grasp the concept if I say that it is something like `fork()`?

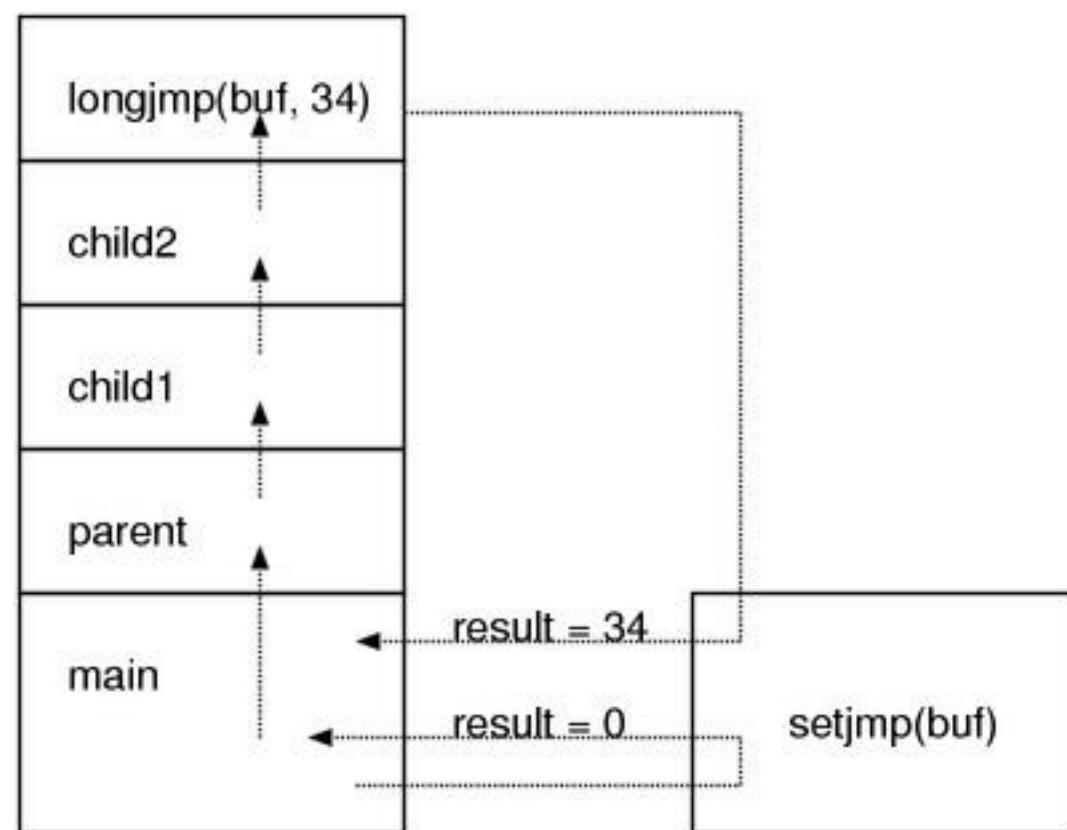


Fig.2: `setjmp()` `longjmp()` Image

Now, we've learned about `setjmp()` as a preparation. In `eval.c`, `EXEC_TAG` corresponds to `setjmp()` and `JUMP_TAG()` corresponds to `longjmp()` respectively. (Fig.3)

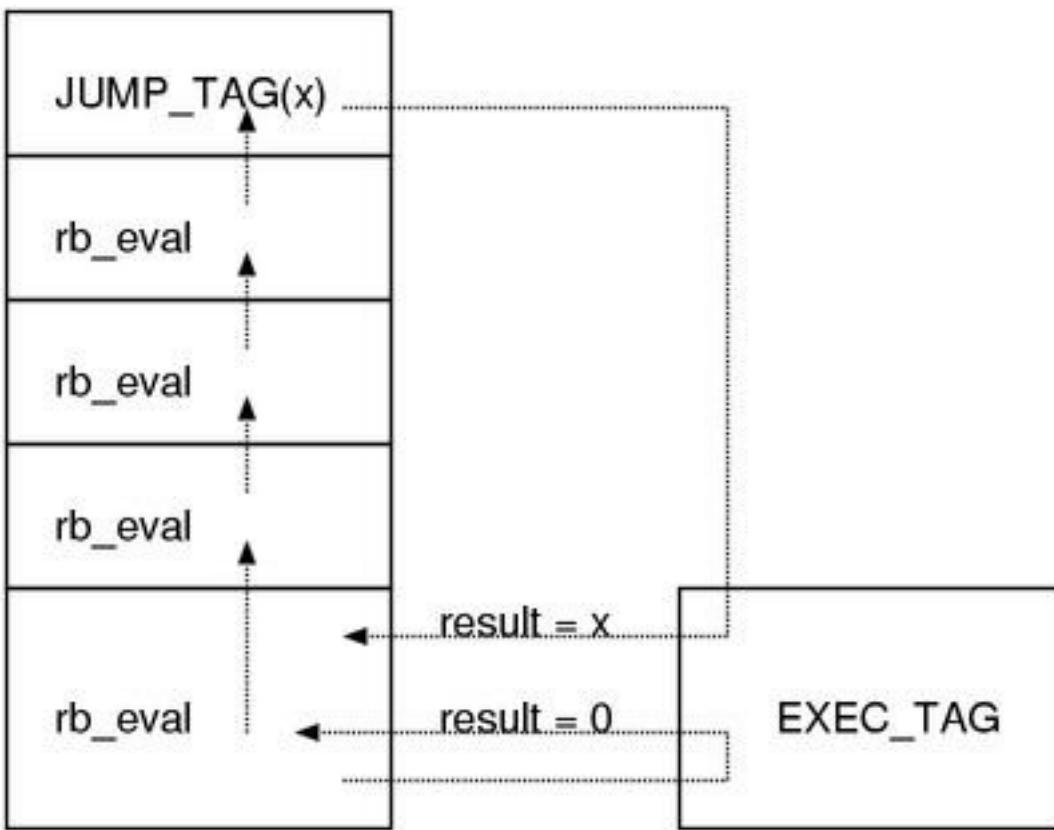


Fig.3: “tag jump” image

Take a look at this image, it seems that `EXEC_TAG()` does not have any arguments. Where has `jmp_buf` gone? Actually, in ruby, `jmp_buf` is wrapped by the struct `struct tag`. Let's look at it.

### ▼ `struct tag`

```

783 struct tag {
784     jmp_buf buf;
785     struct FRAME *frame;      /* FRAME when PUSH_TAG */
786     struct iter *iter;        /* ITER when PUSH_TAG */
787     ID tag;                  /* tag type */
788     VALUE retval;            /* the return value of this jump */
789     struct SCOPE *scope;     /* SCOPE when PUSH_TAG */
790     int dst;                 /* the destination ID */
791     struct tag *prev;
792 };

```

(eval.c)

Because there's the member `prev`, we can infer that `struct tag` is probably a stack structure using a linked list. Moreover, by looking around it, we can find the macros `PUSH_TAG()` and `POP_TAG`, thus it definitely seems a stack.

## ▼ `PUSH_TAG()` `POP_TAG()`

```
793 static struct tag *prot_tag; /* the pointer to the head of

795 #define PUSH_TAG(ptag) do { \
796     struct tag _tag; \
797     _tag.retval = Qnil; \
798     _tag.frame = ruby_frame; \
799     _tag.iter = ruby_iter; \
800     _tag.prev = prot_tag; \
801     _tag.scope = ruby_scope; \
802     _tag.tag = ptag; \
803     _tag.dst = 0; \
804     prot_tag = &_tag

818 #define POP_TAG() \
819     if (_tag.prev) \
820         _tag.prev->retval = _tag.retval; \
821     prot_tag = _tag.prev; \
822 } while (0)
```

(eval.c)

I'd like you to be flabbergasted here because the actual tag is fully allocated at the machine stack as a local variable. (Fig.4).

Moreover, `do ~ while` is divided between the two macros. This might be one of the most awful usages of the C preprocessor. Here is the macros `PUSH` / `POP` coupled and extracted to make it easy to read.

do {

```

struct tag _tag;
_tag.prev = prot_tag; /* save the previous tag */
prot_tag = &_tag; /* push a new tag on the stack */
/* do several things */
prot_tag = _tag.prev; /* restore the previous tag */
} while (0);

```

This method does not have any overhead of function calls, and its cost of the memory allocation is next to nothing. This technique is only possible because the ruby evaluator is made of recursive calls of `rb_eval()`.

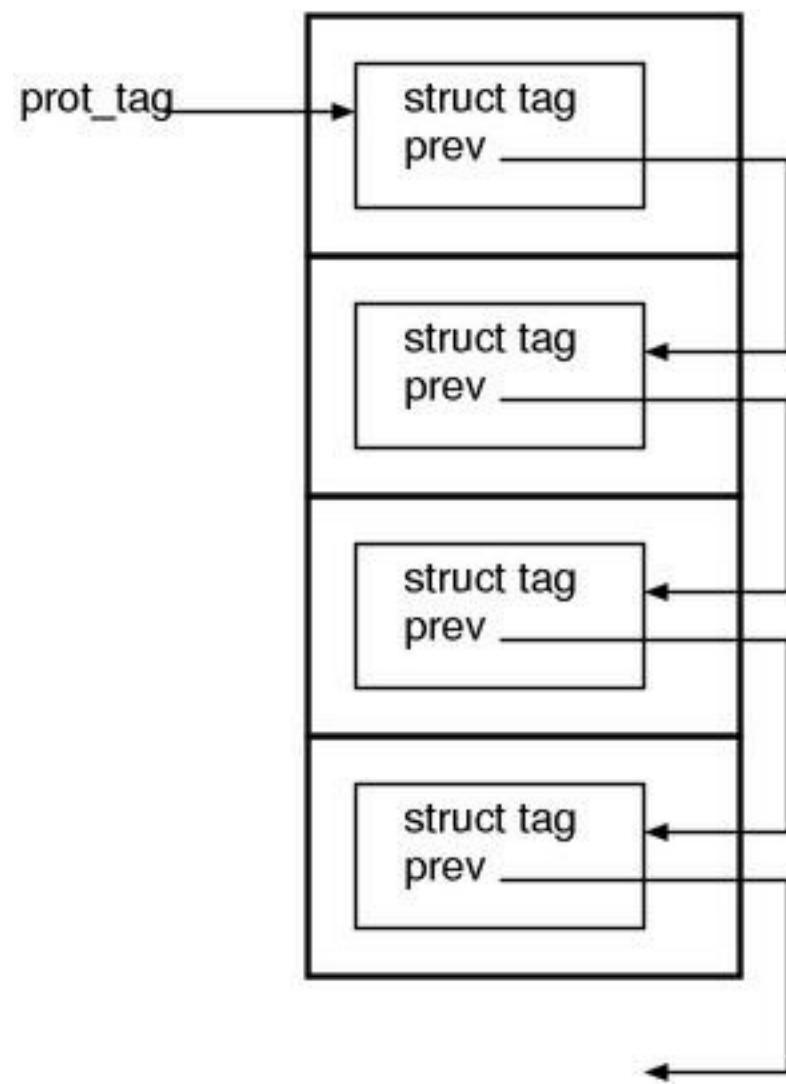


Fig.4: the tag stack is embedded in the machine stack

Because of this implementation, it's necessary that `PUSH_TAG` and

POP\_TAG are in the same one function as a pair. Plus, since it's not supposed to be carelessly used at the outside of the evaluator, we can't make them available to other files.

Additionally, let's also take a look at EXEC\_TAG() and JUMP\_TAG().

### ▼ EXEC\_TAG() JUMP\_TAG()

```
810 #define EXEC_TAG()      setjmp(prot_tag->buf)  
  
812 #define JUMP_TAG(st) do {  
813     ruby_frame = prot_tag->frame;          \  
814     ruby_iter = prot_tag->iter;            \  
815     longjmp(prot_tag->buf, (st));          \  
816 } while (0)  
  
(eval.c)
```

In this way, `setjmp` and `longjmp` are wrapped by `EXEC_TAG()` and `JUMP_TAG()` respectively. The name `EXEC_TAG()` can look like a wrapper of `longjmp()` at first sight, but this one is to execute `setjmp()`.

Based on all of the above, I'll explain the mechanism of `while`. First, when starting `while` it does `EXEC_TAG()` (`setjmp`). After that, it executes the main body by calling `rb_eval()` recursively. If there's `break` or `next`, it does `JUMP_TAG()` (`longjmp`). Then, it can go back to the start point of the `while` loop. (Fig.5)

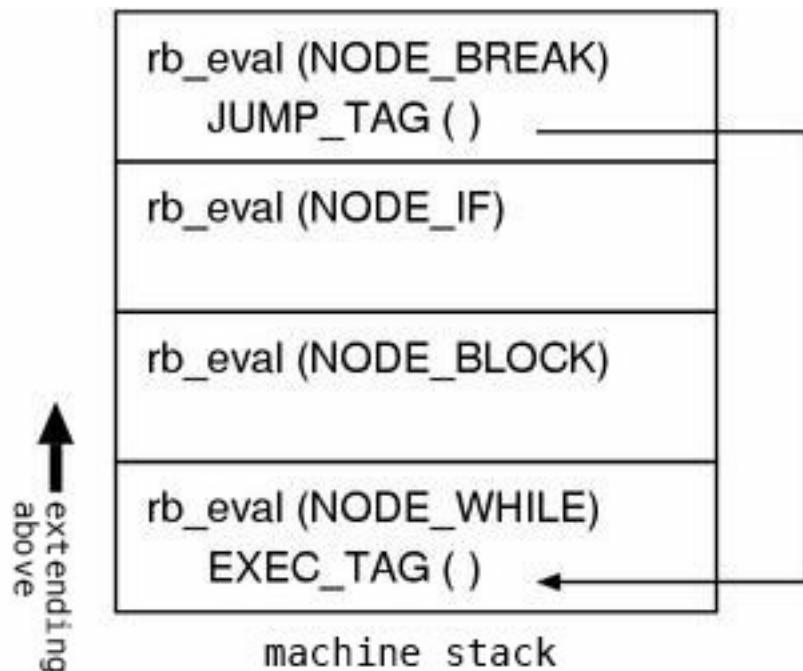


Fig.5: the implementation of `while` by using “tag jump”

Though `break` was taken as an example here, what cannot be implemented without jumping is not only `break`. Even if we limit the case to `while`, there are `next` and `redo`. Additionally, `return` from a method and exceptions also should have to climb over the wall of `rb_eval()`. And since it's cumbersome to use a different tag stack for each case, we want for only one stack to handle all cases in one way or another.

What we need to make it possible is just attaching information about “what the purpose of this jump is”. Conveniently, the return value of `setjmp()` could be specified as the argument of `longjmp()`, thus we can use this. The types are expressed by the following flags:

▼ tag type

```
829 #define TAG_BREAK          0x2  /* break */
830 #define TAG_NEXT           0x3  /* next */
831 #define TAG_RETRY          0x4  /* retry */
832 #define TAG_REDO           0x5  /* redo */
833 #define TAG_RAISE          0x6  /* general exceptions */
834 #define TAG_THROW          0x7  /* throw (won't be explained :(
835 #define TAG_FATAL          0x8  /* fatal : exceptions which a
836 #define TAG_MASK           0xf
```

(eval.c)

The meanings are written as each comment. The last `TAG_MASK` is the bitmask to take out these flags from a return value of `setjmp()`. This is because the return value of `setjmp()` can also include information which is not about a “type of jump”.

## ■ NODE WHILE

Now, by examining the code of `NODE WHILE`, let's check the actual usage of tags.

### ▼ The Source Program

```
while true
  'true_expr'
end
```

### ▼ Its corresponding syntax tree ( nodedump-short )

```
NODE WHILE
nd_state = 1 (while)
nd_cond:
  NODE_TRUE
nd_body:
```

```
NODE_STR
nd_lit = "true_expr":String
```

## ▼ rb\_eval – NODE WHILE

```
2418 case NODE WHILE:
2419     PUSH_TAG(prot_none);
2420     result = Qnil;
2421     switch (state = EXEC_TAG()) {
2422         case 0:
2423             if (node->nd_state && !RTEST(rb_eval(self, node->nd_co
2424                 goto while_out;
2425             do {
2426                 while_redo:
2427                     rb_eval(self, node->nd_body);
2428                 while_next:
2429                     ;
2430             } while (RTEST(rb_eval(self, node->nd_cond)));
2431             break;
2432
2433         case TAG_REDO:
2434             state = 0;
2435             goto while_redo;
2436         case TAG_NEXT:
2437             state = 0;
2438             goto while_next;
2439         case TAG_BREAK:
2440             state = 0;
2441             result = prot_tag->retval;
2442         default:
2443             break;
2444     }
2445     while_out:
2446     POP_TAG();
2447     if (state) JUMP_TAG(state);
2448     RETURN(result);
```

(eval.c)

The idiom which will appear over and over again appeared in the

```

PUSH_TAG(prot_none);
switch (state = EXEC_TAG()) {
    case 0:
        /* process normally */
        break;
    case TAG_a:
        state = 0;      /* clear state because the jump waited for con
        /* do the process of when jumped with TAG_a */
        break;
    case TAG_b:
        state = 0;      /* clear state because the jump waited for con
        /* do the process of when jumped with TAG_b */
        break;
    default
        break;          /* this jump is not waited for, then ... */
}
POP_TAG();
if (state) JUMP_TAG(state);  /* .. jump again here */

```

First, as `PUSH_TAG()` and `POP_TAG()` are the previously described mechanism, it's necessary to be used always as a pair. Also, they need to be written outside of `EXEC_TAG()`. And, apply `EXEC_TAG()` to the just pushed `jmp_buf`. This means doing `setjmp()`. If the return value is 0, since it means immediately returning from `setjmp()`, it does the normal processing (this usually contains `rb_eval()` ). If the return value of `EXEC_TAG()` is not 0, since it means returning via `longjmp()`, it filters only the own necessary jumps by using `case` and lets the rest ( `default` ) pass.

It might be helpful to see also the code of the jumping side. The below code is the handler of the node of `redo`.

## ▼ rb\_eval() – NODE\_REDO

```
2560 case NODE_REDO:  
2561     CHECK_INTS;  
2562     JUMP_TAG(TAG_REDO);  
2563     break;
```

(eval.c)

As a result of jumping via `JUMP_TAG()`, it goes back to the last `EXEC_TAG()`. The return value at the time is the argument `TAG_REDO`. Being aware of this, I'd like you to look at the code of `NODE_WHILE` and check what route is taken.

The idiom has enough explained, now I'll explain about the code of `NODE_WHILE` a little more in detail. As mentioned, since the inside of `case 0:` is the main process, I extracted only that part. Additionally, I moved some labels to enhance readability.

```
if (node->nd_state && !RTEST(rb_eval(self, node->nd_cond)))  
    goto while_out;  
do {  
    rb_eval(self, node->nd_body);  
} while (RTEST(rb_eval(self, node->nd_cond)));  
while_out:
```

There are the two places calling `rb_eval()` on `node->nd_state` which corresponds to the conditional statement. It seems that only the first test of the condition is separated. This is to deal with both `do ~ while` and `while` at once. When `node->nd_state` is `0` it is a `do ~ while`, when `1` it is an ordinary `while`. The rest might be understood by

following step-by-step, I won't particularly explain.

By the way, I feel like it easily becomes an infinite loop if there is `next` or `redo` in the condition statement. Since it is of course exactly what the code means, it's the fault of who wrote it, but I'm a little curious about it. So, I've actually tried it.

```
% ruby -e 'while next do nil end'  
-e:1: void value expression
```

It's simply rejected at the time of parsing. It's safe but not an interesting result. What produces this error is `value_expr()` of `parse.y`.

## ■ The value of an evaluation of `while`

`while` had not had its value for a long time, but it has been able to return a value by using `break` since `ruby 1.7`. This time, let's focus on the flow of the value of an evaluation. Keeping in mind that the value of the local variable `result` becomes the return value of `rb_eval()`, I'd like you to look at the following code:

```
result = Qnil;  
switch (state = EXEC_TAG()) {  
    case 0:  
        /* the main process */  
    case TAG_REDO:  
    case TAG_NEXT:  
        /* each jump */  
  
    case TAG_BREAK:  
        state = 0;
```

```

        result = prot_tag->retval;           (A)
    default:
        break;
    }
RETURN(result);

```

What we should focus on is only (A). The return value of the jump seems to be passed via `prot_tag->retval` which is a `struct tag`. Here is the passing side:

### ▼ `rb_eval()` – `NODE_BREAK`

```

2219 #define return_value(v) prot_tag->retval = (v)

2539 case NODE_BREAK:
2540     if (node->nd_stts) {
2541         return_value(alue_to_svalue(rb_eval(self, node->nd_s
2542     }
2543     else {
2544         return_value(Qnil);
2545     }
2546     JUMP_TAG(TAG_BREAK);
2547     break;

```

(eval.c)

In this way, by using the macro `return_value()`, it assigns the value to the struct of the top of the tag stack.

The basic flow is this, but in practice there could be another `EXEC_TAG` between `EXEC_TAG()` of `NODE WHILE` and `JUMP_TAG()` of `NODE_BREAK`. For example, rescue of an exception handling can exist between them.

```
while cond      # EXEC_TAG() for NODE_WHILE
  begin
    break 1
  rescue
  end
end
```

Therefore, it's hard to determine whether or not the `strict` tag of when doing `JUMP_TAG()` at `NODE_BREAK` is the one which was pushed at `NODE_WHILE`. In this case, because `retval` is propagated in `POP_TAG()` as shown below, the return value can be passed to the next tag without particular thought.

## ▼ `POP_TAG()`

```
818 #define POP_TAG()
819     if (_tag.prev)
820         _tag.prev->retval = _tag(retval); \
821     prot_tag = _tag.prev; \
822 } while (0)
```

`(eval.c)`

This can probably be depicted as Fig.6.

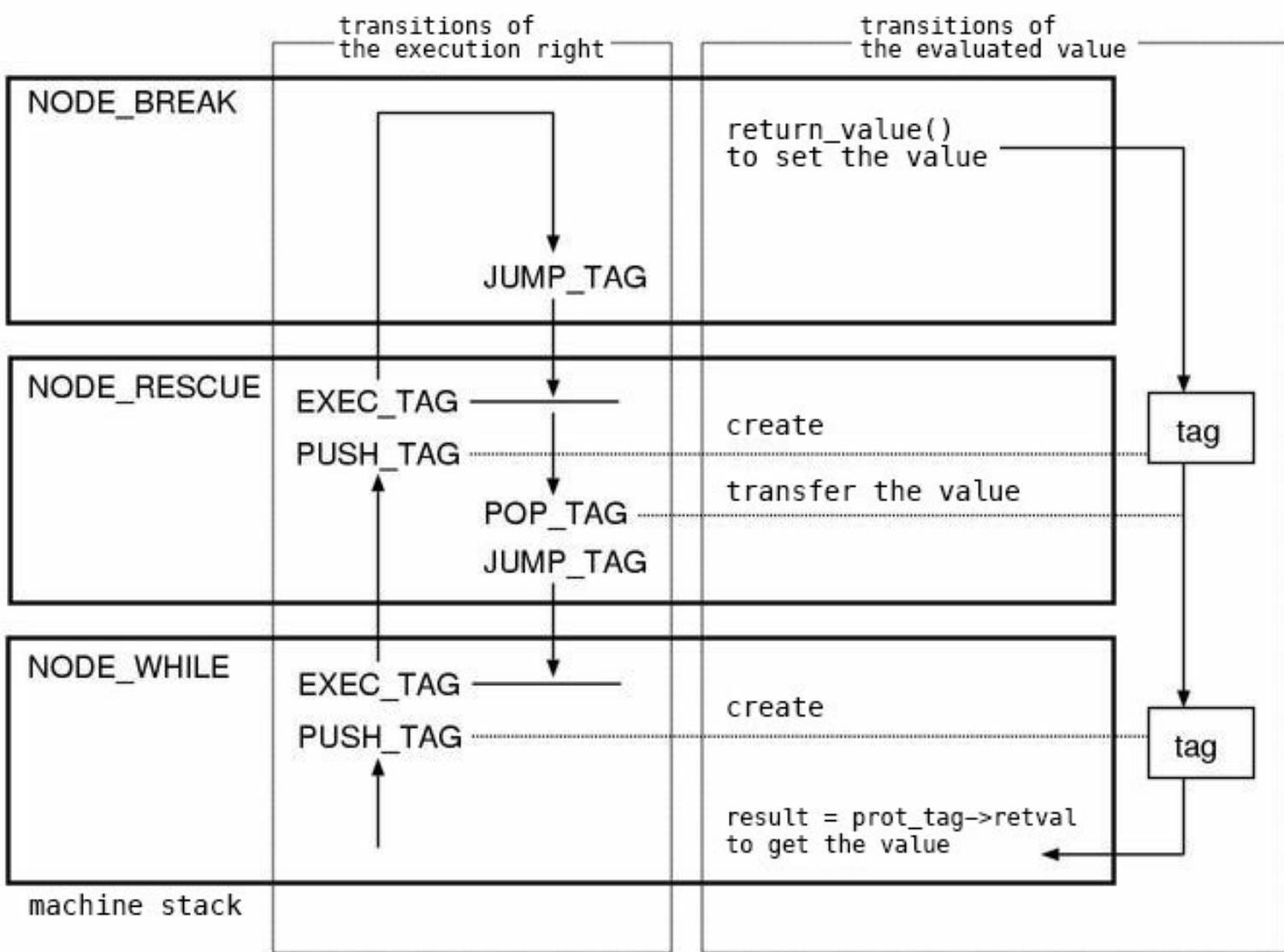


Fig.6: Transferring the return value

## Exception

As the second example of the usage of “tag jump”, we’ll look at how exceptions are dealt with.

raise

When I explained `while`, we looked at the `setjmp()` side first. This

time, we'll look at the `longjmp()` side first for a change. It's `rb_exc_raise()` which is the substance of `raise`.

## ▼ `rb_exc_raise()`

```
3645 void
3646 rb_exc_raise(VALUE mesg)
3647 {
3648     rb_longjmp(TAG_RAISE, mesg);
3650 }
```

`(eval.c)`

`mesg` is an exception object (an instance of `Exception` or one of its subclass). Notice that It seems to jump with `TAG_RAISE` this time. And the below code is very simplified `rb_longjmp()`.

## ▼ `rb_longjmp()` (simplified)

```
static void
rb_longjmp(tag, mesg)
    int tag;
    VALUE mesg;
{
    if (NIL_P(mesg))
        mesg = ruby_errinfo;
    set_backtrace(mesg, get_backtrace(mesg));
    ruby_errinfo = mesg;
    JUMP_TAG(tag);
}
```

Well, though this can be considered as a matter of course, this is just to jump as usual by using `JUMP_TAG()`.

What is `ruby_errinfo`? By doing grep a few times, I figured out that this variable is the substance of the global variable `$!` of Ruby. Since this variable indicates the exception which is currently occurring, naturally its substance `ruby_errinfo` should have the same meaning as well.

## ▀ The Big Picture

### ▼ the source program

```
begin
  raise('exception raised')
rescue
  'rescue clause'
ensure
  'ensure clause'
end
```

### ▼ the syntax tree ( `nodedump-short` )

```
NODE_BEGIN
nd_body:
  NODE_ENSURE
  nd_head:
    NODE_RESCUE
    nd_head:
      NODE_FCALL
      nd_mid = 3857 (raise)
      nd_args:
        NODE_ARRAY [
          0:
            NODE_STR
            nd_lit = "exception raised":String
        ]
nd_resq:
  NODE_RESBODY
```

```

    nd_args = (null)
    nd_body:
        NODE_STR
            nd_lit = "rescue clause":String
    nd_head = (null)
    nd_else = (null)
nd_ensr:
    NODE_STR
    nd_lit = "ensure clause":String

```

As the right order of `rescue` and `ensure` is decided at parser level, the right order is strictly decided at syntax tree as well. `NODE_ENSURE` is always at the “top”, `NODE_RESCUE` comes next, the main body (where `raise` exist) is the last. Since `NODE_BEGIN` is a node to do nothing, you can consider `NODE_ENSURE` is virtually on the top.

This means, since `NODE_ENSURE` and `NODE_RESCUE` are above the main body which we want to protect, we can stop `raise` by merely doing `EXEC_TAG()`. Or rather, the two nodes are put above in syntax tree for this purpose, is probably more accurate to say.

## ensure

We are going to look at the handler of `NODE_ENSURE` which is the node of `ensure`.

### ▼ rb\_eval() – NODE\_ENSURE

```

2634 case NODE_ENSURE:
2635     PUSH_TAG(PROT_NONE);
2636     if ((state = EXEC_TAG()) == 0) {
2637         result = rb_eval(self, node->nd_head);      (A-1)
2638     }

```

```
2639     POP_TAG();
2640     if (node->nd_ensr) {
2641         VALUE retval = prot_tag->retval;      (B-1)
2642         VALUE errinfo = ruby_errinfo;
2643
2644         rb_eval(self, node->nd_ensr);          (A-2)
2645         return_value(retval);                  (B-2)
2646         ruby_errinfo = errinfo;
2647     }
2648     if (state) JUMP_TAG(state);            (B-3)
2649     break;
```

(eval.c)

This branch using `if` is another idiom to deal with tag. It interrupts a jump by doing `EXEC_TAG()` then evaluates the `ensure` clause ( ( `node->nd_ensr` ). As for the flow of the process, it's probably straightforward.

Again, we'll try to think about the value of an evaluation. To check the specification first,

```
begin
  expr0
ensure
  expr1
end
```

for the above statement, the value of the whole `begin` will be the value of `expr0` regardless of whether or not `ensure` exists. This behavior is reflected to the code (A-1,2), so the value of the evaluation of an `ensure` clause is completely discarded.

At (B-1,3), it deals with the evaluated value of when a jump

occurred at the main body. I mentioned that the value of this case is stored in `prot_tag->retval`, so it saves the value to a local variable to prevent from being carelessly overwritten during the execution of the `ensure` clause (B-1). After the evaluation of the `ensure` clause, it restores the value by using `return_value()` (B-2). When any jump has not occurred, `state==0` in this case, `prot_tag->retval` is not used in the first place.

## rescue

It's been a little while, I'll show the syntax tree of `rescue` again just in case.

### ▼ Source Program

```
begin
  raise()
rescue ArgumentError, TypeError
  'error raised'
end
```

### ▼ Its Syntax Tree ( nodedump-short )

```
NODE_BEGIN
nd_body:
  NODE_RESCUE
  nd_head:
    NODE_FCALL
    nd_mid = 3857 (raise)
    nd_args = (null)
  nd_resq:
    NODE_RESBODY
    nd_args:
```

```

NODE_ARRAY [
0:
  NODE_CONST
  nd_vid  = 4733 (ArgumentError)
1:
  NODE_CONST
  nd_vid  = 4725 (TypeError)
]
nd_body:
  NODE_STR
  nd_lit = "error raised":String
  nd_head = (null)
  nd_else = (null)

```

I'd like you to make sure that (the syntax tree of) the statement to be rescued is “under” NODE\_RESCUE.

## ▼ rb\_eval() – NODE\_RESCUE

```

2590 case NODE_RESCUE:
2591   retry_entry:
2592   {
2593     volatile VALUE e_info = ruby_errinfo;
2594
2595     PUSH_TAG(prot_none);
2596     if ((state = EXEC_TAG()) == 0) {
2597       result = rb_eval(self, node->nd_head); /* evaluate
2598     }
2599     POP_TAG();
2600     if (state == TAG_RAISE) { /* an exception occurred at
2601       NODE * volatile resq = node->nd_resq;
2602
2603       while (resq) { /* deal with the rescue clause one
2604         ruby_current_node = resq;
2605         if (handle_rescue(self, resq)) { /* If dealt w
2606           state = 0;
2607           PUSH_TAG(prot_none);
2608           if ((state = EXEC_TAG()) == 0) {
2609             result = rb_eval(self, resq->nd_body);
2610           } /* evaluate t

```

```
2611             POP_TAG();
2612             if (state == TAG_RETRY) { /* Since retry or
2613                 state = 0;
2614                 ruby_errinfo = Qnil; /* the exception
2615                     goto retry_entry; /* convert to go
2616                 */
2617                 if (state != TAG_RAISE) { /* Also by rescue
2618                     ruby_errinfo = e_info; /* the exception
2619                 */
2620                     break;
2621                 }
2622                 resq = resq->nd_head; /* move on to the next rescue
2623             */
2624         }
2625         else if (node->nd_else) { /* when there is an else clause
2626             if (!state) { /* evaluate it only when any exception
2627                 result = rb_eval(self, node->nd_else);
2628             }
2629         }
2630         if (state) JUMP_TAG(state); /* the jump was not waited
2631     */
2632     break;

```

(eval.c)

Even though the size is not small, it's not difficult because it only simply deal with the nodes one by one. This is the first time `handle_rescue()` appeared, but for some reasons we cannot look at this function now. I'll explain only its effects here. Its prototype is this,

```
static int handle_rescue(VALUE self, NODE *resq)
```

and it determines whether the currently occurring exception (`ruby_errinfo`) is a subclass of the class that is expressed by `resq` (`TypeError`, for instance). The reason why passing `self` is that it's

necessary to call `rb_eval()` inside this function in order to evaluate  
resq.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

# Chapter 14: Context

The range covered by this chapter is really broad. First of all, I'll describe about how the internal state of the evaluator is expressed. After that, as an actual example, we'll read how the state is changed on a class definition statement. Subsequently, we'll examine how the internal state influences method definition statements. Lastly, we'll observe how the both statements change the behaviors of the variable definitions and the variable references.

## The Ruby stack

### Context and Stack

With an image of a typical procedural language, each time calling a procedure, the information which is necessary to execute the procedure such as the local variable space and the place to return is stored in a struct (a stack frame) and it is pushed on the stack.

When returning from a procedure, the struct which is on the top of the stack is popped and the state is returned to the previous method. The executing image of a C program which was explained at Chapter 5: Garbage collection is a perfect example.

What to be careful about here is, what is changing during the

execution is only the stack, on the contrary, the program remains unchanged wherever it is. For example, if it is “a reference to the local variable *i*”, there’s just an order of “give me *i* of the current frame”, it is not written as “give me *i* of that frame”. In other words, “only” the state of the stack influences the consequence. This is why, even if a procedure is called anytime and any number of times, we only have to write its code once (Fig. 1).

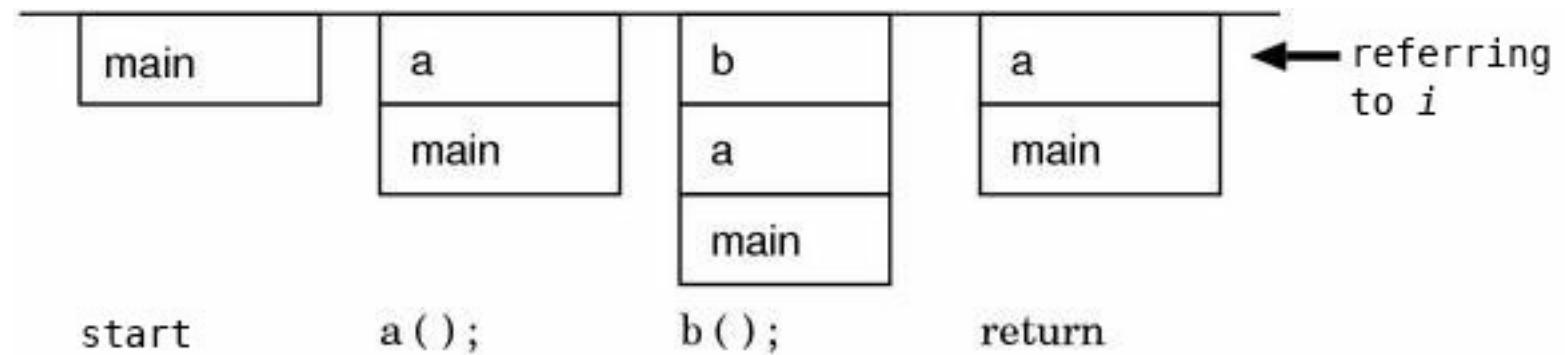


Fig.1: What is changing is only the stack

The execution of Ruby is also basically nothing but chained calls of methods which are procedures, so essentially it has the same image as above. In other words, with the same code, things being accessed such as local variable scope and the block local scope will be changing. And these kind of scopes are expressed by stacks.

However in Ruby, for instance, you can temporarily go back to the scope previously used by using iterators or Proc. This cannot be implemented with just simply pushing/popping a stack. Therefore the frames of the Ruby stack will be intricately rearranged during execution. Although I call it “stack”, it could be better to consider it as a list.

Other than the method call, the local variable scope can also be changed on the class definitions. So, the method calls does not match the transitions of the local variable scope. Since there are also blocks, it's necessary to handle them separately. For these various reasons, surprisingly, there are seven stacks.

<b>Stack Pointer</b>	<b>Stack Frame Type</b>	<b>Description</b>
ruby_frame	struct FRAME	the records of method calls
ruby_scope	struct SCOPE	the local variable scope
ruby_block	struct BLOCK	the block scope
ruby_iter	struct iter	whether or not the current FRAME is an iterator
ruby_class	VALUE	the class to define methods on
ruby_cref	NODE ( NODE_CREF )	the class nesting information

C has only one stack and Ruby has seven stacks, by simple arithmetic, the executing image of Ruby is at least seven times more complicated than C. But it is actually not seven times at all, it's at least twenty times more complicated.

First, I'll briefly describe about these stacks and their stack frame structs. The defined file is either eval.c or evn.h. Basically these stack frames are touched only by eval.c ... is what it should be if it were possible, but gc.c needs to know the struct types when marking, so some of them are exposed in env.h.

Of course, marking could be done in the other file but gc.c, but it requires separated functions which cause slowing down. The

ordinary programs had better not care about such things, but both the garbage collector and the core of the evaluator is the ruby's biggest bottleneck, so it's quite worth to optimize even for just one method call.

## ruby\_frame

`ruby_frame` is a stack to record method calls. The stack frame struct is `struct FRAME`. This terminology is a bit confusing but please be aware that I'll distinctively write it just a frame when it means a "stack frame" as a general noun and `FRAME` when it means `struct FRAME`.

### ruby\_frame

```
16 extern struct FRAME {  
17     VALUE self;          /* self */  
18     int argc;           /* the argument count */  
19     VALUE *argv;         /* the array of argument values */  
20     ID last_func;       /* the name of this FRAME (when cal  
21     ID orig_func;       /* the name of this FRAME (when def  
22     VALUE last_class;   /* the class of last_func's receive  
23     VALUE cbase;        /* the base point for searching con  
24     struct FRAME *prev; /* to protect from GC. this will be  
25     struct FRAME *tmp;  /* the file name and the line numbe  
26     struct RNode *node; /* is this called with a block? */  
27     int iter;            /* the below two */  
28     int flags;           /* */  
29 } *ruby_frame;  
  
33 #define FRAME_ALLOCA 0 /* FRAME is allocated on the machin  
34 #define FRAME_MALLOC 1 /* FRAME is allocated by malloc */
```

(env.h)

First of all, since there's the `prev` member, you can infer that the stack is made of a linked list. (Fig.2)

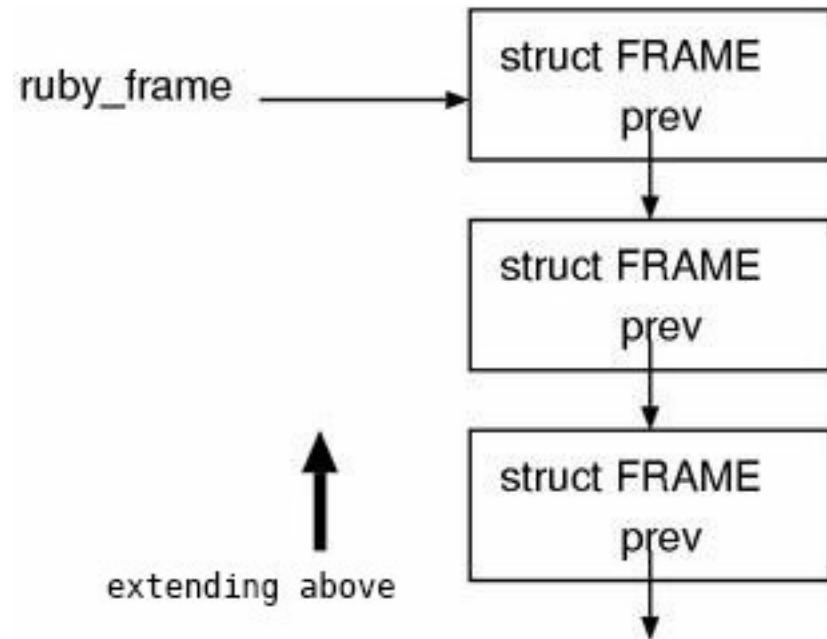


Fig.2: `ruby_frame`

The fact that `ruby_xxxx` points to the top stack frame is common to all stacks and won't be mentioned every time.

The first member of the struct is `self`. There is also `self` in the arguments of `rb_eval()`, but why this struct remembers another `self`? This is for the C-level functions. More precisely, it's for `rb_call_super()` that is corresponding to `super`. In order to execute `super`, it requires the receiver of the current method, but the caller side of `rb_call_super()` could not have such information. However, the chain of `rb_eval()` is interrupted before the time when the execution of the user-defined C code starts. Therefore, the conclusion is that there need a way to obtain the information of `self` out of nothing. And, `FRAME` is the right place to store it.

Thinking a little further, It's mysterious that there are `argc` and `argv`. Because parameter variables are local variables after all, it is unnecessary to preserve the given arguments after assigning them into the local variable with the same names at the beginning of the method, isn't it? Then, what is the use of them ? The answer is that this is actually for `super` again. In Ruby, when calling `super` without any arguments, the values of the parameter variables of the method will be passed to the method of the superclass. Thus, (the local variable space for) the parameter variables must be reserved.

Additionally, the difference between `last_func` and `orig_func` will be come out in the cases like when the method is `alias` ed. For instance,

```
class C
  def orig() end
  alias ali orig
end
C.new.ali
```

in this case, `last_func=ali` and `orig_func=orig`. Not surprisingly, these members also have to do with `super`.

## ruby\_scope

`ruby_scope` is the stack to represent the local variable scope. The method and class definition statements, the module definition statements and the singleton class definition statements, all of them are different scopes. The stack frame struct is `struct SCOPE`.

I'll call this frame SCOPE.

## ▼ ruby\_scope

```
36 extern struct SCOPE {  
37     struct RBasic super;  
38     ID *local_tbl;          /* an array of the local variable  
39     VALUE *local_vars;     /* the space to store local variab  
40     int flags;            /* the below four */  
41 } *ruby_scope;  
  
43 #define SCOPE_ALLOCA 0          /* local_vars is allocated b  
44 #define SCOPE_MALLOC 1          /* local_vars is allocated b  
45 #define SCOPE_NOSTACK 2         /* POP_SCOPE is done */  
46 #define SCOPE_DONT_RECYCLE 4    /* Proc is created with this
```

(env.h)

Since the first element is struct RBasic, this is a Ruby object. This is in order to handle Proc objects. For example, let's try to think about the case like this:

```
def make_counter  
  lvar = 0  
  return Proc.new { lvar += 1 }  
end  
  
cnt = make_counter()  
p cnt.call    # 1  
p cnt.call    # 2  
p cnt.call    # 3  
cnt = nil    # cut the reference. The created Proc finally becomes
```

The Proc object created by this method will persist longer than the method that creates it. And, because the Proc can refer to the local variable lvar, the local variables must be preserved until the Proc

will disappear. Thus, if it were not handled by the garbage collector, no one can determine the time to free.

There are two reasons why `struct SCOPE` is separated from `struct FRAME`. Firstly, the things like class definition statements are not method calls but create distinct local variable scopes. Secondly, when a called method is defined in C the Ruby's local variable space is unnecessary.

## █ ruby\_block

`struct BLOCK` is the real body of a Ruby's iterator block or a `Proc` object, it is also kind of a snapshot of the evaluator at some point. This frame will also be briefly written as `BLOCK` as in the same manner as `FRAME` and `SCOPE`.

### ▼ ruby\_block

```
580 static struct BLOCK *ruby_block;

559 struct BLOCK {
560     NODE *var;                      /* the block parameters (mlhs)
561     NODE *body;                     /* the code of the block body *
562     VALUE self;                    /* the self when this BLOCK is
563     struct FRAME frame;          /* the copy of ruby_frame when
564     struct SCOPE *scope;          /* the ruby_scope when this BLO
565     struct BLOCKTAG *tag;         /* the identity of this BLOCK *
566     VALUE klass;                  /* the ruby_class when this BLO
567     int iter;                     /* the ruby_iter when this BLOC
568     int vmode;                    /* the scope_vmode when this BL
569     int flags;                   /* BLOCK_D_SCOPE, BLOCK_DYNAMIC
570     struct RVarmap *dyna_vars;   /* the block local variable
571     VALUE orig_thread;           /* the thread that creates this
572     VALUE wrapper;               /* the ruby_wrapper when this B
```

```
573     struct BLOCK *prev;
574 }

553 struct BLOCKTAG {
554     struct RBasic super;
555     long dst;                      /* destination, that is, the pl
556     long flags;                   /* BLOCK_DYNAMIC, BLOCK_ORPHAN
557 }

576 #define BLOCK_D_SCOPE 1          /* having distinct block local
577 #define BLOCK_DYNAMIC 2          /* BLOCK was taken from a Ruby
578 #define BLOCK_ORPHAN 4           /* the FRAME that creates this
```

(eval.c)

Note that `frame` is not a pointer. This is because the entire content of `struct FRAME` will be all copied and preserved. The entire `struct FRAME` is (for better performance) allocated on the machine stack, but `BLOCK` could persist longer than the `FRAME` that creates it, the preservation is a preparation for that case.

Additionally, `struct BLOCKTAG` is separated in order to detect the same block when multiple `Proc` objects are created from the block. The `Proc` objects which were created from the one same block have the same `BLOCKTAG`.

## ruby\_iter

The stack `ruby_iter` indicates whether currently calling method is an iterator (whether it is called with a block). The frame is `struct iter`. But for consistency I'll call it `ITER`.

## ▼ ruby\_iter

```
767 static struct iter *ruby_iter;  
  
763 struct iter {  
764     int iter;           /* the below three */  
765     struct iter *prev;  
766 };  
  
769 #define ITER_NOT 0      /* the currently evaluated method is  
770 #define ITER_PRE 1      /* the method which is going to be e  
771 #define ITER_CUR 2      /* the currently evaluated method is  
(eval.c)
```

Although for each method we can determine whether it is an iterator or not, there's another struct that is distinct from `struct FRAME`. Why?

It's obvious you need to inform it to the method when “it is an iterator”, but you also need to inform the fact when “it is not an iterator”. However, pushing a whole `BLOCK` just for this is very heavy. It will also cause that in the caller side the procedures such as variable references would needlessly increase. Thus, it's better to push the smaller and lighter `ITER` instead of `BLOCK`. This will be discussed in detail in Chapter 16: Blocks.

## █ ruby\_dyna\_vars

The block local variable space. The frame struct is `struct RVarmap` that has already seen in Part 2. Form now on, I'll call it just `VARS`.

## ▼ struct RVarmap

```
52 struct RVarmap {  
53     struct RBasic super;  
54     ID id;           /* the name of the variable */  
55     VALUE val;       /* the value of the variable */  
56     struct RVarmap *next;  
57 };
```

(env.h)

Note that a frame is not a single struct RVarmap but a list of the structs (Fig.3). And each frame is corresponding to a local variable scope. Since it corresponds to “local variable scope” and not “block local variable scope”, for instance, even if blocks are nested, only a single list is used to express. The break between blocks are similar to the one of the parser, it is expressed by a RVarmap (header) whose id is 0. Details are deferred again. It will be explained in Chapter 16: Blocks.

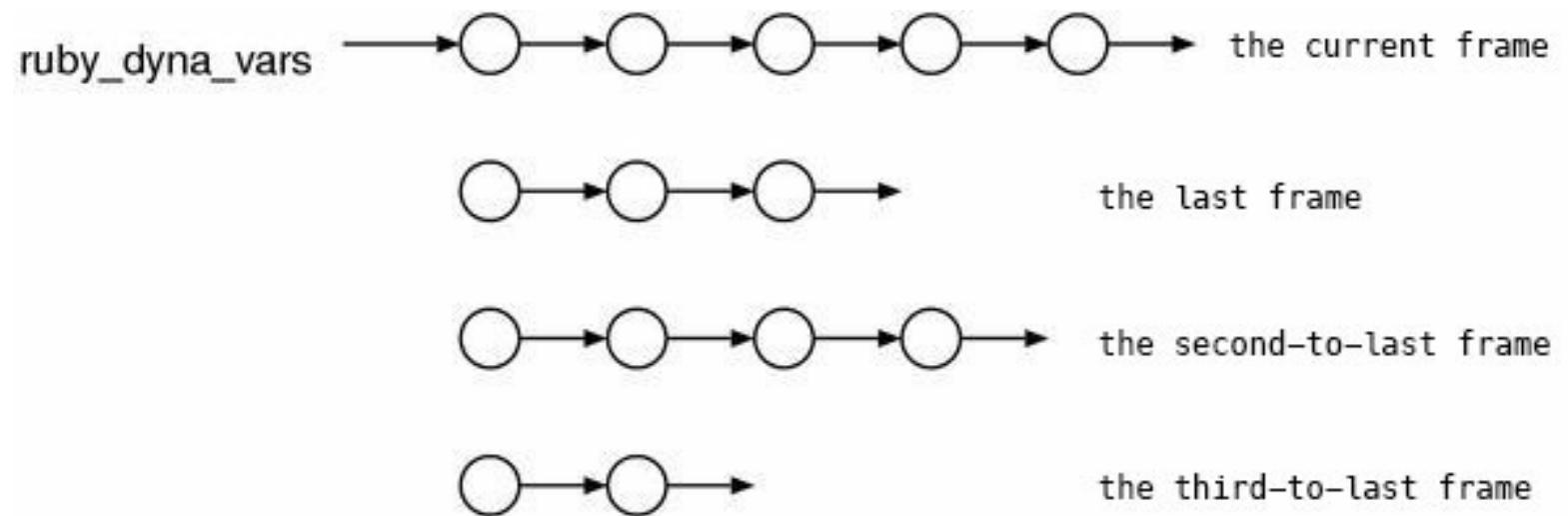


Fig.3: ruby\_dyna\_vars

## ruby\_class

`ruby_class` represents the current class to which a method is

defined. Since `self` will be that class when it's a normal class definition statement, `ruby_class == self`. But, when it is the top level or in the middle of particular methods like `eval` and `instance_eval`, `self != ruby_class` is possible.

The frame of `ruby_class` is a simple `VALUE` and there's no particular frame struct. Then, how could it be like a stack? Moreover, there were many structs without the `prev` pointer, how could these form a stack? The answer is deferred to the next section.

From now on, I'll call this frame `CLASS`.

## ▀ ruby\_cref

`ruby_cref` represents the information of the nesting of a class. I'll call this frame `CREF` with the same way of naming as before. Its struct is ...

### ▼ ruby\_cref

```
847 static NODE *ruby_cref = 0;
```

```
(eval.c)
```

... surprisingly `NODE`. This is used just as a “defined struct which can be pointed by a `VALUE`”. The node type is `NODE_CREF` and the assignments of its members are shown below:

Union Member	Macro To Access
--------------	-----------------

Usage
-------

u1.value	nd_clss	the outer class ( VALUE )
u2	-	-
u3.node	nd_next	preserve the previous CREF

Even though the member name is `nd_next`, the value it actually has is the “previous (prev)” CREF. Taking the following program as an example, I’ll explain the actual appearance.

```
class A
  class B
    class C
      nil  # (A)
    end
  end
end
```

Fig.4 shows how `ruby_cref` is when evaluating the code (A).

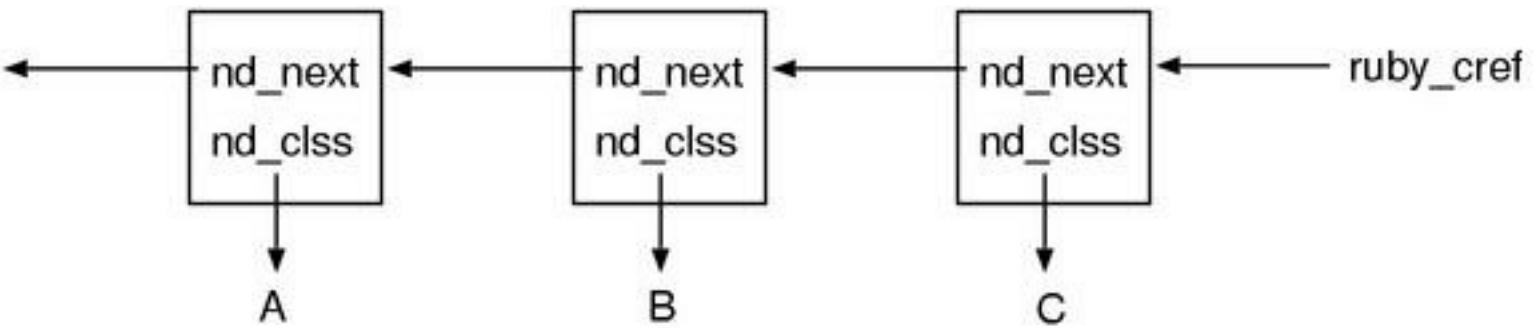


Fig.4: `ruby_cref`

However, illustrating this image everytime is tedious and its intention becomes unclear. Therefore, the same state as Fig.4 will be expressed in the following notation:

`A ← B ← C`

# PUSH / POP Macros

For each stack frame struct, the macros to push and pop are available. For instance, `PUSH_FRAME` and `POP_FRAME` for `FRAME`. Because these will appear in a moment, I'll then explain the usage and content.

## The other states

While they are not so important as the main stacks, the evaluator of `ruby` has the several other states. This is a brief list of them. However, some of them are not stacks. Actually, most of them are not.

Variable Name	Type	Meaning
<code>scope_vmode</code>	<code>int</code>	the default visibility when a method is defined
<code>ruby_in_eval</code>	<code>int</code>	whether or not parsing after the evaluation is started
<code>ruby_current_node</code>	<code>NODE*</code>	the file name and the line number of what currently being evaluated
<code>ruby_safe_level</code>	<code>int</code>	<code>\$SAFE</code>
<code>ruby_errinfo</code>	<code>VALUE</code>	the exception currently being handled
<code>ruby_wrapper</code>	<code>VALUE</code>	the wrapper module to isolate the environment

# Module Definition

---

The `class` statement and the `module` statement and the singleton class definition statement, they are all implemented in similar ways.

Because seeing similar things continuously three times is not interesting, this time let's examine the `module` statement which has the least elements (thus, is simple).

First of all, what is the `module` statement? Conversely, what should happen is the `module` statement ? Let's try to list up several features:

- a new module object should be created
- the created module should be `self`
- it should have an independent local variable scope
- if you write a constant assignment, a constant should be defined on the module
- if you write a class variable assignment, a class variable should be defined on the module.
- if you write a `def` statement, a method should be defined on the module

What is the way to archive these things? ... is the point of this section. Now, let's start to look at the codes.

## ■ Investigation

## ▼ The Source Program

```
module M
  a = 1
end
```

## ▼ Its Syntax Tree

```
NODE_MODULE
nd_cname = 9621 (M)
nd_body:
  NODE_SCOPE
    nd_rval = (null)
    nd_tbl = 3 [ _ ~ a ]
    nd_next:
      NODE_LASGN
        nd_cnt = 2
        nd_value:
          NODE_LIT
            nd_lit = 1:Fixnum
```

`nd_cname` seems the module name. `cname` is probably either Const NAME or Class NAME. I dumped several things and found that there's always `NODE_SCOPE` in `nd_body`. Since its member `nd_tbl` holds a local variable table and its name is similar to `struct SCOPE`, it appears certain that this `NODE_SCOPE` plays an important role to create a local variable scope.

### ■ NODE\_MODULE

Let's examine the handler of `NODE_MODULE` of `rb_eval()`. The parts that are not close to the main line, such as `ruby_raise()` and error handling were cut drastically. So far, there have been a lot of

cutting works for 200 pages, it has already became unnecessary to show the original code.

## ▼ rb\_eval() – NODE\_MODULE (simplified)

```
case NODE_MODULE:
{
    VALUE module;

    if (rb_const_defined_at(ruby_class, node->nd_cname)) {
        /* just obtain the already created module */
        module = rb_const_get(ruby_class, node->nd_cname);
    }
    else {
        /* create a new module and set it into the constant */
        module = rb_define_module_id(node->nd_cname);
        rb_const_set(ruby_cbase, node->nd_cname, module);
        rb_set_class_path(module, ruby_class, rb_id2name(node->nd_
    }

    result = module_setup(module, node->nd_body);
}
break;
```

First, we'd like to make sure the module is nested and defined above (the module holded by) `ruby_class`. We can understand it from the fact that it calls `ruby_const_xxxx()` on `ruby_class`. Just once `ruby_cbase` appears, but it is usually identical to `ruby_class`, so we can ignore it. Even if they are different, it rarely causes a problem.

The first half, it is branching by `if` because it needs to check if the module has already been defined. This is because, in Ruby, we can do “additional” definitions on the same one module any number of times.

```
module M
  def a      # M#a is defined
  end
end
module M    # add a definition (not re-defining or overwriting)
  def b      # M#b is defined
  end
end
```

In this program, the two methods, `a` and `b`, will be defined on the module `M`.

In this case, on the second definition of `M` the module `M` was already set to the constant, just obtaining and using it would be sufficient. If the constant `M` does not exist yet, it means the first definition and the module is created (by `rb_define_module_id()` )

Lastly, `module_setup()` is the function executing the body of a module statement. Not only the module statements but the class statements and the singleton class statements are executed by `module_setup()`. This is the reason why I said “all of these three type of statements are similar things”. For now, I’d like you to note that `node->nd_body ( NODE_SCOPE )` is passed as an argument.

## ■ `module_setup`

For the module and class and singleton class statements, `module_setup()` executes their bodies. Finally, the Ruby stack manipulations will appear in large amounts.

## ▼ `module_setup()`



```
3464         }
3465         result = rb_eval(ruby_class, node->nd_next);
3466     }
3467     POP_TAG();
3468     POP_CREF();
3469     POP_VARS();
3470     POP_SCOPE();
3471     POP_CLASS();
3472
3473     ruby_frame = frame.tmp;
3474     if (trace_func) {
3475         call_trace_func("end", ruby_last_node, 0,
3476                         ruby_frame->last_func, ruby_frame->l
3477     }
3478     if (state) JUMP_TAG(state);
3479
3480     return result;
3481 }
```

(eval.c)

This is too big to read all in one gulp. Let's cut the parts that seems unnecessary.

First, the parts around `trace_func` can be deleted unconditionally.

We can see the idioms related to tags. Let's simplify them by expressing with the Ruby's `ensure`.

Immediately after the start of the function, the argument `n` is purposefully assigned to the local variable `node`, but `volatile` is attached to `node` and it would never be assigned after that, thus this is to prevent from being garbage collected. If we assume that the argument was `node` from the beginning, it would not change the meaning.

In the first half of the function, there's the part manipulating `ruby_frame` complicatedly. It is obviously paired up with the part `ruby_frame = frame.tmp` in the last half. We'll focus on this part later, but for the time being this can be considered as `push pop` of `ruby_frame`.

Plus, it seems that the code (A) can be, as commented, summarized as the initialization of `ruby_scope->local_vars`. This will be discussed later.

Consequently, it could be summarized as follows:

### ▼ `module_setup` (simplified)

```
static VALUE
module_setup(module, node)
    VALUE module;
    NODE *node;
{
    struct FRAME frame;
    VALUE result;

    push FRAME
    PUSH_CLASS();
    ruby_class = module;
    PUSH_SCOPE();
    PUSH_VARS();
    ruby_scope->local_vars initialization
    PUSH_CREF(module);
    ruby_frame->cbase = (VALUE)ruby_cref;
    begin
        result = rb_eval(ruby_class, node->nd_next);
    ensure
        POP_TAG();
        POP_CREF();
        POP_VARS();
```

```
POP_SCOPE();
POP_CLASS();
pop FRAME
end
return result;
}
```

It does `rb_eval()` with `node->nd_next`, so it's certain that this is the code of the module body. The problems are about the others. There are 5 points to see.

- Things occur on `PUSH_SCOPE()` `PUSH_VARS()`
- How the local variable space is allocated
- The effect of `PUSH_CLASS`
- The relationship between `ruby_cref` and `ruby_frame->cbase`
- What is done by manipulating `ruby_frame`

Let's investigate them in order.

## Creating a local variable scope

`PUSH_SCOPE` pushes a local variable space and `PUSH_VARS()` pushes a block local variable space, thus a new local variable scope is created by these two. Let's examine the contents of these macros and what is done.

### ▼ `PUSH_SCOPE()` `POP_SCOPE()`

```
852 #define PUSH_SCOPE() do { \
853     volatile int _vmode = scope_vmode; \
854     struct SCOPE * volatile _old; \

```

```

855     NEWOBJ(_scope, struct SCOPE);           \
856     OBJSETUP(_scope, 0, T_SCOPE);           \
857     _scope->local_tbl = 0;                  \
858     _scope->local_vars = 0;                 \
859     _scope->flags = 0;                     \
860     _old = ruby_scope;                    \
861     ruby_scope = _scope;                  \
862     scope_vmode = SCOPE_PUBLIC

869 #define POP_SCOPE()
870     if (ruby_scope->flags & SCOPE_DONT_RECYCLE) { \
871         if (_old) scope_dup(_old); \
872     } \
873     if (!(ruby_scope->flags & SCOPE_MALLOC)) { \
874         ruby_scope->local_vars = 0; \
875         ruby_scope->local_tbl = 0; \
876         if (!(ruby_scope->flags & SCOPE_DONT_RECYCLE) && \
877             ruby_scope != top_scope) { \
878             rb_gc_force_recycle((VALUE)ruby_scope); \
879         } \
880     } \
881     ruby_scope->flags |= SCOPE_NOSTACK; \
882     ruby_scope = _old; \
883     scope_vmode = _vmode; \
884 } while (0)

```

(eval.c)

As the same as tags, SCOPE s also create a stack by being synchronized with the machine stack. What differentiate slightly is that the spaces of the stack frames are allocated in the heap, the machine stack is used in order to create the stack structure (Fig.5.).

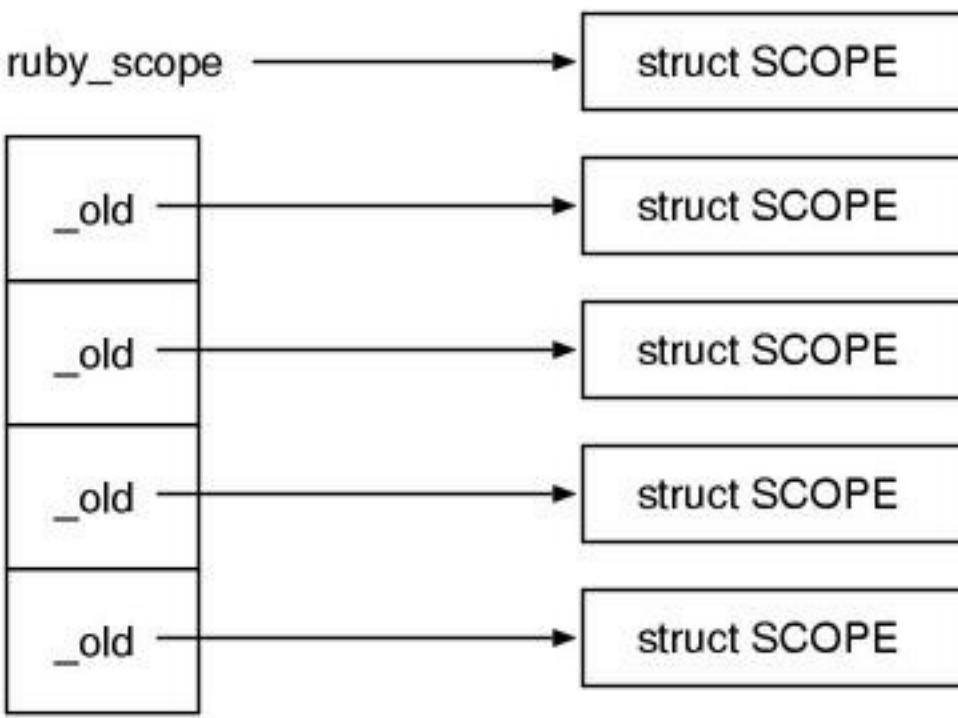


Fig.5. The machine stack and the SCOPE Stack

Additionally, the flags like `SCOPE_` something repeatedly appearing in the macros are not able to be explained until I finish to talk all about in what form each stack frame is remembered and about blocks. Thus, these will be discussed in Chapter 16: Blocks all at once.

## Allocating the local variable space

As I mentioned many times, the local variable scope is represented by `struct SCOPE`. But `struct SCOPE` is literally a “scope” and it does not have the real body to store local variables. To put it more precisely, it has the pointer to a space but there’s still no array at the place where the one points to. The following part of `module_setup` prepares the array.

▼ The preparation of the local variable slots

```

3444 if (node->nd_tbl) {
3445     VALUE *vars = TMP_ALLOC(node->nd_tbl[0]+1);
3446     *vars++ = (VALUE)node;
3447     ruby_scope->local_vars = vars;
3448     rb_mem_clear(ruby_scope->local_vars, node->nd_tbl[0]);
3449     ruby_scope->local_tbl = node->nd_tbl;
3450 }
3451 else {
3452     ruby_scope->local_vars = 0;
3453     ruby_scope->local_tbl = 0;
3454 }

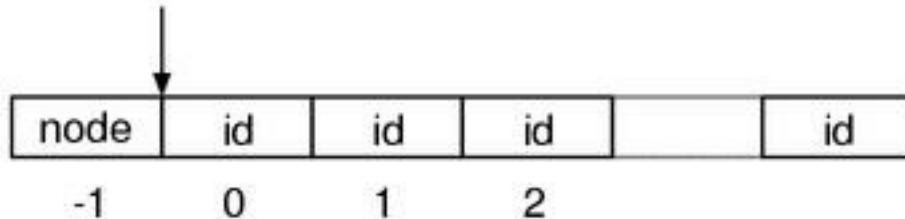
```

(eval.c)

The `TMP_ALLOC()` at the beginning will be described in the next section. If I put it shortly, it is “`alloca` that is assured to allocate on the stack (therefore, we do not need to worry about GC)”.

`node->nd_tbl` holds in fact the local variable name table that has appeared in Chapter 12: Syntax tree construction. It means that `nd_tbl[0]` contains the table size and the rest is an array of ID. This table is directly preserved to `local_tbl` of `SCOPE` and `local_vars` is allocated to store the local variable values. Because they are confusing, it's a good thing writing some comments such as “This is the variable name”, “this is the value”. The one with `tbl` is for the names.

`ruby_scope -> local_vars`



## Fig.6. ruby\_scope->local\_vars

Where is this node used? I examined the all `local_vars` members but could not find the access to index `-1` in `eval.c`. Expanding the range of files to investigate, I found the access in `gc.c`.

### ▼ `rb_gc_mark_children() — T_SCOPE`

```
815 case T_SCOPE:
816     if (obj->as.scope.local_vars &&
817         (obj->as.scope.flags & SCOPE_MALLOC)) {
818         int n = obj->as.scope.local_tbl[0]+1;
819         VALUE *vars = &obj->as.scope.local_vars[-1];
820         while (n--) {
821             rb_gc_mark(*vars);
822             vars++;
823         }
824     }
825     break;
```

(`gc.c`)

Apparently, this is a mechanism to protect `node` from GC. But why is it necessary to mark it here? `node` is purposefully stored into the `volatile` local variable, so it would not be garbage-collected during the execution of `module_setup()`.

Honestly speaking, I was thinking it might merely be a mistake for a while but it turned out it's actually very important. The issue is this at the next line of the next line:

### ▼ `ruby_scope->local_tbl`

```
3449 ruby_scope->local_tbl = node->nd_tbl;  
(eval.c)
```

The local variable name table prepared by the parser is directly used. When is this table freed? It's the time when the `node` become not to be referred from anywhere. Then, when should `node` be freed? It's the time after the `SCOPE` assigned on this line will disappear completely. Then, when is that?

`SCOPE` sometimes persists longer than the statement that causes the creation of it. As it will be discussed at Chapter 16: Blocks, if a `Proc` object is created, it refers `SCOPE`. Thus, If `module_setup()` has finished, the `SCOPE` created there is not necessarily be what is no longer used. That's why it's not sufficient that `node` is only referred from (the stack frame of) `module_setup()`. It must be referred “directly” from `SCOPE`.

On the other hand, the `volatile` `node` of the local variable cannot be removed. Without it, `node` is floating on air until it will be assigned to `local_vars`.

However then, `local_vars` of `SCOPE` is not safe, isn't it? `TMP_ALLOC()` is, as I mentioned, the allocation on the stack, it becomes invalid at the time `module_setup()` ends. This is in fact, at the moment when `Proc` is created, the allocation method is abruptly switched to `malloc()`. Details will be described in Chapter 16: Blocks.

Lastly, `rb_mem_clear()` seems zero-filling but actually it is `Qnil`-filling to an array of `VALUE` (`array.c`). By this, all defined local variables are initialized as `nil`.

## ■ TMP\_ALLOC

Next, let's read `TMP_ALLOC` that allocates the local variable space. This macro is actually paired with `TMP_PROTECT` existing silently at the beginning of `module_setup()`. Its typical usage is this:

```
VALUE *ptr;  
TMP_PROTECT;  
  
ptr = TMP_ALLOC(size);
```

The reason why `TMP_PROTECT` is in the place for the local variable definitions is that ... Let's see its definition.

## ▼ TMP\_ALLOC()

```
1769 #ifdef C_ALLOCA  
1770 # define TMP_PROTECT NODE * volatile tmp__protect_tmp=0  
1771 # define TMP_ALLOC(n) \  
1772     (tmp__protect_tmp = rb_node_newnode(NODE_ALLOCA,  
1773                                         ALLOC_N(VALUE,n), tmp__protect_  
1774                                         (void*)tmp__protect_tmp->nd_head)  
1775 #else  
1776 # define TMP_PROTECT typedef int foobazzz  
1777 # define TMP_ALLOC(n) ALLOCA_N(VALUE,n)  
1778 #endif  
  
(eval.c)
```

... it is because it defines a local variable.

As described in Chapter 5: Garbage collection, in the environment of `#ifdef C_ALLOCA` (that is, the native `alloca()` does not exist) `malloca()` is used to emulate `alloca()`. However, the arguments of a method are obviously `VALUE`s and the GC could not find a `VALUE` if it is stored in the heap. Therefore, it is enforced that GC can find it through `NODE`.

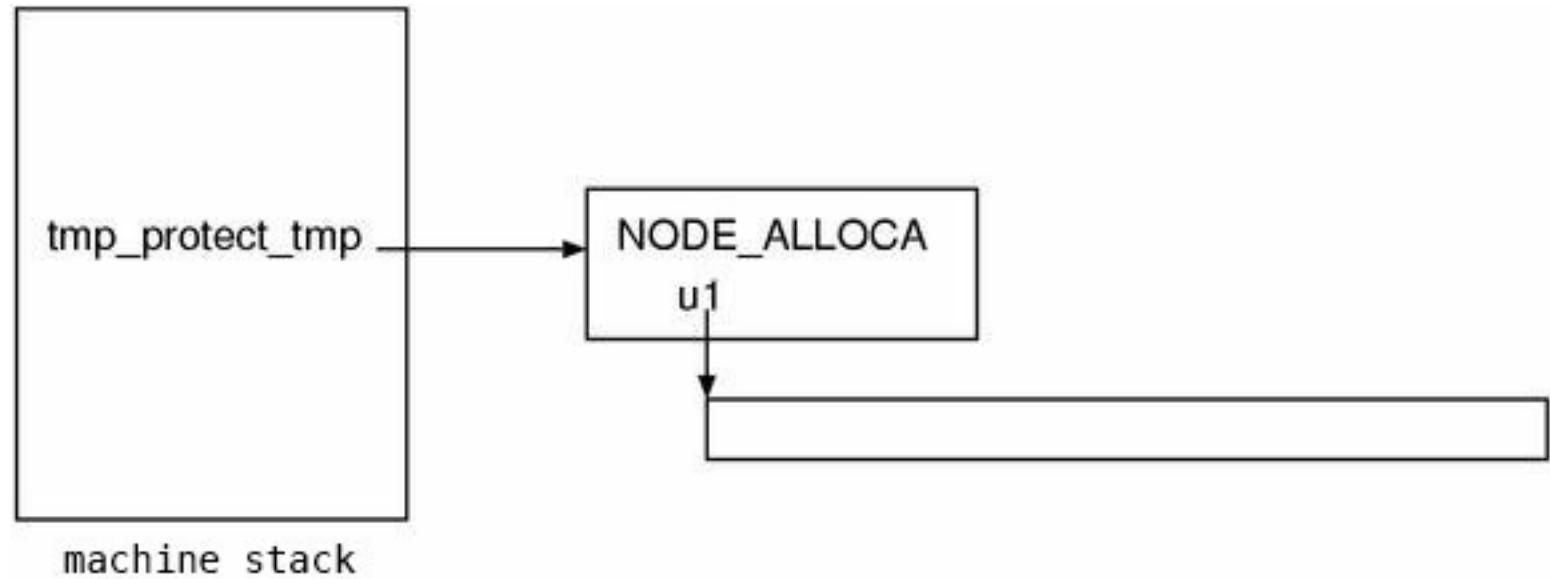


Fig.7. anchor the space to the stack through `NODE`

On the contrary, in the environment with the true `alloca()`, we can naturally use `alloca()` and there's no need to use `TMP_PROTECT`. Thus, a harmless statement is arbitrarily written.

By the way, why do they want to use `alloca()` very much by all means. It's merely because "`alloca()` is faster than `malloc()`", they said. One can think that it's not so worth to care about such tiny difference, but because the core of the evaluator is the biggest

bottleneck of ruby, ... the same as above.

## Changing the place to define methods on.

The value of the stack `ruby_class` is the place to define a method on at the time. Conversely, if one push a value to `ruby_class`, it changes the class to define a method on. This is exactly what is necessary for a class statement. Therefore, It's also necessary to do `PUSH_CLASS()` in `module_setup()`. Here is the code for it:

```
PUSH_CLASS();
ruby_class = module;
:
:
POP_CLASS();
```

Why is there the assignment to `ruby_class` after doing `PUSH_CLASS()`. We can understand it unexpectedly easily by looking at the definition.

### ▼ `PUSH_CLASS()` `POP_CLASS()`

```
841 #define PUSH_CLASS() do { \
842     VALUE _class = ruby_class
844 #define POP_CLASS() ruby_class = _class; \
845 } while (0)
```

`(eval.c)`

Because `ruby_class` is not modified even though `PUSH_CLASS` is done,

it is not actually pushed until setting by hand. Thus, these two are closer to “save and restore” rather than “push and pop”.

You might think that it can be a cleaner macro if passing a class as the argument of `PUSH_CLASS()` ... It's absolutely true, but because there are some places we cannot obtain the class before pushing, it is in this way.

# Nesting Classes

`ruby_cref` represents the class nesting information at runtime. Therefore, it's naturally predicted that `ruby_cref` will be pushed on the module statements or on the class statements. In `module_setup()`, it is pushed as follows:

```
PUSH_CREF(module);
ruby_frame->cbase = (VALUE)ruby_cref;
:
:
:
POP_CREF();
```

Here, `module` is the module being defined. Let's also see the definitions of `PUSH_CREF()` and `POP_CREF()`.

▼ PUSH CREF() POP CREF()

```
849 #define PUSH_CREF(c) \
      ruby_cref = rb_node_newnode(NODE_CREF, (c), 0, ruby_cref)
850 #define POP_CREF() ruby_cref = ruby_cref->nd_next
```

Unlike `PUSH_SCOPE` or something, there are not any complicated techniques and it's very easy to deal with. It's also not good if there's completely not any such thing.

The problem remains unsolved is what is the meaning of `ruby_frame->cbase`. It is the information to refer a class variable or a constant from the current `FRAME`. Details will be discussed in the last section of this chapter.

## Replacing frames

Lastly, let's focus on the manipulation of `ruby_frame`. The first thing is its definition:

```
struct FRAME frame;
```

It is not a pointer. This means that the entire `FRAME` is allocated on the stack. Both the management structure of the Ruby stack and the local variable space are on the stack, but in the case of `FRAME` the entire struct is stored on the stack. The extreme consumption of the machine stack by `ruby` is the fruit of these “small techniques” piling up.

Then next, let's look at where doing several things with `frame`.

```
frame = *ruby_frame;      /* copy the entire struct */
frame.tmp = ruby_frame;  /* protect the original FRAME from GC
ruby_frame = &frame;    /* replace ruby_frame */
:
:
```

```
ruby_frame = frame.tmp; /* restore */
```

That is, `ruby_frame` seems temporarily replaced (not pushing). Why is it doing such thing?

I described that `FRAME` is “pushed on method calls”, but to be more precise, it is the stack frame to represent “the main environment to execute a Ruby program”. You can infer it from, for instance, `ruby_frame->cbase` which appeared previously. `last_func` which is “the last called method name” also suggests it.

Then, why is `FRAME` not straightforwardly pushed? It is because this is the place where it is not allowed to push `FRAME`. `FRAME` is wanted to be pushed, but if `FRAME` is pushed, it will appear in the backtraces of the program when an exception occurs. The backtraces are things displayed like followings:

```
% ruby t.rb
t.rb:11:in `c': some error occurred (ArgumentError)
  from t.rb:7:in `b'
  from t.rb:3:in `a'
  from t.rb:14
```

But the module statements and the class statements are not method calls, so it is not desirable to appear in this. That’s why it is “replaced” instead of “pushed”.

## The method definition

As the next topic of the module definitions, let's look at the method definitions.

## ■ Investigation

### ▼ The Source Program

```
def m(a, b, c)
  nil
end
```

### ▼ Its Syntax Tree

```
NODE_DEFN
nd_mid  = 9617 (m)
nd_noex = 2 (NOEX_PRIVATE)
nd_defn:
  NODE_SCOPE
  nd_rval = (null)
  nd_tbl = 5 [ _ ~ a b c ]
  nd_next:
    NODE_ARGS
    nd_cnt  = 3
    nd_rest = -1
    nd_opt = (null)
  NODE NIL
```

I dumped several things and found that there's always NODE\_SCOPE in nd\_defn. NODE\_SCOPE is, as we've seen at the module statements, the node to store the information to push a local variable scope.

## ■ NODE\_DEFN

Subsequently, we will examine the corresponding code of `rb_eval()`. This part contains a lot of error handlings and tedious, they are all omitted again. The way of omitting is as usual, deleting the every parts to directly or indirectly call `rb_raise()` `rb_warn()` `rb_warning()`.

## ▼ `rb_eval()` – NODE\_DEFN (simplified)

```
NODE *defn;
int noex;

if (SCOPE_TEST(SCOPE_PRIVATE) || node->nd_mid == init) {
    noex = NOEX_PRIVATE;                                (A)
}
else if (SCOPE_TEST(SCOPE_PROTECTED)) {
    noex = NOEX_PROTECTED;                            (B)
}
else if (ruby_class == rb_cObject) {
    noex = node->nd_noex;                            (C)
}
else {
    noex = NOEX_PUBLIC;                            (D)
}

defn = copy_node_scope(node->nd_defn, ruby_cref);
rb_add_method(ruby_class, node->nd_mid, defn, noex);
result = Qnil;
```

In the first half, there are the words like `private` or `protected`, so it is probably related to visibility. `noex`, which is used as the names of flags, seems Node EXposure. Let's examine the `if` statements in order.

(A) `SCOPE_TEST()` is a macro to check if there's an argument flag in `scope_vmode`. Therefore, the first half of this conditional statement

means “is it a private scope?”. The last half means “it’s private if this is defining `initialize`”. The method `initialize` to initialize an object will unquestionably become `private`.

- (B) It is protected if the scope is protected (not surprisingly). My feeling is that there’re few cases `protected` is required in Ruby.
- (C) This is a bug. I found this just before the submission of this book, so I couldn’t fix this beforehand. In the latest code this part is probably already removed. The original intention is to enforce the methods defined at top level to be `private`.
- (D) If it is not any of the above conditions, it is `public`.

Actually, there’s not a thing to worth to care about until here. The important part is the next two lines.

```
defn = copy_node_scope(node->nd_defn, ruby_cref);
rb_add_method(ruby_class, node->nd_mid, defn, noex);
```

`copy_node_scope()` is a function to copy (only) `NODE_SCOPE` attached to the top of the method body. It is important that `ruby_cref` is passed ... but details will be described soon.

After copying, the definition is finished by adding it by `rb_add_method()`. The place to define on is of course `ruby_class`.

## copy\_node\_scope()

`copy_node_scope()` is called only from the two places: the method definition ( `NODE_DEFN` ) and the singleton method definition ( `NODE_DEFS` ) in `rb_eval()`. Therefore, looking at these two is sufficient to detect how it is used. Plus, the usages at these two places are almost the same.

## ▼ `copy_node_scope()`

```
1752 static NODE*
1753 copy_node_scope(node, rval)
1754     NODE *node;
1755     VALUE rval;
1756 {
1757     NODE *copy = rb_node_newnode(NODE_SCOPE, 0, rval, node->nd_
1758
1759     if (node->nd_tbl) {
1760         copy->nd_tbl = ALLOC_N(ID, node->nd_tbl[0]+1);
1761         MEMCPY(copy->nd_tbl, node->nd_tbl, ID, node->nd_tbl[
1762     }
1763     else {
1764         copy->nd_tbl = 0;
1765     }
1766     return copy;
1767 }
```

`(eval.c)`

I mentioned that the argument `rval` is the information of the class nesting ( `ruby_cref` ) of when the method is defined. Apparently, it is `rval` because it will be set to `nd_rval`.

In the main `if` statement copies `nd_tbl` of `NODE_SCOPE`. It is a local variable name table in other words. The `+1` at `ALLOC_N` is to additionally allocate the space for `nd_tbl[0]`. As we've seen in Part

2, `nd_tbl[0]` holds the local variables count, that was “the actual length of `nd_tbl - 1`”.

To summarize, `copy_node_scope()` makes a copy of the `NODE_SCOPE` which is the header of the method body. However, `nd_rval` is additionally set and it is the `ruby_cref` (the class nesting information) of when the class is defined. This information will be used later when referring constants or class variables.

## ■ `rb_add_method()`

The next thing is `rb_add_method()` that is the function to register a method entry.

### ▼ `rb_add_method()`

```
237 void
238 rb_add_method(klass, mid, node, noex)
239     VALUE klass;
240     ID mid;
241     NODE *node;
242     int noex;
243 {
244     NODE *body;
245
246     if (NIL_P(klass)) klass = rb_cObject;
247     if (ruby_safe_level >= 4 &&
248         (klass == rb_cObject || !OBJ_TAINTED(klass))) {
249         rb_raise(rb_eSecurityError, "Insecure: can't define
250             ");
251         if (OBJ_FROZEN(klass)) rb_error_frozen("class/module");
252         rb_clear_cache_by_id(mid);
253         body = NEW_METHOD(node, noex);
254         st_insert(RCLASS(klass)->m_tbl, mid, body);
255     }
```

NEW\_METHOD() is a macro to create NODE. rb\_clear\_cache\_by\_id() is a function to manipulate the method cache. This will be explained in the next chapter “Method”.

Let's look at the syntax tree which is eventually stored in `m_tbl` of a class. I prepared `nodedump-method` for this kind of purposes.  
 (nodedump-method : comes with nodedump. nodedump is tools/nodedump.tar.gz of the attached CD-ROM)

```
% ruby -e '
class C
  def m(a)
    puts "ok"
  end
end
require "nodedump-method"
NodeDump.dump C, :m          # dump the method m of the class C
'

NODE_METHOD
nd_noex = 0 (NOEX_PUBLIC)
nd_cnt = 0
nd_body:
  NODE_SCOPE
  nd_rval = Object <- C
  nd_tbl = 3 [ _ ~ a ]
  nd_next:
    NODE_ARGS
    nd_cnt = 1
    nd_rest = -1
    nd_opt = (null)
    U牙S頑著

** unhandled**
```

There are `NODE_METHOD` at the top and `NODE_SCOPE` previously copied by `copy_node_scope()` at the next. These probably represent the header of a method. I dumped several things and there's not any `NODE_SCOPE` with the methods defined in C, thus it seems to indicate that the method is defined at Ruby level.

Additionally, at `nd_tbl` of `NODE_SCOPE` the parameter variable name ( `a` ) appears. I mentioned that the parameter variables are equivalent to the local variables, and this briefly implies it.

I'll omit the explanation about `NODE_ARGS` here because it will be described at the next chapter "Method".

Lastly, the `nd_cnt` of the `NODE_METHOD`, it's not so necessary to care about this time. It is used when having to do with alias.

## Assignment and Reference

---

Come to think of it, most of the stacks are used to realize a variety of variables. We have learned to push various stacks, this time let's examine the code to reference variables.

### Local variable

The all necessary information to assign or refer local variables has appeared, so you are probably able to predict. There are the following two points:

- local variable scope is an array which is pointed by `ruby_scope->local_vars`
- the correspondence between each local variable name and each array index has already resolved at the parser level.

Therefore, the code for the local variable reference node `NODE_LVAR` is as follows:

### ▼ `rb_eval()` – `NODE_LVAR`

```
2975 case NODE_LVAR:  
2976     if (ruby_scope->local_vars == 0) {  
2977         rb_bug("unexpected local variable");  
2978     }  
2979     result = ruby_scope->local_vars[node->nd_cnt];  
2980     break;  
  
(eval.c)
```

It goes without saying but `node->nd_cnt` is the value that `local_cnt()` of the parser returns.

## Constant

## Complete Specification

In Chapter 6: Variables and constants, I talked about in what form

constants are stored and API. Constants are belong to classes and inherited as the same as methods. As for their actual appearances, they are registered to `iv_tbl` of struct `RClass` with instance variables and class variables.

The searching path of a constant is firstly the outer class, secondly the superclass, however, `rb_const_get()` only searches the superclass. Why? To answer this question, I need to reveal the last specification of constants. Take a look at the following code:

```
class A
  C = 5
  def A.new
    puts C
    super
  end
end
```

`A.new` is a singleton method of `A`, so its class is the singleton class (`A`). If it is interpreted by following the rule, it cannot obtain the constant `c` which is belongs to `A`.

But because it is written so close, to become to want refer the constant `c` is human nature. Therefore, such reference is possible in Ruby. It can be said that this specification reflects the characteristic of Ruby “The emphasis is on the appearance of the source code”.

If I generalize this rule, when referring a constant from inside of a method, by setting the place which the method definition is “written” as the start point, it refers the constant of the outer class.

And, “the class of where the method is written” depends on its context, thus it could not be handled without the information from both the parser and the evaluator. This is why `rb_cost_get()` did not have the searching path of the outer class.

## cbase

Then, let's look at the code to refer constants including the outer class. The ordinary constant references to which `::` is not attached, become `NODE_CONST` in the syntax tree. The corresponding code in `rb_eval()` is ...

### ▼ `rb_eval()` – `NODE_CONST`

```
2994 case NODE_CONST:  
2995     result = ev_const_get(RNODE(ruby_frame->cbase), node->nd_v  
2996     break;
```

`(eval.c)`

First, `nd_vid` appears to be `variable ID` and it probably means a constant name. And, `ruby_frame->cbase` is “the class where the method definition is written”. The value will be set when invoking the method, thus the code to set has not appeared yet. And the place where the value to be set comes from is the `nd_rval` that has appeared in `copy_node_scope()` of the method definition. I'd like you to go back a little and check that the member holds the `ruby_cref` of when the method is defined.

This means, first, the `ruby_cref` link is built when defining a class or a module. Assume that the just defined class is `c` (Fig.81),

Defining the method `m` (this is probably `c#m`) here, then the current `ruby_cref` is memorized by the method entry (Fig.82).

After that, when the class statement finished the `ruby_cref` would start to point another node, but `node->nd_rval` naturally continues to point to the same thing. (Fig.83)

Then, when invoking the method `c#m`, get `node->nd_rval` and insert into the just pushed `ruby_frame->cbase` (Fig.84)

... This is the mechanism. Complicated.

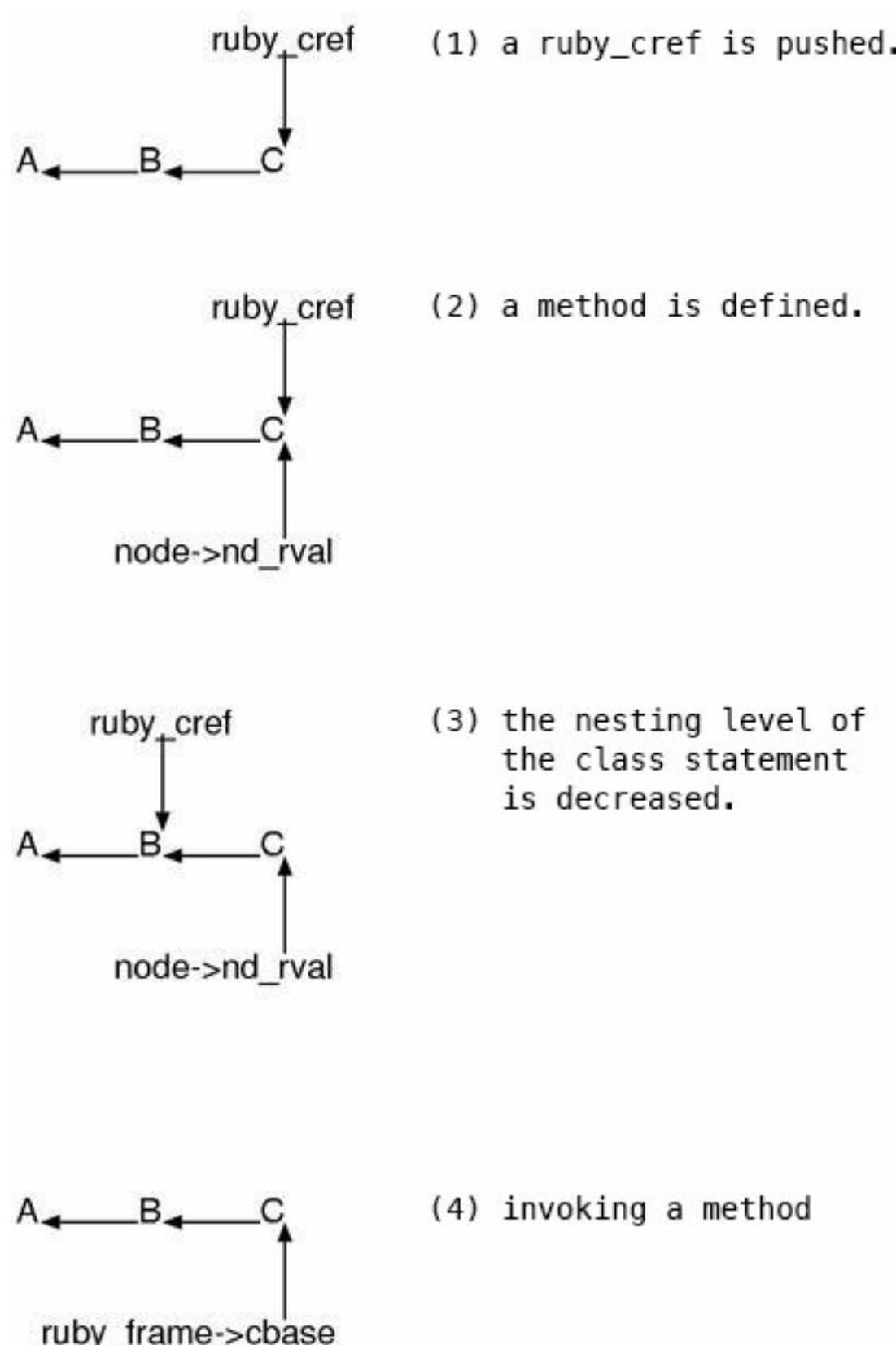


Fig 8. CREF Transfer

## ev\_const\_get()

Now, let's go back to the code of `NODE_CONST`. Since only `ev_const_get()` is left, we'll look at it.

## ▼ ev\_const\_get()

```
1550 static VALUE
1551 ev_const_get(cref, id, self)
1552     NODE *cref;
1553     ID id;
1554     VALUE self;
1555 {
1556     NODE *cbase = cref;
1557     VALUE result;
1558
1559     while (cbase && cbase->nd_next) {
1560         VALUE klass = cbase->nd_clss;
1561
1562         if (NIL_P(klass)) return rb_const_get(CLASS_OF(self))
1563         if (RCLASS(klass)->iv_tbl &&
1564             st_lookup(RCLASS(klass)->iv_tbl, id, &result)) {
1565             return result;
1566         }
1567         cbase = cbase->nd_next;
1568     }
1569     return rb_const_get(cref->nd_clss, id);
1570 }
```

(eval.c)

(( According to the errata, the description of ev\_const\_get() was wrong. I omit this part for now. ))

## ■ Class variable

What class variables refer to is also `ruby_cref`. Needless to say, unlike the constants which search over the outer classes one after another, it uses only the first element. Let's look at the code of `NODE_CVAR` which is the node to refer to a class variable.

What is the `cvar_cbase()` ? As `cbase` is attached, it is probably related to `ruby_frame->cbase`, but how do they differ? Let's look at it.

## ▼ `cvar_cbase()`

```
1571 static VALUE
1572 cvar_cbase()
1573 {
1574     NODE *cref = RNODE(ruby_frame->cbase);
1575
1576     while (cref && cref->nd_next &&
1577             FL_TEST(cref->nd_clss, FL_SINGLETON)) {
1578         cref = cref->nd_next;
1579         if (!cref->nd_next) {
1580             rb_warn("class variable access from toplevel singleton met
1581         }
1582         return cref->nd_clss;
1583     }
1584 }
```

(eval.c)

It traverses `cbase` up to the class that is not the singleton class, it seems. This feature is added to counter the following kind of code:

```
class C
  @@cvar = 1
  class << C
    def m
      @@cvar
    end
    def m2
      @@cvar + @@cvar
    end
  end
end
```

```
class C
  @@cvar = 1
  def C.m
    @@cvar
  end
  def C.m2
    @@cvar + @@cvar
  end
end
```

Both the left and right code ends up defining the same method, but if you write in the way of the right side it is tedious to write the class name repeatedly as the number of methods increases. Therefore, when defining multiple singleton methods, many people choose to write in the left side way of using the singleton class definition statement to bundle.

However, these two differs in the value of `ruby_cref`. The one using the singleton class definition is `ruby_cref=(C)` and the other one defining singleton methods separately is `ruby_cref=C`. This may cause to differ in the places where class variables refer to, so this is not convenient.

Therefore, assuming it's rare case to define class variables on singleton classes, it skips over singleton classes. This reflects again that the emphasis is more on the usability rather than the consistency.

And, when the case is a constant reference, since it searches all of the outer classes, `C` is included in the search path in either way, so there's no problem. Plus, as for an assignment, since it couldn't be written inside methods in the first place, it is also not related.

## ■ **Multiple Assignment**

If someone asked “where is the most complicated specification of Ruby?”, I would instantly answer that it is multiple assignment. It is even impossible to understand the big picture of multiple

assignment, I have an account of why I think so. In short, the specification of the multiple assignment is defined without even a subtle intention to construct so that the whole specification is well-organized. The basis of the specification is always “the behavior which seems convenient in several typical use cases”. This can be said about the entire Ruby, but particularly about the multiple assignment.

Then, how could we avoid being lost in the jungle of codes. This is similar to reading the stateful scanner and it is not seeing the whole picture. There’s no whole picture in the first place, we could not see it. Cutting the code into blocks like, this code is written for this specification, that code is written for that specification, ... understanding the correspondences one by one in such manner is the only way.

But this book is to understand the overall structure of ruby and is not “Advanced Ruby Programming”. Thus, dealing with very tiny things is not fruitful. So here, we only think about the basic structure of multiple assignment and the very simple “multiple-to-multiple” case.

First, following the standard, let’s start with the syntax tree.

## ▼ The Source Program

```
a, b = 7, 8
```

## ▼ Its Syntax Tree

```

NODE_MASGN
nd_head:
  NODE_ARRAY [
    0:
      NODE_LASGN
      nd_cnt = 2
      nd_value:
    1:
      NODE_LASGN
      nd_cnt = 3
      nd_value:
    ]
nd_value:
  NODE_REXPAND
  nd_head:
    NODE_ARRAY [
      0:
        NODE_LIT
        nd_lit = 7:Fixnum
      1:
        NODE_LIT
        nd_lit = 8:Fixnum
    ]

```

Both the left-hand and right-hand sides are the lists of NODE\_ARRAY, there's additionally NODE\_REXPAND in the right side. REXPAND may be “Right value EXPAND”. We are curious about what this node is doing. Let's see.

## ▼ rb\_eval() – NODE\_REXPAND

```

2575 case NODE_REXPAND:
2576   result = avalue_to_svalue(rb_eval(self, node->nd_head));
2577   break;

(eval.c)

```

You can ignore `avalue_to_svalue()`. `NODE_ARRAY` is evaluated by `rb_eval()`, (because it is the node of the array literal), it is turned into a Ruby array and returned back. So, before the left-hand side is handled, all in the right-hand side are evaluated. This enables even the following code:

```
a, b = b, a      # swap variables in oneline
```

Let's look at `NODE_MASGN` in the left-hand side.

### ▼ `rb_eval()` – `NODE_MASGN`

```
2923 case NODE_MASGN:  
2924     result = massign(self, node, rb_eval(self, node->nd_value))  
2925     break;  
  
(eval.c)
```

Here is only the evaluation of the right-hand side, the rests are delegated to `massign()`.

## massign()

### ▼ `massi` .....

```
3917 static VALUE  
3918 massign(self, node, val, pcall)  
3919     VALUE self;  
3920     NODE *node;  
3921     VALUE val;  
3922     int pcall;  
3923 {
```

I'm sorry this is halfway, but I'd like you to stop and pay attention to the 4th argument. `pcall` is `Proc CALL`, this indicates whether or not the function is used to call `Proc` object. Between `Proc` calls and the others there's a little difference in the strictness of the check of the multiple assignments, so a flag is received to check. Obviously, the value is decided to be either 0 or 1.

Then, I'd like you to look at the previous code calling `massign()`, it was `pcall=0`. Therefore, we probably don't mind if assuming it is `pcall=0` for the time being and extracting the variables. That is, when there's an argument like `pcall` which is slightly changing the behavior, we always need to consider the two patterns of scenarios, so it is really cumbersome. If there's only one actual function `massign()`, to think as if there were two functions, `pcall=0` and `pcall=1`, is way simpler to read.

When writing a program we must avoid duplications as much as possible, but this principle is unrelated if it is when reading. If patterns are limited, copying it and letting it to be redundant is rather the right approach. There are wordings “optimize for speed” “optimize for the code size”, in this case we'll “optimize for readability”.

So, assuming it is `pcall=0` and cutting the codes as much as possible and the final appearance is shown as follows:

## ▼ massign() (simplified)

```
static VALUE
massign(self, node, val /* , pcall=0 */)
{
    VALUE self;
    NODE *node;
    VALUE val;
    NODE *list;
    long i = 0, len;

    val = svalue_to_mvalue(val);
    len = RARRAY(val)->len;
    list = node->nd_head;
    /* (A) */
    for (i=0; list && i<len; i++) {
        assign(self, list->nd_head, RARRAY(val)->ptr[i], pcall);
        list = list->nd_next;
    }
    /* (B) */
    if (node->nd_args) {
        if (node->nd_args == (NODE*)-1) {
            /* no check for mere '*' */
        }
        else if (!list && i<len) {
            assign(self, node->nd_args,
                   rb_ary_new4(len-i, RARRAY(val)->ptr+i), pcall);
        }
        else {
            assign(self, node->nd_args, rb_ary_new2(0), pcall);
        }
    }
    /* (C) */
    while (list) {
        i++;
        assign(self, list->nd_head, Qnil, pcall);
        list = list->nd_next;
    }
    return val;
}
```

`val` is the right-hand side value. And there's the suspicious conversion called `svalue_to_mvalue()`, since `mvalue_to_svalue()` appeared previously and `svalue_to_mvalue()` in this time, so you can infer “it must be getting back”. ((errata: it was `avalue_to_svalue()` in the previous case. Therefore, it's hard to infer “getting back”, but you can ignore them anyway.)) Thus, the both are deleted. In the next line, since it uses `RARRAY()`, you can infer that the right-hand side value is an `Array` of Ruby. Meanwhile, the left-hand side is `node->nd_head`, so it is the value assigned to the local variable `list`. This `list` is also a node (`NODE_ARRAY`).

We'll look at the code by clause.

(A) `assign` is, as the name suggests, a function to perform an one-to-one assignment. Since the left-hand side is expressed by a node, if it is, for instance, `NODE_IASGN` (an assignment to an instance variable), it assigns with `rb_ivar_set()`. So, what it is doing here is adjusting to either `list` and `val` which is shorter and doing one-to-one assignments. (Fig.9)

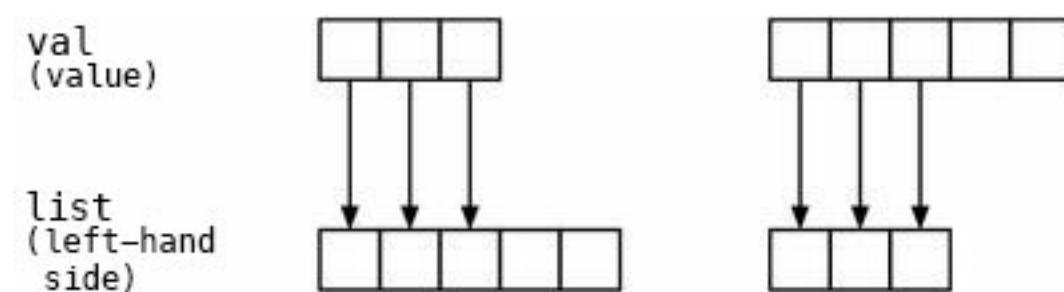


Fig.9. `assign` when corresponded

(B) if there are remainders on the right-hand side, turn them into a

Ruby array and assign it into (the left-hand side expressed by) the `node->nd_args`.

(C) if there are remainders on the left-hand side, assign `nil` to all of them.

By the way, the procedure which is assuming `pcall=0` then cutting out is very similar to the data flow analytics / constant foldings used on the optimization phase of compilers. Therefore, we can probably automate it to some extent.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

# Chapter 15: Methods

In this chapter, I'll talk about method searching and invoking.

## Searching methods

---

### Terminology

In this chapter, both method calls and method definitions are discussed, and there will appear really various “arguments”. Therefore, to make it not confusing, let's strictly define terms here:

```
m(a)          # a is a "normal argument"  
m(*list)      # list is an "array argument"  
m(&block)      # block is a "block argument"  
  
def m(a)      # a is a "normal parameter"  
def m(a=nil)  # a is an "option parameter", nil is "it default \n  
def m(*rest)  # rest is a "rest parameter"  
def m(&block)  # block is a "block parameter"
```

In short, they are all “arguments” when passing and “parameters” when receiving, and each adjective is attached according to its type.

However, among the above things, the “block arguments” and the “block parameters” will be discussed in the next chapter.

# Investigation

## ▼ The Source Program

```
obj.method(7,8)
```

## ▼ Its Syntax Tree

```
NODE_CALL
nd_mid = 9049 (method)
nd_recv:
  NODE_VCALL
    nd_mid = 9617 (obj)
nd_args:
  NODE_ARRAY [
    0:
      NODE_LIT
        nd_lit = 7:Fixnum
    1:
      NODE_LIT
        nd_lit = 8:Fixnum
  ]
```

The node for a method call is `NODE_CALL`. The `nd_args` holds the arguments as a list of `NODE_ARRAY`.

Additionally, as the nodes for method calls, there are also `NODE_FCALL` and `NODE_VCALL`. `NODE_FCALL` is for the “`method(args)`” form, `NODE_VCALL` corresponds to method calls in the “`method`” form that is the same form as the local variables. `FCALL` and `VCALL` could actually be integrated into one, but because there’s no need to prepare arguments when it is `VCALL`, they are separated from each other only in order to save both times and memories for it.

Now, let's look at the handler of `NODE_CALL` in `rb_eval()`.

## ▼ `rb_eval()` – `NODE_CALL`

```
2745 case NODE_CALL:
2746 {
2747     VALUE recv;
2748     int argc; VALUE *argv; /* used in SETUP_ARGS */
2749     TMP_PROTECT;
2750
2751     BEGIN_CALLARGS;
2752     recv = rb_eval(self, node->nd_recv);
2753     SETUP_ARGS(node->nd_args);
2754     END_CALLARGS;
2755
2756     SET_CURRENT_SOURCE();
2757     result = rb_call(CLASS_OF(recv), recv, node->nd_mid, argc
2758 }
2759 break;
```

(eval.c)

The problems are probably the three macros, `BEGIN_CALLARGS` `SETUP_ARGS()` `END_CALLARGS`. It seems that `rb_eval()` is to evaluate the receiver and `rb_call()` is to invoke the method, we can roughly imagine that the evaluation of the arguments might be done in the three macros, but what is actually done? `BEGIN_CALLARGS` and `END_CALLARGS` are difficult to understand before talking about the iterators, so they are explained in the next chapter “Block”. Here, let's investigate only about `SETUP_ARGS()`.

## ■ `SETUP_ARGS()`

`SETUP_ARGS()` is the macro to evaluate the arguments of a method.

Inside of this macro, as the comment in the original program says, the variables named `argc` and `argv` are used, so they must be defined in advance. And because it uses `TMP_ALLOC()`, it must use `TMP_PROTECT` in advance. Therefore, something like the following is a boilerplate:

```
int argc; VALUE *argv; /* used in SETUP_ARGS */
TMP_PROTECT;

SETUP_ARGS(args_node);
```

`args_node` is (the node represents) the arguments of the method, turn it into an array of the values obtained by evaluating it, and store it in `argv`. Let's look at it:

## ▼ SETUP\_ARGS()

```
1780 #define SETUP_ARGS(anode) do {\ \
1781     NODE *n = anode; \
1782     if (!n) { \
1783         argc = 0; \
1784         argv = 0; \
1785     } \
1786     else if (nd_type(n) == NODE_ARRAY) { \
1787         argc=n->nd_alen; \
1788         if (argc > 0) { \
1789             int i; \
1790             n = anode; \
1791             argv = TMP_ALLOC(argc); \
1792             for (i=0;i<argc;i++) { \
1793                 argv[i] = rb_eval(self,n->nd_head); \
1794                 n=n->nd_next; \
1795             } \
1796         } \
1797     } \
1798     else { \
1799         no arguments
2000     } \
2001 }
```

```

1798                 argc = 0; \
1799                 argv = 0; \
1800             } \
1801         } \
1802     else { \
1803         VALUE args = rb_eval(self, n); \
1804         if (TYPE(args) != T_ARRAY) \
1805             args = rb_ary_to_ary(args); \
1806         argc = RARRAY(args)->len; \
1807         argv = ALLOCA_N(VALUE, argc); \
1808         MEMCPY(argv, RARRAY(args)->ptr, VALUE, argc); \
1809     } \
1810 } while (0)

(eval.c)

```

This is a bit long, but since it clearly branches in three ways, not so terrible actually. The meaning of each branch is written as comments.

We don't have to care about the case with no arguments, the two rest branches are doing similar things. Roughly speaking, what they are doing consists of three steps:

- allocate a space to store the arguments
- evaluate the expressions of the arguments
- copy the value into the variable space

If I write in the code (and tidy up a little), it becomes as follows.

```

/***** else if clause, argc!=0 *****/
int i;
n = anode;
argv = TMP_ALLLOC(argc); /* 1 */
for (i = 0; i < argc; i++) {
    argv[i] = rb_eval(self, n->nd_head); /* 2,3 */

```

```
    n = n->nd_next;
}

***** else clause *****/
VALUE args = rb_eval(self, n);                                /* 2 */
if (TYPE(args) != T_ARRAY)
    args = rb_ary_to_ary(args);
argc = RARRAY(args)->len;
argv = ALLOCA_N(VALUE, argc);                                /* 1 */
MEMCPY(argv, RARRAY(args)->ptr, VALUE, argc);    /* 3 */
```

`TMP_ALLOC()` is used in the `else if` side, but `ALLOCA_N()`, which is ordinary `alloca()`, is used in the `else` side. Why? Isn't it dangerous in the `C_ALLOCA` environment because `alloca()` is equivalent to `malloc()`?

The point is that “in the `else` side the values of arguments are also stored in `args`”. If I illustrate, it would look like Figure 1.

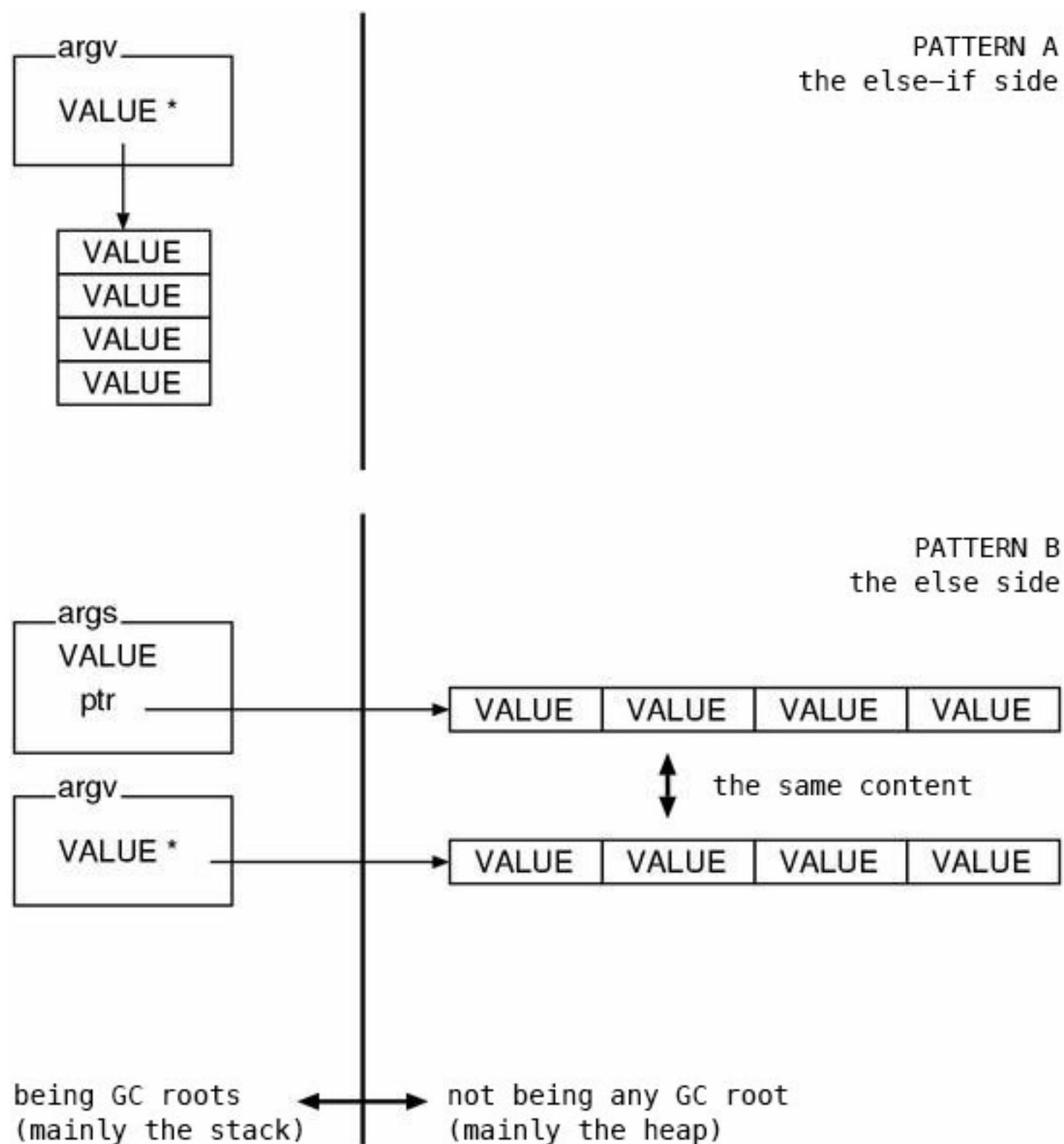


Figure 1: Being in the heap is all right.

If at least one `VALUE` is on the stack, others can be successively marked through it. This kind of `VALUE` plays a role to tie up the other `VALUES` to the stack like an anchor. Namely, it becomes “anchor `VALUE`”. In the `else` side, `args` is the anchor `VALUE`.

For your information, “anchor VALUE” is the word just coined now.

## rb\_call()

SETUP\_ARGS() is relatively off the track. Let's go back to the main line. The function to invoke a method, it is rb\_call(). In the original there're codes like raising exceptions when it could not find anything, as usual I'll skip all of them.

### ▼ rb\_call() (simplified)

```
static VALUE
rb_call(klass, recv, mid, argc, argv, scope)
    VALUE klass, recv;
    ID mid;
    int argc;
    const VALUE *argv;
    int scope;
{
    NODE *body;
    int noex;
    ID id = mid;
    struct cache_entry *ent;

    /* search over method cache */
    ent = cache + EXPR1(klass, mid);
    if (ent->mid == mid && ent->klass == klass) {
        /* cache hit */
        klass = ent->origin;
        id = ent->mid0;
        noex = ent->noex;
        body = ent->method;
    }
    else {
        /* cache miss, searching step-by-step */
        body = rb_get_method_body(&klass, &id, &noex);
    }
}
```

```
/* ... check the visibility ... */

return rb_call0(klass, recv, mid, id,
               argc, argv, body, noex & NOEX_UNDEF);
}
```

The basic way of searching methods was discussed in chapter 2: “Object”. It is following its superclasses and searching `m_tbl`. This is done by `search_method()`.

The principle is certainly this, but when it comes to the phase to execute actually, if it searches by looking up its hash many times for each method call, its speed would be too slow. To improve this, in `ruby`, once a method is called, it will be cached. If a method is called once, it’s often immediately called again. This is known as an experiential fact and this cache records the high hit rate.

What is looking up the cache is the first half of `rb_call()`. Only with

```
ent = cache + EXPR1(klass, mid);
```

this line, the cache is searched. We’ll examine its mechanism in detail later.

When any cache was not hit, the next `rb_get_method_body()` searches the class tree step-by-step and caches the result at the same time. Figure 2 shows the entire flow of searching.

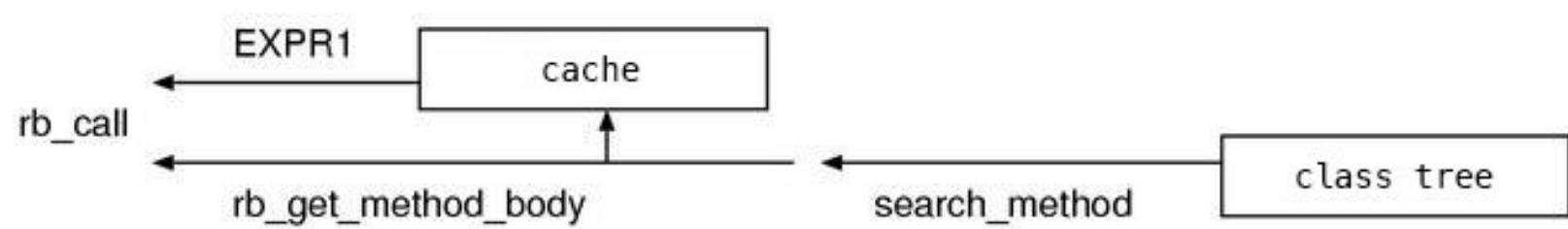


Figure 2: Method Search

## Method Cache

Next, let's examine the structure of the method cache in detail.

### Method Cache

```

180 #define CACHE_SIZE 0x800
181 #define CACHE_MASK 0x7ff
182 #define EXPR1(c,m) (((c)>>3)^(m))&CACHE_MASK)
183
184 struct cache_entry {           /* method hash table. */
185     ID mid;                   /* method's id */
186     ID mid0;                  /* method's original id */
187     VALUE klass;              /* receiver's class */
188     VALUE origin;             /* where method defined */
189     NODE *method;
190     int noex;
191 };
192
193 static struct cache_entry cache[CACHE_SIZE];
  
```

(eval.c)

If I describe the mechanism shortly, it is a hash table. I mentioned that the principle of the hash table is to convert a table search to an indexing of an array. Three things are necessary to accomplish: an array to store the data, a key, and a hash function.

First, the array here is an array of `struct cache_entry`. And the

method is uniquely determined by only the class and the method name, so these two become the key of the hash calculation. The rest is done by creating a hash function to generate the index (0x000 ~ 0x7ff) of the cache array form the key. It is `EXPR1()`. Among its arguments, `c` is the class object and `m` is the method name (`ID`). (Figure 3)

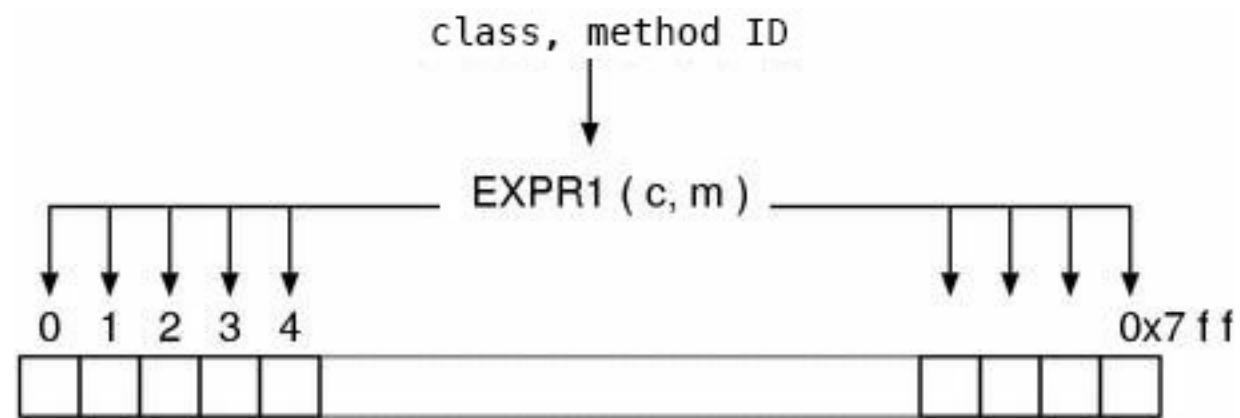


Figure 3: Method Cache

However, `EXPR1()` is not a perfect hash function or anything, so a different method can generate the same index coincidentally. But because this is nothing more than a cache, conflicts do not cause a problem. It just slows its performance down a little.

## The effect of Method Cache

By the way, how much effective is the method cache in actuality? We could not be convinced just by being said “it is known as ...”. Let’s measure by ourselves.

Type	Program	Hit Rate
generating <u>LALR</u> parser	racc ruby.y	99.9%

generating a mail thread a mailer	99.1%
generating a document rd2html rubyrefm.rd	97.8%

Surprisingly, in all of the three experiments the hit rate is more than 95%. This is awesome. Apparently, the effect of “it is known as ...” is outstanding.

## Invocation

---

### rb\_call0()

There have been many things and finally we arrived at the method invoking. However, this `rb_call0()` is huge. As it's more than 200 lines, it would come to 5,6 pages. If the whole part is laid out here, it would be disastrous. Let's look at it by dividing into small portions. Starting with the outline:

#### ▼ rb\_call0() (Outline)

```
4482 static VALUE
4483 rb_call0(klass, recv, id, oid, argc, argv, body, nosuper)
4484     VALUE klass, recv;
4485     ID id;
4486     ID oid;
4487     int argc;                      /* OK */
4488     VALUE *argv;                   /* OK */
4489     NODE *body;                   /* OK */
4490     int nosuper;
4491 {
4492     NODE *b2;                      /* OK */
4493     volatile VALUE result = Qnil;
```

```
4494     int itr;
4495     static int tick;
4496     TMP_PROTECT;
4497
4498     switch (ruby_iter->iter) {
4499         case ITER_PRE:
5000             itr = ITER_CUR;
5001             break;
5002         case ITER_CUR:
5003             default:
5004                 itr = ITER_NOT;
5005                 break;
5006     }
5007
5008     if (((++tick & 0xff) == 0) {
5009         CHECK_INTS; /* better than nothing */
5010         stack_check();
5011     }
5012     PUSH_ITER(itr);
5013     PUSH_FRAME();
5014
5015     ruby_frame->last_func = id;
5016     ruby_frame->orig_func = oid;
5017     ruby_frame->last_class = nosuper?0:klass;
5018     ruby_frame->self = recv;
5019     ruby_frame->argc = argc;
5020     ruby_frame->argv = argv;
5021
5022     switch (nd_type(body)) {
5023         /* ... main process ... */
5024
5025         default:
5026             rb_bug("unknown node type %d", nd_type(body));
5027             break;
5028     }
5029     POP_FRAME();
5030     POP_ITER();
5031     return result;
5032 }
```

(eval.c)

First, an `ITER` is pushed and whether or not the method is an iterator is finally fixed. As its value is used by the `PUSH_FRAME()` which comes immediately after it, `PUSH_ITER()` needs to appear beforehand. `PUSH_FRAME()` will be discussed soon.

And if I first describe about the “... main process ...” part, it branches based on the following node types and each branch does its invoking process.

<code>NODE_CFUNC</code>	methods defined in C
<code>NODE_IVAR</code>	<code>attr_reader</code>
<code>NODE_ATTRSET</code>	<code>attr_writer</code>
<code>NODE_SUPER</code>	<code>super</code>
<code>NODE_ZSUPER</code>	<code>super</code> without arguments
<code>NODE_DMETHOD</code>	invoke <code>UnboundMethod</code>
<code>NODE_BMETHOD</code>	invoke <code>Method</code>
<code>NODE_SCOPE</code>	methods defined in Ruby

Some of the above nodes are not explained in this book but not so important and could be ignored. The important things are only `NODE_CFUNC`, `NODE_SCOPE` and `NODE_ZSUPER`.

## ■ `PUSH_FRAME()`

### ▼ `PUSH_FRAME()` `POP_FRAME()`

```
536 #define PUSH_FRAME() do { \
537     struct FRAME _frame; \
538     _frame.prev = ruby_frame; \
539     _frame.tmp  = 0; \
540     _frame.node = ruby_current_node; \

```

```
541     _frame.iter = ruby_iter->iter;      \
542     _frame.cbase = ruby_frame->cbase;    \
543     _frame.argvc = 0;                   \
544     _frame.argv = 0;                   \
545     _frame.flags = FRAME_ALLOCA;      \
546     ruby_frame = &_frame

548 #define POP_FRAME()                      \
549     ruby_current_node = _frame.node;      \
550     ruby_frame = _frame.prev;            \
551 } while (0)
```

(eval.c)

First, we'd like to make sure the entire FRAME is allocated on the stack. This is identical to `module_setup()`. The rest is basically just doing ordinary initializations.

If I add one more description, the flag FRAME\_ALLOCA indicates the allocation method of the FRAME. FRAME\_ALLOCA obviously indicates “it is on the stack”.

## ■ `rb_call0()` – NODE\_CFUNC

A lot of things are written in this part of the original code, but most of them are related to `trace_func` and substantive code is only the following line:

### ▼ `rb_call0()` – NODE\_CFUNC (simplified)

```
case NODE_CFUNC:
    result = call_cfunc(body->nd_cfnc, recv, len, argc, argv);
    break;
```

Then, as for `call_cfunc()` ...

▼ `call_cfunc()` (simplified)

```
4394 static VALUE
4395 call_cfunc(func, recv, len, argc, argv)
4396     VALUE (*func)();
4397     VALUE recv;
4398     int len, argc;
4399     VALUE *argv;
4400 {
4401     if (len >= 0 && argc != len) {
4402         rb_raise(rb_eArgError, "wrong number of arguments(%d
4403                 argc, len);
4404     }
4405
4406     switch (len) {
4407         case -2:
4408             return (*func)(recv, rb_ary_new4(argc, argv));
4409             break;
4410         case -1:
4411             return (*func)(argc, argv, recv);
4412             break;
4413         case 0:
4414             return (*func)(recv);
4415             break;
4416         case 1:
4417             return (*func)(recv, argv[0]);
4418             break;
4419         case 2:
4420             return (*func)(recv, argv[0], argv[1]);
4421             break;
4422             :
4423             :
4424         default:
4425             rb_raise(rb_eArgError, "too many arguments(%d)", len
4426                     break;
4427     }
4428     return Qnil; /* not reached */
4429 }
```

As shown above, it branches based on the argument count. The maximum argument count is 15.

Note that neither `SCOPE` or `VARS` is pushed when it is `NODE_CFUNC`. It makes sense because a method defined in C does not use Ruby's local variables. But it simultaneously means that if the "current" local variables are accessed by `c`, they are actually the local variables of the previous `FRAME`. And in some places, say, `rb_svar` (`eval.c`), it is actually done.

## rb\_call0() – NODE\_SCOPE

`NODE_SCOPE` is to invoke a method defined in Ruby. This part forms the foundation of Ruby.

### ▼ rb\_call0() – NODE\_SCOPE (outline)

```

4568 case NODE_SCOPE:
4569 {
4570     int state;
4571     VALUE *local_vars; /* OK */
4572     NODE *saved_cref = 0;
4573
4574     PUSH_SCOPE();
4575
4576     /* (A) forward CREF */
4577     if (body->nd_rval) {
4578         saved_cref = ruby_cref;
4579         ruby_cref = (NODE*)body->nd_rval;
4580         ruby_frame->cbase = body->nd_rval;
4581     }

```

```
4581 /* (B) initialize ruby_scope->local_vars */
4582 if (body->nd_tbl) {
4583     local_vars = TMP_ALLOC(body->nd_tbl[0]+1);
4584     *local_vars++ = (VALUE)body;
4585     rb_mem_clear(local_vars, body->nd_tbl[0]);
4586     ruby_scope->local_tbl = body->nd_tbl;
4587     ruby_scope->local_vars = local_vars;
4588 }
4589 else {
4590     local_vars = ruby_scope->local_vars = 0;
4591     ruby_scope->local_tbl = 0;
4592 }
4593 b2 = body = body->nd_next;
4594
4595 PUSH_VARS();
4596 PUSH_TAG(Prot_FUNC);
4597
4598 if ((state = EXEC_TAG()) == 0) {
4599     NODE *node = 0;
4600     int i;
4601
4602     /* ..... (C) assign the arguments to the local variable */
4603
4604     if (trace_func) {
4605         call_trace_func("call", b2, recv, id, klass);
4606     }
4607     ruby_last_node = b2;
4608     /* (D) method body */
4609     result = rb_eval(recv, body);
4610 }
4611 else if (state == TAG_RETURN) { /* back via return */
4612     result = prot_tag->retval;
4613     state = 0;
4614 }
4615 POP_TAG();
4616 POP_VARS();
4617 POP_SCOPE();
4618 ruby_cref = saved_cref;
4619 if (trace_func) {
4620     call_trace_func("return", ruby_last_node, recv, id);
4621 }
4622 switch (state) {
4623     case 0:
```

```
4685         break;
4686
4687         case TAG_RETRY:
4688             if (rb_block_given_p()) {
4689                 JUMP_TAG(state);
4690             }
4691             /* fall through */
4692             default:
4693                 jump_tag_but_local_jump(state);
4694                 break;
4695             }
4696         }
4697     break;
```

(eval.c)

- (A) CREF forwarding, which was described at the section of constants in the previous chapter. In other words, `cbase` is transplanted to `FRAME` from the method entry.
- (B) The content here is completely identical to what is done at `module_setup()`. An array is allocated at `local_vars` of `SCOPE`. With this and `PUSH_SCOPE()` and `PUSH_VARS()`, the local variable scope creation is completed. After this, one can execute `rb_eval()` in the exactly same environment as the interior of the method.
- (C) This sets the received arguments to the parameter variables. The parameter variables are in essence identical to the local variables. Things such as the number of arguments are specified by `NODE_ARGS`, all it has to do is setting one by one. Details will be explained soon. And,
- (D) this executes the method body. Obviously, the receiver (`recv`)

becomes `self`. In other words, it becomes the first argument of `rb_eval()`. After all, the method is completely invoked.

## Set Parameters

Then, we'll examine the totally skipped part, which sets parameters. But before that, I'd like you to first check the syntax tree of the method again.

```
% ruby -rnodedump -e 'def m(a) nil end'  
NODE_SCOPE  
nd_rval = (null)  
nd_tbl = 3 [ _ ~ a ]  
nd_next:  
  NODE_BLOCK  
  nd_head:  
    NODE_ARGS  
    nd_cnt  = 1  
    nd_rest = -1  
    nd_opt = (null)  
  nd_next:  
    NODE_BLOCK  
    nd_head:  
      NODE_NEWLINE  
      nd_file = "-e"  
      nd_nth  = 1  
      nd_next:  
        NODE NIL  
  nd_next = (null)
```

`NODE_ARGS` is the node to specify the parameters of a method. I aggressively dumped several things, and it seemed its members are used as follows:

`nd_cnt` the number of the normal parameters

nd\_rest the variable ID of the rest parameter. -1 if the rest parameter is missing

nd\_opt holds the syntax tree to represent the default values of the option parameters. a list of NODE\_BLOCK

If one has this amount of the information, the local variable ID for each parameter variable can be uniquely determined. First, I mentioned that 0 and 1 are always `$_` and `$~`. In 2 and later, the necessary number of ordinary parameters are in line. The number of option parameters can be determined by the length of NODE\_BLOCK. Again next to them, the rest-parameter comes.

For example, if you write a definition as below,

```
def m(a, b, c = nil, *rest)
  lvar1 = nil
end
```

local variable IDs are assigned as follows.

0	1	2	3	4	5	6
<code>\$_</code>	<code>\$~</code>	<code>a</code>	<code>b</code>	<code>c</code>	<code>rest</code>	<code>lvar1</code>

Are you still with me? Taking this into considerations, let's look at the code.

## ▼ rb\_call0() – NODE\_SCOPE – assignments of arguments

```
4601  if (nd_type(body) == NODE_ARGS) { /* no body */
4602      node = body;                  /* NODE_ARGS */
4603      body = 0;                   /* the method body */
4604  }
```



```
4646                     argv++; argc--;
4647                     opt = opt->nd_next;
4648                 }
4649                 if (opt) {
4650                     rb_eval(recv, opt);
4651                 }
4652             }
4653             local_vars = ruby_scope->local_vars;
4654             if (node->nd_rest >= 0) { /* has rest parameter */
4655                 VALUE v;
4656
4657                 /* make an array of the remaining parameters and
4658                 if (argc > 0)
4659                     v = rb_ary_new4(argc, argv);
4660                 else
4661                     v = rb_ary_new2(0);
4662                 ruby_scope->local_vars[node->nd_rest] = v;
4663             }
4664         }
```

(eval.c)

Since comments are added more than before, you might be able to understand what it is doing by following step-by-step.

One thing I'd like to mention is about `argc` and `argv` of `ruby_frame`. It seems to be updated only when any rest-parameter does not exist, why is it only when any rest-parameter does not exist?

This point can be understood by thinking about the purpose of `argc` and `argv`. These members actually exist for `super` without arguments. It means the following form:

`super`

This `super` has a behavior to directly pass the parameters of the currently executing method. To enable to pass at the moment, the arguments are saved in `ruby_frame->argv`.

Going back to the previous story here, if there's a rest-parameter, passing the original parameters list somehow seems more convenient. If there's not, the one after option parameters are assigned seems better.

```
def m(a, b, *rest)
  super      # probably 5, 6, 7, 8 should be passed
end
m(5, 6, 7, 8)

def m(a, b = 6)
  super      # probably 5, 6 should be passed
end
m(5)
```

This is a question of which is better as a specification rather than “it must be”. If a method has a rest-parameter, it supposed to also have a rest-parameter at superclass. Thus, if the value after processed is passed, there's the high possibility of being inconvenient.

Now, I've said various things, but the story of method invocation is all done. The rest is, as the ending of this chapter, looking at the implementation of `super` which is just discussed.

What corresponds to `super` are `NODE_SUPER` and `NODE_ZSUPER`.

`NODE_SUPER` is ordinary super, and `NODE_ZSUPER` is super without arguments.

## ▼ rb\_eval() – NODE\_SUPER

```
2810                                     argc, argv, 3);  
2811             POP_ITER();  
2812         }  
2813         break;  
  
(eval.c)
```

For super without arguments, I said that `ruby_frame->argv` is directly used as arguments, this is directly shown at (B).

(C) just before calling `rb_call()`, doing `PUSH_ITER()`. This is also what cannot be explained in detail, but in this way the block passed to the current method can be handed over to the next method (meaning, the method of superclass that is going to be called).

And finally, (A) when `ruby_frame->last_class` is 0, calling super seems forbidden. Since the error message says “must be enabled by `rb_enable_super()`”, it seems it becomes callable by calling `rb_enable_super()`.

((errata: The error message “must be enabled by `rb_enable_super()`” exists not in this list but in `rb_call_super()`.))

Why?

First, If we investigate in what kind of situation `last_class` becomes 0, it seems that it is while executing the method whose substance is defined in C (`NODE_CFUNC`). Moreover, it is the same when doing alias or replacing such method.

I've understood until there, but even though reading source codes, I couldn't understand the subsequents of them. Because I couldn't,

I searched “rb\_enable\_super” over the ruby’s mailing list archives and found it. According to that mail, the situation looks like as follows:

For example, there’s a method named `String.new`. Of course, this is a method to create a string. `String.new` creates a struct of `T_STRING`. Therefore, you can expect that the receiver is always of `T_STRING` when writing an instance methods of `String`.

Then, `super` of `String.new` is `Object.new`. `Object.new` create a struct of `T_OBJECT`. What happens if `String.new` is replaced by new definition and `super` is called?

```
def String.new
  super
end
```

As a consequence, an object whose struct is of `T_OBJECT` but whose class is `String` is created. However, a method of `String` is written with expectation of a struct of `T_STRING`, so naturally it downs.

How can we avoid this? The answer is to forbid to call any method expecting a struct of a different struct type. But the information of “expecting struct type” is not attached to method, and also not to class. For example, if there’s a way to obtain `T_STRING` from `String` class, it can be checked before calling, but currently we can’t do such thing. Therefore, as the second-best plan, “super from methods defined in C is forbidden” is defined. In this way, if the layer of methods at C level is precisely created, it cannot be got

down at least. And, when the case is “It’s absolutely safe, so allow super”, `super` can be enabled by calling `rb_enable_super()`.

In short, the heart of the problem is miss match of struct types. This is the same as the problem that occurs at the allocation framework.

Then, how to solve this is to solve the root of the problem that “the class does not know the struct-type of the instance”. But, in order to resolve this, at least new API is necessary, and if doing more deeply, compatibility will be lost. Therefore, for the time being, the final solution has not decided yet.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](#)

# Ruby Hacking Guide

# Chapter 16: Blocks

## Iterator

---

In this chapter, `BLOCK`, which is the last big name among the seven Ruby stacks, comes in. After finishing this, the internal state of the evaluator is virtually understood.

### ▀ The Whole Picture

What is the mechanism of iterators? First, let's think about a small program as below:

#### ▼ The Source Program

```
iter_method() do
  9  # a mark to find this block
end
```

Let's check the terms just in case. As for this program, `iter_method` is an iterator method, `do ~ end` is an iterator block. Here is the syntax tree of this program being dumped.

#### ▼ Its Syntax Tree

```
NODE_ITER
nd_iter:
  NODE_FCALL
  nd_mid = 9617 (iter_method)
  nd_args = (null)
nd_var = (null)
nd_body:
  NODE_LIT
  nd_lit = 9:Fixnum
```

Looking for the block by using the 9 written in the iterator block as a trace, we can understand that `NODE_ITER` seems to represent the iterator block. And `NODE_FCALL` which calls `iter_method` is at the “below” of that `NODE_ITER`. In other words, the node of iterator block appears earlier than the call of the iterator method. This means, before calling an iterator method, a block is pushed at another node.

And checking by following the flow of code with debugger, I found that the invocation of an iterator is separated into 3 steps: `NODE_ITER` `NODE_CALL` and `NODE_YIELD`. This means,

1. push a block (`NODE_ITER`)
2. call the method which is an iterator (`NODE_CALL`)
3. yield (`NODE_YIELD`)

that's all.

## Push a block

First, let's start with the first step, that is `NODE_ITER`, which is the

node to push a block.

## ▼ rb\_eval() – NODE\_ITER (simplified)

```
case NODE_ITER:
{
    iter_retry:
        PUSH_TAG(Prot_FUNC);
        PUSH_BLOCK(node->nd_var, node->nd_body);

        state = EXEC_TAG();
        if (state == 0) {
            PUSH_ITER(ITER_PRE);
            result = rb_eval(self, node->nd_iter);
            POP_ITER();
        }
        else if (_block.tag->dst == state) {
            state &= TAG_MASK;
            if (state == TAG_RETURN || state == TAG_BREAK) {
                result = prot_tag->retval;
            }
        }
        POP_BLOCK();
        POP_TAG();
        switch (state) {
            case 0:
                break;

            case TAG_RETRY:
                goto iter_retry;

            case TAG_BREAK:
                break;

            case TAG_RETURN:
                return_value(result);
                /* fall through */
            default:
                JUMP_TAG(state);
        }
}
```

```
break;
```

Since the original code contains the support of the `for` statement, it is deleted. After removing the code relating to tags, there are only push/pop of `ITER` and `BLOCK` left. Because the rest is ordinarily doing `rb_eval()` with `NODE_FCALL`, these `ITER` and `BLOCK` are the necessary conditions to turn a method into an iterator.

The necessity of pushing `BLOCK` is fairly reasonable, but what's `ITER` for? Actually, to think about the meaning of `ITER`, you need to think from the viewpoint of the side that uses `BLOCK`.

For example, suppose a method is just called. And `ruby_block` exists. But since `BLOCK` is pushed regardless of the break of method calls, the existence of a block does not mean the block is pushed for that method. It's possible that the block is pushed for the previous method. (Figure 1)

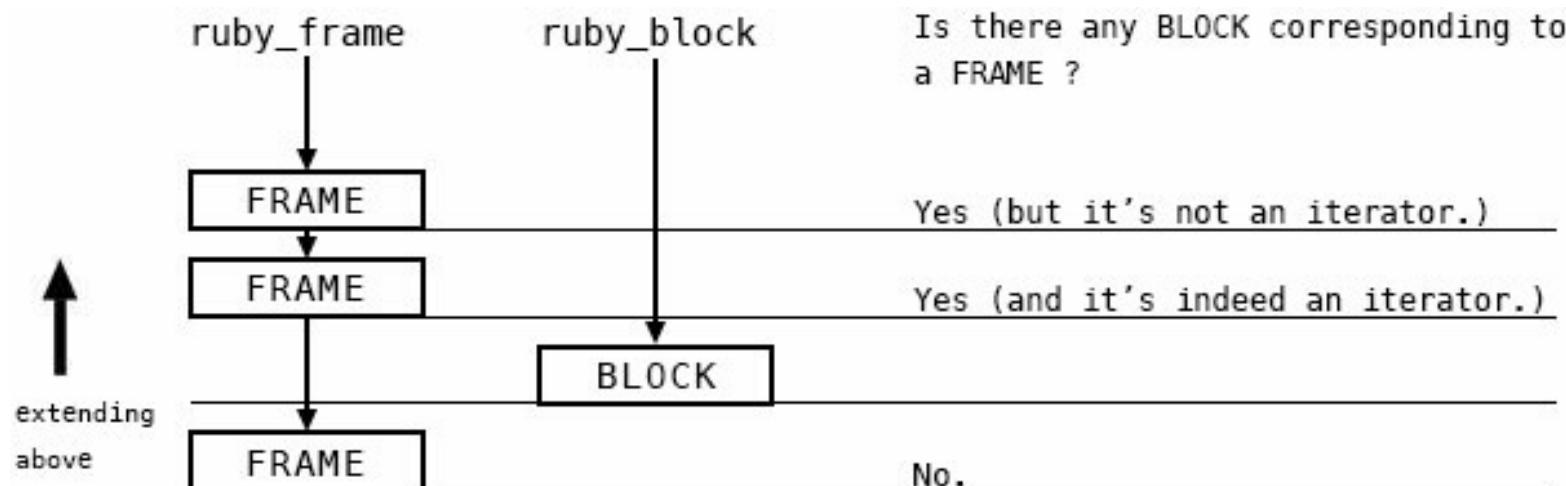


Figure 1: no one-to-one correspondence between `FRAME` and `BLOCK`

So, in order to determine for which method the block is pushed,

ITER is used. BLOCK is not pushed for each FRAME because pushing BLOCK is a little heavy. How much heavy is, let's check it in practice.

## PUSH\_BLOCK()

The argument of PUSH\_BLOCK() is (the syntax tree of) the block parameter and the block body.

### ▼ PUSH\_BLOCK() POP\_BLOCK()

```
592 #define PUSH_BLOCK(v,b) do { \
593     struct BLOCK _block; \
594     _block.tag = new_blktag(); \
595     _block.var = v; \
596     _block.body = b; \
597     _block.self = self; \
598     _block.frame = *ruby_frame; \
599     _block.klass = ruby_class; \
600     _block.frame.node = ruby_current_node; \
601     _block.scope = ruby_scope; \
602     _block.prev = ruby_block; \
603     _block.iter = ruby_iter->iter; \
604     _block.vmode = scope_vmode; \
605     _block.flags = BLOCK_D_SCOPE; \
606     _block.dyna_vars = ruby_dyna_vars; \
607     _block.wrapper = ruby_wrapper; \
608     ruby_block = &_block \
610 #define POP_BLOCK() \
611     if (_block.tag->flags & (BLOCK_DYNAMIC)) \
612         _block.tag->flags |= BLOCK_ORPHAN; \
613     else if (!(_block.scope->flags & SCOPE_DONT_RECYCLE)) \
614         rb_gc_force_recycle((VALUE)_block.tag); \
615     ruby_block = _block.prev; \
616 } while (0)
```

Let's make sure that a `BLOCK` is “the snapshot of the environment of the moment of creation”. As a proof of it, except for `CREF` and `BLOCK`, the six stack frames are saved. `CREF` can be substituted by `ruby_frame->cbase`, there's no need to push.

And, I'd like to check the three points about the mechanism of push. `BLOCK` is fully allocated on the stack. `BLOCK` contains the full copy of `FRAME` at the moment. `BLOCK` is different from the other many stack frame structs in having the pointer to the previous `BLOCK` (`prev`).

The flags used in various ways at `POP_BLOCK()` is not explained now because it can only be understood after seeing the implementation of `Proc` later.

And the talk is about “BLOCK is heavy”, certainly it seems a little heavy. When looking inside of `new_blktag()`, we can see it does `malloc()` and store plenty of members. But let’s defer the final judge until after looking at and comparing with `PUSH_ITER()`.

## PUSH\_ITER()

▼ PUSH ITER() POP ITER()

```
773 #define PUSH_ITER(i) do {  
774     struct iter _iter;  
775     _iter.prev = ruby_iter;  
776     _iter.iter = (i);  
777     ruby_iter = &_iter
```

```
779 #define POP_ITER()  
780     ruby_iter = _iter.prev;  
781 } while (0)  
  
(eval.c)
```

On the contrary, this is apparently light. It only uses the stack space and has only two members. Even if this is pushed for each FRAME, it would probably matter little.

## Iterator Method Call

After pushing a block, the next thing is to call an iterator method (a method which is an iterator). There also needs a little machinery. Do you remember that there's a code to modify the value of `ruby_iter` at the beginning of `rb_call0`? Here.

### ▼ `rb_call0()` – moving to `ITER_CUR`

```
4498     switch (ruby_iter->iter) {  
4499         case ITER_PRE:  
4500             itr = ITER_CUR;  
4501             break;  
4502         case ITER_CUR:  
4503             default:  
4504                 itr = ITER_NOT;  
4505                 break;  
4506     }  
  
(eval.c)
```

Since `ITER_PRE` is pushed previously at `NODE_TER`, this code makes `ruby_iter` `ITER_CUR`. At this moment, a method finally “becomes” an

iterator. Figure 2 shows the state of the stacks.

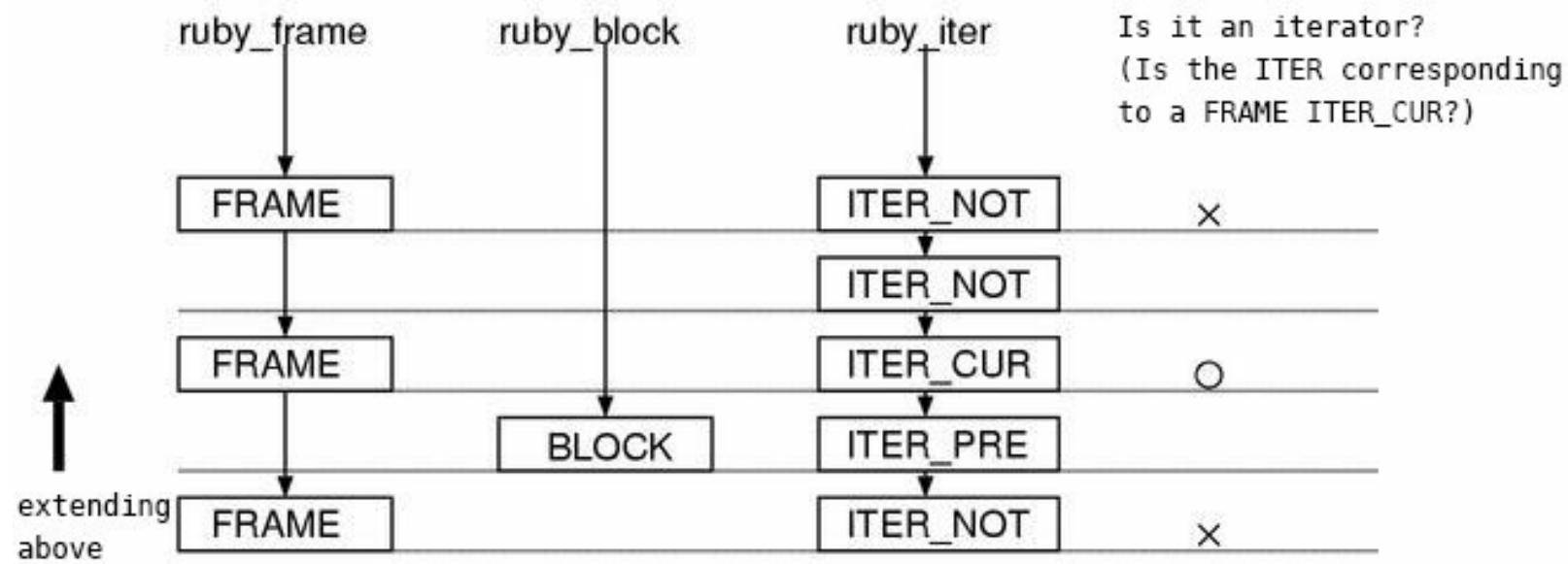


Figure 2: the state of the Ruby stacks on an iterator call.

The possible value of `ruby_iter` is not the one of two boolean values (for that method or not), but one of three steps because there's a little gap between the timings when pushing a block and invoking an iterator method. For example, there's the evaluation of the arguments of an iterator method. Since it's possible that it contains method calls inside it, there's the possibility that one of that methods mistakenly thinks that the just pushed block is for itself and uses it during the evaluation. Therefore, the timing when a method becomes an iterator, this means turning into `ITER_CUR`, has to be the place inside of `rb_call()` that is just before finishing the invocation.

## ▼ the processing order

```
method(arg) { block } # push a block
method(arg) { block } # evaluate the arguments
```

```
method(arg) { block } # a method call
```

For example, in the last chapter “Method”, there’s a macro named BEGIN\_CALLARGS at a handler of NODE\_CALL. This is where making use of the third step ITER. Let’s go back a little and try to see it.

## BEGIN\_CALLARGS END\_CALLARGS

### ▼ BEGIN\_CALLARGS END\_CALLARGS

```
1812 #define BEGIN_CALLARGS do {\  
1813     struct BLOCK *tmp_block = ruby_block;\  
1814     if (ruby_iter->iter == ITER_PRE) {\  
1815         ruby_block = ruby_block->prev;\  
1816     }\  
1817     PUSH_ITER(ITER_NOT)  
  
1819 #define END_CALLARGS \  
1820     ruby_block = tmp_block;\  
1821     POP_ITER();\  
1822 } while (0)  
  
(eval.c)
```

When `ruby_iter` is `ITER_PRE`, a `ruby_block` is set aside. This code is important, for instance, in the below case:

```
obj.m1 { yield }.m2 { nil }
```

The evaluation order of this expression is:

1. push the block of `m2`
2. push the block of `m1`

3. call the method `m1`
4. call the method `m2`

Therefore, if there was not `BEGIN_CALLARGS`, `m1` will call the block of `m2`.

And, if there's one more iterator connected, the number of `BEGIN_CALLARGS` increases at the same time in this case, so there's no problem.

## Block Invocation

The third phase of iterator invocation, it means the last phase, is block invocation.

### ▼ `rb_eval() – NODE_YIELD`

```
2579     case NODE_YIELD:
260      if (node->nd_stts) {
261          result = avalue_to_yvalue(rb_eval(self, node->nd
262      }
263      else {
264          result = Qundef; /* no arg */
265      }
266      SET_CURRENT_SOURCE();
267      result = rb_yield_0(result, 0, 0, 0);
268      break;

(eval.c)
```

`nd_stts` is the parameter of `yield`. `avalue_to_yvalue()` was mentioned a little at the multiple assignments, but you can ignore this.

((errata: actually, it was not mentioned. You can ignore this anyway.))

The heart of the behavior is not this but `rb_yield_0()`. Since this function is also very long, I show the code after extremely simplifying it. Most of the methods to simplify are previously used.

- cut the codes relating to `trace_func`.
- cut errors
- cut the codes exist only to prevent from GC
- As the same as `massign()`, there's the parameter `pcall`. This parameter is to change the level of restriction of the parameter check, so not important here. Therefore, assume `pcal=0` and perform constant foldings.

And this time, I turn on the “optimize for readability option” as follows.

- when a code branching has equivalent kind of branches, leave the main one and cut the rest.
- if a condition is true/false in the almost all case, assume it is true/false.
- assume there's no tag jump occurs, delete all codes relating to tag.

If things are done until this, it becomes very shorter.

## ▼ `rb_yield_0()` (simplified)

```
static VALUE
```

```
rb_yield_0(val, self, klass, /* pcall=0 */)
    VALUE val, self, klass;
{
    volatile VALUE result = Qnil;
    volatile VALUE old_cref;
    volatile VALUE old_wrapper;
    struct BLOCK * volatile block;
    struct SCOPE * volatile old_scope;
    struct FRAME frame;
    int state;

    PUSH_VARS();
    PUSH_CLASS();
    block = ruby_block;
    frame = block->frame;
    frame.prev = ruby_frame;
    ruby_frame = &(frame);
    old_cref = (VALUE)ruby_cref;
    ruby_cref = (NODE*)ruby_frame->cbase;
    old_wrapper = ruby_wrapper;
    ruby_wrapper = block->wrapper;
    old_scope = ruby_scope;
    ruby_scope = block->scope;
    ruby_block = block->prev;
    ruby_dyna_vars = new_dvar(0, 0, block->dyna_vars);
    ruby_class = block->klass;
    self = block->self;

    /* set the block arguments */
    massign(self, block->var, val, pcall);

    PUSH_ITER(block->iter);
    /* execute the block body */
    result = rb_eval(self, block->body);
    POP_ITER();

    POP_CLASS();
    /* .....collect ruby_dyna_vars..... */
    POP_VARS();
    ruby_block = block;
    ruby_frame = ruby_frame->prev;
    ruby_cref = (NODE*)old_cref;
    ruby_wrapper = old_wrapper;
```

```

    ruby_scope = old_scope;
    return result;
}

```

As you can see, the most stack frames are replaced with what saved at `ruby_block`. Things to simple save/restore are easy to understand, so let's see the handling of the other frames we need to be careful about.

## FRAME

```

struct FRAME frame;

frame = block->frame;      /* copy the entire struct */
frame.prev = ruby_frame;   /* by these two lines..... */
ruby_frame = &(frame);    /* .....frame is pushed */

```

Differing from the other frames, a `FRAME` is not used in the saved state, but a new `FRAME` is created by duplicating. This would look like Figure 3.

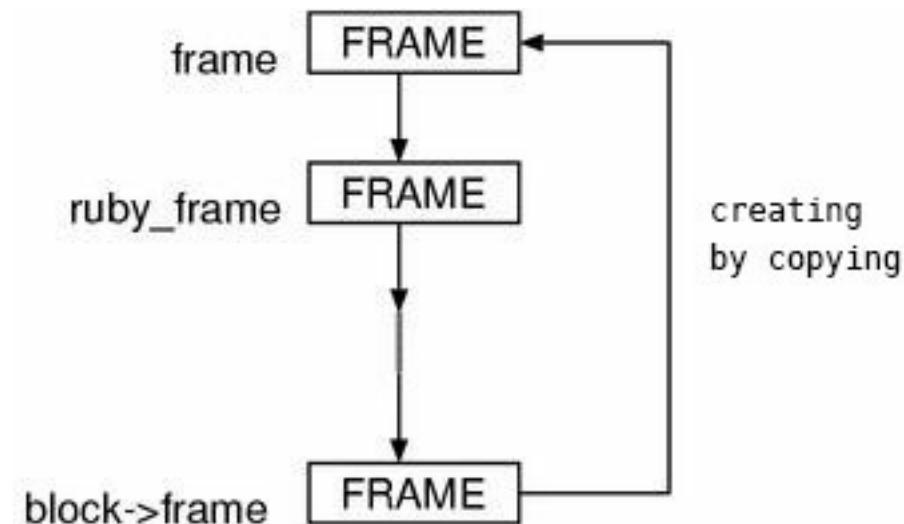


Figure 3: push a copied frame

As we've seen the code until here, it seems that FRAME will never be "reused". When pushing FRAME, a new FRAME will always be created.

## BLOCK

```
block = ruby_block;
:
ruby_block = block->prev;
:
ruby_block = block;
```

What is the most mysterious is this behavior of BLOCK. We can't easily understand whether it is saving or popping. It's comprehensible that the first statement and the third statement are as a pair, and the state will be eventually back. However, what is the consequence of the second statement?

To put the consequence of I've pondered a lot in one phrase, "going back to the `ruby_block` of at the moment when pushing the block". An iterator is, in short, the syntax to go back to the previous frame. Therefore, all we have to do is turning the state of the stack frame into what was at the moment when creating the block. And, the value of `ruby_block` at the moment when creating the block is, it seems certain that it was `block->prev`. Therefore, it is contained in `prev`.

Additionally, for the question "is it no problem to assume what invoked is always the top of `ruby_block`? ", there's no choice but saying "as the `rb_yield_0` side, you can assume so". To push the

block which should be invoked on the top of the `ruby_block` is the work of the side to prepare the block, and not the work of `rb_yield_0`.

An example of it is `BEGIN_CALLARGS` which was discussed in the previous chapter. When an iterator call cascades, the two blocks are pushed and the top of the stack will be the block which should not be used. Therefore, it is purposefully checked and set aside.

## VARS

Come to think of it, I think we have not looked the contents of `PUSH_VARS()` and `POP_VARS()` yet. Let's see them here.

### ▼ `PUSH_VARS()` `POP_VARS()`

```
619 #define PUSH_VARS() do { \
620     struct RVarmap * volatile _old; \
621     _old = ruby_dyna_vars; \
622     ruby_dyna_vars = 0

624 #define POP_VARS() \
625     if (_old && (ruby_scope->flags & SCOPE_DONT_RECYCLE)) { \
626         if (RBASIC(_old)->flags) /* if were not recycled */ \
627             FL_SET(_old, DVAR_DONT_RECYCLE); \
628     } \
629     ruby_dyna_vars = _old;
630 } while (0)
```

`(eval.c)`

This is also not pushing a new struct, to say “set aside/restore” is closer. In practice, in `rb_yield_0`, `PUSH_VARS()` is used only to set

aside the value. What actually prepares `ruby_dyna_vars` is this line.

```
ruby_dyna_vars = new_dvar(0, 0, block->dyna_vars);
```

This takes the `dyna_vars` saved in `BLOCK` and sets it. An entry is attached at the same time. I'd like you to recall the description of the structure of `ruby_dyna_vars` in Part 2, it said the `RVarmap` whose `id` is 0 such as the one created here is used as the break between block scopes.

However, in fact, between the parser and the evaluator, the form of the link stored in `ruby_dyna_vars` is slightly different. Let's look at the `dvar_asgn_curr()` function, which assigns a block local variable at the current block.

### ▼ `dvar_asgn_curr()`

```
737 static inline void
738 dvar_asgn_curr(id, value)
739     ID id;
740     VALUE value;
741 {
742     dvar_asgn_internal(id, value, 1);
743 }

699 static void
700 dvar_asgn_internal(id, value, curr)
701     ID id;
702     VALUE value;
703     int curr;
704 {
705     int n = 0;
706     struct RVarmap *vars = ruby_dyna_vars;
707 }
```

```
708     while (vars) {
709         if (curr && vars->id == 0) {
710             /* first null is a dvar header */
711             n++;
712             if (n == 2) break;
713         }
714         if (vars->id == id) {
715             vars->val = value;
716             return;
717         }
718         vars = vars->next;
719     }
720     if (!ruby_dyna_vars) {
721         ruby_dyna_vars = new_dvar(id, value, 0);
722     }
723     else {
724         vars = new_dvar(id, value, ruby_dyna_vars->next);
725         ruby_dyna_vars->next = vars;
726     }
727 }
```

(eval.c)

The last `if` statement is to add a variable. If we focus on there, we can see a link is always pushed in at the “next” to `ruby_dyna_vars`. This means, it would look like Figure 4.

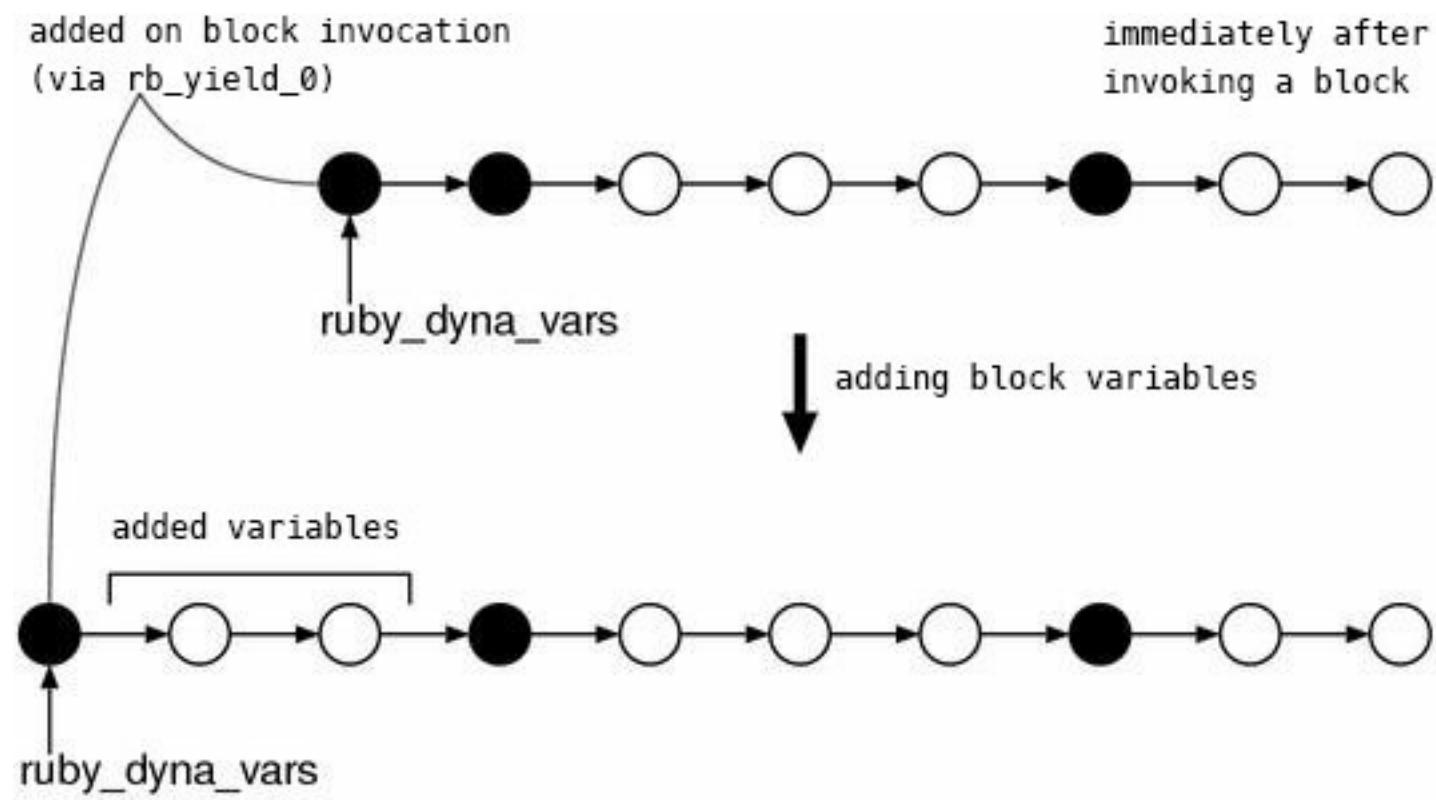


Figure 4: the structure of `ruby_dyna_vars`

This differs from the case of the parser in one point: the headers (`id=0`) to indicate the breaks of scopes are attached before the links. If a header is attached after the links, the first one of the scope cannot be inserted properly. (Figure 5)  
 ((errata: It was described that `ruby_dyna_vars` of the evaluator always forms a single straight link. But according to the errata, it was wrong. That part and relevant descriptions are removed.))

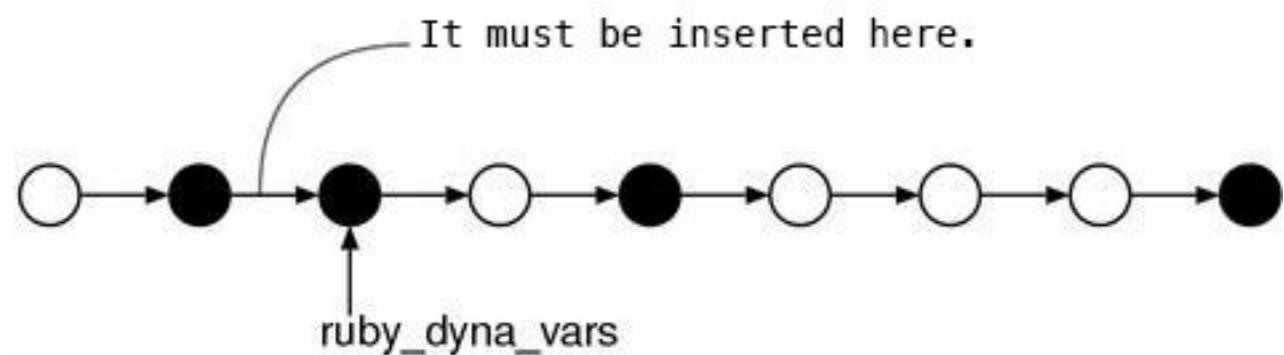


Figure 5: The entry cannot be inserted properly.

# Target Specified Jump

The code relates to jump tags are omitted in the previously shown code, but there's an effort that we've never seen before in the jump of `rb_yield_0`. Why is the effort necessary? I'll tell the reason in advance. I'd like you to see the below program:

```
[0].each do
  break
end
# the place to reach by break
```

like this way, in the case when doing `break` from inside of a block, it is necessary to get out of the block and go to the method that pushed the block. What does it actually mean? Let's think by looking at the (dynamic) call graph when invoking an iterator.

```
rb_eval(NODE_ITER)
  rb_eval(NODE_CALL)
    rb_eval(NODE_YIELD)
      rb_yield_0
        rb_eval(NODE_BREAK) .... throw(TAG_BREAK)
          .... catch(TAG_BREAK)
          .... catch(TAG_BREAK)
```

Since what pushed the block is `NODE_ITER`, it should go back to a `NODE_ITER` when doing `break`. However, `NODE_CALL` is waiting for `TAG_BREAK` before `NODE_ITER`, in order to turn a `break` over methods into an error. This is a problem. We need to somehow find a way to go straight back to a `NODE_ITER`.

And actually, “going back to a `NODE_ITER`” will still be a problem. If iterators are nesting, there could be multiple `NODE_ITERS`, thus the

one corresponds to the current block is not always the first NODE\_ITER. In other words, we need to restrict only “the NODE\_ITER that pushed the currently being invoked block”

Then, let's see how this is resolved.

▼ rb\_yield\_0() – the parts relates to tags

```
3826      PUSH_TAG(prot_none);
3827      if ((state = EXEC_TAG()) == 0) {
3828          /* .....evaluate the body..... */
3829      }
3830      else {
3831          switch (state) {
3832              case TAG_REDO:
3833                  state = 0;
3834                  CHECK_INTS;
3835                  goto redo;
3836              case TAG_NEXT:
3837                  state = 0;
3838                  result = prot_tag->retval;
3839                  break;
3840              case TAG_BREAK:
3841              case TAG_RETURN:
3842                  state |= (serial++ << 8);
3843                  state |= 0x10;
3844                  block->tag->dst = state;
3845                  break;
3846              default:
3847                  break;
3848          }
3849      }
3850      POP_TAG();
```

(eval.c)

The parts of TAG\_BREAK and TAG\_RETURN are crucial.

First, `serial` is a static variable of `rb_yield_0()`, its value will be different every time calling `rb_yield_0`. “`serial`” is the serial of “serial number”.

The reason why left shifting by 8 bits seems in order to avoid overlapping the values of `TAG_xxxx`. `TAG_xxxx` is in the range between `0x1 ~ 0x8`, 4 bits are enough. And, the bit-or of `0x10` seems to prevent `serial` from overflow. In 32-bit machine, `serial` can use only 24 bits (only 16 million times), recent machine can let it overflow within less than 10 seconds. If this happens, the top 24 bits become all 0 in line. Therefore, if `0x10` did not exist, state would be the same value as `TAG_xxxx` (See also Figure 6).

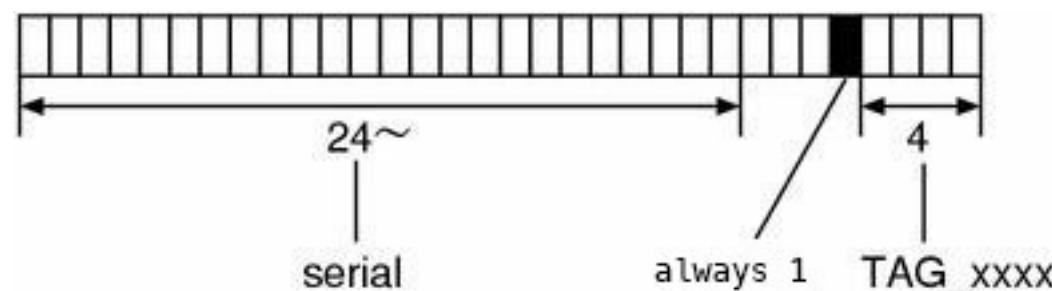


Figure 6: block->tag->dst

Now, `tag->dst` became the value which differs from `TAG_xxxx` and is unique for each call. In this situation, because an ordinary switch as previous ones cannot receive it, the side to stop jumps should need efforts to some extent. The place where making an effort is this place of `rb_eval:NODE_ITER`:

▼ `rb_eval() - NODE_ITER (to stop jumps)`

```

case NODE_ITER:
{
    state = EXEC_TAG();
    if (state == 0) {
        /* ..... invoke an iterator ..... */
    }
    else if (_block.tag->dst == state) {
        state &= TAG_MASK;
        if (state == TAG_RETURN || state == TAG_BREAK) {
            result = prot_tag->retval;
        }
    }
}

```

In corresponding `NODE_ITER` and `rb_yield_0`, `block` should point to the same thing, so `tag->dst` which was set at `rb_yield_0` comes in here. Because of this, only the corresponding `NODE_ITER` can properly stop the jump.

## Check of a block

Whether or not a currently being evaluated method is an iterator, in other words, whether there's a block, can be checked by `rb_block_given_p()`. After reading the above all, we can tell its implementation.

### ▼ `rb_block_given_p()`

```

3726 int
3727 rb_block_given_p()
3728 {
3729     if (ruby_frame->iter && ruby_block)
3730         return Qtrue;
3731     return Qfalse;

```

```
3732 }
```

```
(eval.c)
```

I think there's no problem. What I'd like to talk about this time is actually another function to check, it is `rb_f_block_given_p()`.

## ▼ `rb_f_block_given_p()`

```
3740 static VALUE
3741 rb_f_block_given_p()
3742 {
3743     if (ruby_frame->prev && ruby_frame->prev->iter && ruby_b
3744         return Qtrue;
3745     return Qfalse;
3746 }
```

```
(eval.c)
```

This is the substance of Ruby's `block_given?`. In comparison to `rb_block_given_p()`, this is different in checking the `prev` of `ruby_frame`. Why is this?

Thinking about the mechanism to push a block, to check the current `ruby_frame` like `rb_block_given_p()` is right. But when calling `block_given?` from Ruby-level, since `block_given?` itself is a method, an extra `FRAME` is pushed. Hence, we need to check the previous one.

To describe a `Proc` object from the viewpoint of implementing, it is “a `BLOCK` which can be bring out to Ruby level”. Being able to bring out to Ruby level means having more latitude, but it also means when and where it will be used becomes completely unpredictable. Focusing on how the influence of this fact is, let’s look at the implementation.

## Proc object creation

A `Proc` object is created with `Proc.new`. Its substance is `proc_new()`.

### ▼ `proc_new()`

```
6418 static VALUE
6419 proc_new(klass)
6420     VALUE klass;
6421 {
6422     volatile VALUE proc;
6423     struct BLOCK *data, *p;
6424     struct RVarmap *vars;
6425
6426     if (!rb_block_given_p() && !rb_f_block_given_p()) {
6427         rb_raise(rb_eArgError,
6428                 "tried to create Proc object without a block");
6429     }
6430
6431     /* (A) allocate both struct RData and struct BLOCK */
6432     proc = Data_Make_Struct(klass, struct BLOCK,
6433                             blk_mark, blk_free, data);
6434     *data = *ruby_block;
6435
6436     data->orig_thread = rb_thread_current();
6437     data->wrapper = ruby_wrapper;
6438     data->iter = data->prev?Qtrue:Qfalse;
6439     /* (B) the essential initialization is finished by here
6440        frame_dup(&data->frame);
```

```

6437     if (data->iter) {
6438         blk_copy_prev(data);
6439     }
6440     else {
6441         data->prev = 0;
6442     }
6443     data->flags |= BLOCK_DYNAMIC;
6444     data->tag->flags |= BLOCK_DYNAMIC;
6445
6446     for (p = data; p; p = p->prev) {
6447         for (vars = p->dyna_vars; vars; vars = vars->next) {
6448             if (FL_TEST(vars, DVAR_DONT_RECYCLE)) break;
6449             FL_SET(vars, DVAR_DONT_RECYCLE);
6450         }
6451     }
6452     scope_dup(data->scope);
6453     proc_save_safe_level(proc);
6454
6455     return proc;
6456 }

```

(eval.c)

The creation of a `Proc` object itself is unexpectedly simple. Between (A) and (B), a space for an `Proc` object is allocated and its initialization completes. `Data_Make_Struct()` is a simple macro that does both `malloc()` and `Data_Wrap_Struct()` at the same time.

The problems exist after that:

- `frame_dup()`
- `blk_copy_prev()`
- `FL_SET(vars, DVAR_DONT_RECYCLE)`
- `scope_dup()`

These four have the same purposes. They are:

- move all of what were put on the machine stack to the heap.
- prevent from collecting even if after POP

Here, “all” means the all things including `prev`. For the all stack frames pushed there, it duplicates each frame by doing `malloc()` and copying. `VARS` is usually forced to be collected by `rb_gc_force_recycle()` at the same moment of POP, but this behavior is stopped by setting the `DVAR_DONT_RECYCLE` flag. And so on. Really extreme things are done.

Why are these extreme things necessary? This is because, unlike iterator blocks, a `Proc` can persist longer than the method that created it. And the end of a method means the things allocated on the machine stack such as `FRAME`, `ITER`, and `local_vars` of `SCOPE` are invalidated. It’s easy to predict what the consequence of using the invalidated memories. (An example answer: it becomes troublesome).

I tried to contrive a way to at least use the same `FRAME` from multiple `Proc`, but since there are the places such as `old_frame` where setting aside the pointers to the local variables, it does not seem going well. If it requires a lot efforts in anyway, another effort, say, allocating all of them with `malloc()` from the frist place, seems better to give it a try.

Anyway, I sentimentally think that it’s surprising that it runs with that speed even though doing these extreme things. Indeed, it has become a good time.

# Floating Frame

Previously, I mentioned it just in one phrase “duplicate all frames”, but since that was unclear, let’s look at more details. The points are the next two:

- How to duplicate all
- Why all of them are duplicated

Then first, let’s start with the summary of how each stack frame is saved.

**Frame location has prev pointer?**

FRAME	stack	yes
SCOPE	stack	no
local_tbl	heap	
local_vars	stack	
VARS	heap	no
BLOCK	stack	yes

CLASS CREF ITER are not necessary this time. Since CLASS is a general Ruby object, `rb_gc_force_recycle()` is not called with it even by mistake (it’s impossible) and both CREF and ITER becomes unnecessary after storing its values at the moment in FRAME. The four frames in the above table are important because these will be modified or referred to multiple times later. The rest three will not.

Then, this talk moves to how to duplicate all. I said “how”, but it does not about such as “by `malloc()`”. The problem is how to

duplicate “all”. It is because, here I’d like you to see the above table, there are some frames without any `prev` pointer. In other words, we cannot follow links. In this situation, how can we duplicate all?

A fairly clever technique used to counter this. Let’s take `SCOPE` as an example. A function named `scope_dup()` is used previously in order to duplicate `SCOPE`, so let’s see it first.

## ▼ `scope_dup()` only the beginning

```
6187 static void
6188 scope_dup(scope)
6189     struct SCOPE *scope;
6190 {
6191     ID *tbl;
6192     VALUE *vars;
6193
6194     scope->flags |= SCOPE_DONT_RECYCLE;

```

(eval.c)

As you can see, `SCOPE_DONT_RECYCLE` is set. Then next, take a look at the definition of `POP_SCOPE()`:

## ▼ `POP_SCOPE()` only the beginning

```
869 #define POP_SCOPE()
870     if (ruby_scope->flags & SCOPE_DONT_RECYCLE) {
871         if (_old) scope_dup(_old);
872     }

```

(eval.c)

When it pops, if `SCOPE_DONT_RECYCLE` flag was set to the current `SCOPE` (`ruby_scope`), it also does `scope_dup()` of the previous `SCOPE` (`_old`). In other words, `SCOPE_DONT_RECYCLE` is also set to this one. In this way, one by one, the flag is propagated at the time when it pops. (Figure 7)

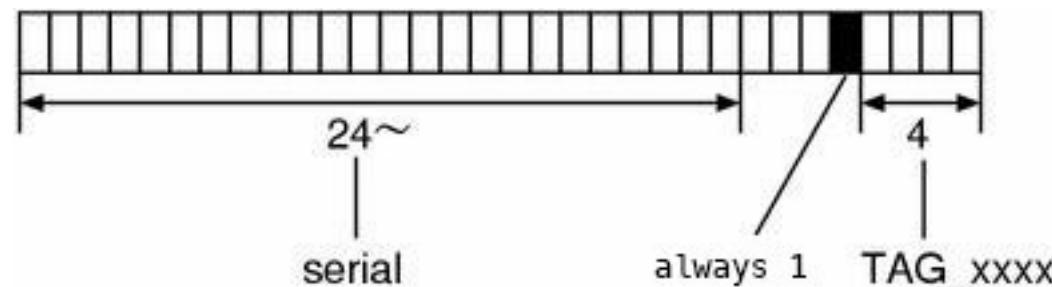


Figure 7: flag propagation

Since `VARS` also does not have any `prev` pointer, the same technique is used to propagate the `DVAR_DONT_RECYCLE` flag.

Next, the second point, try to think about “why all of them are duplicated”. We can understand that the local variables of `SCOPE` can be referred to later if its `Proc` is created. However, is it necessary to copy all of them including the previous `SCOPE` in order to accomplish that?

Honestly speaking, I couldn't find the answer of this question and has been worried about how can I write this section for almost three days, I've just got the answer. Take a look at the next program:

```
def get_proc
  Proc.new { nil }
```

```
end
```

```
env = get_proc { p 'ok' }
eval("yield", env)
```

I have not explained this feature, but by passing a `Proc` object as the second argument of `eval`, you can evaluate the string in that environment.

It means, as the readers who have read until here can probably tell, it pushes the various environments taken from the `Proc` (meaning `BLOCK`) and evaluates. In this case, it naturally also pushes `BLOCK` and you can turn the `BLOCK` into a `Proc` again. Then, using the `Proc` when doing `eval ...` if things are done like this, you can access almost all information of `ruby_block` from Ruby level as you like. This is the reason why the entire stacks need to be fully duplicated.

((errata: we cannot access `ruby_block` as we like from Ruby level. The reason why all `SCOPES` are duplicated was not understood. It seems all we can do is to investigate the mailing list archives of the time when this change was applied. (It is still not certain whether we can find out the reason in this way.))

## Invocation of Proc

Next, we'll look at the invocation of a created `Proc`. Since `Proc#call` can be used from Ruby to invoke, we can follow the substance of it.

The substance of `Proc#call` is `proc_call()`:

## ▼ proc\_call()

```
6570 static VALUE
6571 proc_call(proc, args)
6572     VALUE proc, args;             /* OK */
6573 {
6574     return proc_invoke(proc, args, Qtrue, Qundef);
6575 }
```

(eval.c)

Delegate to `proc_invoke()`. When I look up `invoke` in a dictionary, it was written such as “call on (God, etc.) for help”, but when it is in the context of programming, it is often used in the almost same meaning as “activate”.

The prototype of the `proc_invoke()` is,

```
proc_invoke(VALUE proc, VALUE args, int pcall, VALUE self)
```

However, according to the previous code, `pcall=Qtrue` and `self=Qundef` in this case, so these two can be removed by constant foldings.

## ▼ proc\_invoke (simplified)

```
static VALUE
proc_invoke(proc, args, /* pcall=Qtrue */, /* self=Qundef */)
    VALUE proc, args;
    VALUE self;
{
    struct BLOCK * volatile old_block;
    struct BLOCK _block;
    struct BLOCK *data;
```

```
volatile VALUE result = Qnil;
int state;
volatile int orphan;
volatile int safe = ruby_safe_level;
volatile VALUE old_wrapper = ruby_wrapper;
struct RVarmap * volatile old_dvars = ruby_dyna_vars;

/* (A) take BLOCK from proc and assign it to data */
Data_Get_Struct(proc, struct BLOCK, data);
/* (B) blk_orphan */
orphan = blk_orphan(data);

ruby_wrapper = data->wrapper;
ruby_dyna_vars = data->dyna_vars;
/* (C) push BLOCK from data */
old_block = ruby_block;
_block = *data;
ruby_block = &_block;

/* (D) transition to ITER_CUR */
PUSH_ITER(ITER_CUR);
ruby_frame->iter = ITER_CUR;

PUSH_TAG(PROT_NONE);
state = EXEC_TAG();
if (state == 0) {
    proc_set_safe_level(proc);
    /* (E) invoke the block */
    result = rb_yield_0(args, self, 0, pcall);
}
POP_TAG();

POP_ITER();
if (ruby_block->tag->dst == state) {
    state &= TAG_MASK;           /* target specified jump */
}
ruby_block = old_block;
ruby_wrapper = old_wrapper;
ruby_dyna_vars = old_dvars;
ruby_safe_level = safe;

switch (state) {
    case 0:
```

```

        break;
    case TAG_BREAK:
        result = prot_tag->retval;
        break;
    case TAG_RETURN:
        if (orphan) { /* orphan procedure */
            localjump_error("return from proc-closure", prot_tag->
        }
        /* fall through */
    default:
        JUMP_TAG(state);
    }
    return result;
}

```

The crucial points are three: C, D, and E.

- (C) At NODE\_ITER a BLOCK is created from the syntax tree and pushed, but this time, a BLOCK is taken from Proc and pushed.
- (D) It was ITER\_PRE before becoming ITER\_CUR at rb\_call0(), but this time it goes directly into ITER\_CUR.
- (E) If the case was an ordinary iterator, its method call exists before yield occurs then going to rb\_yield\_0, but this time rb\_yield\_() is directly called and invokes the just pushed block.

In other words, in the case of iterator, the procedures are separated into three places, NODE\_ITER ~ rb\_call0() ~ NODE\_YIELD. But this time, they are done all at once.

Finally, I'll talk about the meaning of blk\_orphan(). As the name suggests, it is a function to determine the state of "the method

which created the Proc has finished”. For example, the scope used by a BLOCK has already been popped, you can determine it has finished.

## Block and Proc

In the previous chapter, various things about arguments and parameters of methods are discussed, but I have not described about block parameters yet. Although it is brief, here I'll perform the final part of that series.

```
def m(&block)
end
```

This is a “block parameter”. The way to enable this is very simple. If `m` is an iterator, it is certain that a BLOCK was already pushed, turn it into a Proc and assign into (in this case) the local variable `block`. How to turn a block into a Proc is just calling `proc_new()`, which was previously described. The reason why just calling is enough can be a little incomprehensible. However whichever `Proc.new` or `m`, the situation “a method is called and a BLOCK is pushed” is the same. Therefore, from C level, anytime you can turn a block into a Proc by just calling `proc_new()`.

And if `m` is not an iterator, all we have to do is simply assigning `nil`.

Next, it is the side to pass a block.

`m(&block)`

This is a “block argument”. This is also simple, take a `BLOCK` from (a `Proc` object stored in) `block` and push it. What differs from `PUSH_BLOCK()` is only whether a `BLOCK` has already been created in advance or not.

The function to do this procedure is `block_pass()`. If you are curious about, check and confirm around it. However, it really does just only what was described here, it’s possible you’ll be disappointed...

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5 License](#)

# Ruby Hacking Guide

# Chapter 17: Dynamic evaluation

## Overview

---

I have already finished to describe about the mechanism of the evaluator by the previous chapter. In this chapter, by including the parser in addition to it, let's examine the big picture as “the evaluator in a broad sense”. There are three targets: `eval`, `Module#module_eval` and `Object#instance_eval`.

### eval

I've already described about `eval`, but I'll introduce more tiny things about it here.

By using `eval`, you can compile and evaluate a string at runtime in the place. Its return value is the value of the last expression of the program.

```
p eval("1 + 1")  # 2
```

You can also refer to a variable in its scope from inside of a string to eval.

```
lvar = 5
@ivar = 6
p eval("lvar + @ivar")    # 11
```

Readers who have been reading until here cannot simply read and pass over the word “its scope”. For instance, you are curious about how is its “scope” of constants, aren’t you? I am. To put the bottom line first, basically you can think it directly inherits the environment of outside of eval.

And you can also define methods and define classes.

```
def a
  eval('class C;  def test() puts("ok") end  end')
end

a()          # define class C and C#test
C.new.test  # shows ok
```

Moreover, as mentioned a little in the previous chapter, when you pass a Proc as the second argument, the string can be evaluated in its environment.

```
def new_env
  n = 5
  Proc.new { nil }  # turn the environment of this method into
end

p eval('n * 3', new_env())  # 15
```

## module\_eval and instance\_eval

When a Proc is passed as the second argument of eval, the evaluations can be done in its environment. `module_eval` and `instance_eval` is its limited (or shortcut) version. With `module_eval`, you can evaluate in an environment that is as if in a module statement or a class statement.

```
lvar = "toplevel lvar"  # a local variable to confirm this scope

module M
end
M.module_eval(<<'EOS')  # a suitable situation to use here-doc
  p lvar  # referable
  p self  # shows M
  def ok  # define M#ok
    puts 'ok'
  end
EOS
```

With `instance_eval`, you can evaluate in an environment whose `self` of the singleton class statement is the object.

```
lvar = "toplevel lvar"  # a local variable to confirm this scope

obj = Object.new
obj.instance_eval(<<'EOS')
  p lvar  # referable
  p self  # shows #<Object:0x40274f5c>
  def ok  # define obj.ok
    puts 'ok'
  end
EOS
```

Additionally, these `module_eval` and `instance_eval` can also be used

as iterators, a block is evaluated in each environment in that case. For instance,

```
obj = Object.new
p obj           # #<Object:0x40274fac>
obj.instance_eval {
  p self         # #<Object:0x40274fac>
}
```

Like this.

However, between the case when using a string and the case when using a block, the behavior around local variables is different each other. For example, when creating a block in the `a` method then doing `instance_eval` it in the `b` method, the block would refer to the local variables of `a`. When creating a string in the `a` method then doing `instance_eval` it in the `b` method, from inside of the string, it would refer to the local variables of `b`. The scope of local variables is decided “at compile time”, the consequence differs because a string is compiled every time but a block is compiled when loading files.

## eval

---

### eval()

The `eval` of Ruby branches many times based on the presence and absence of the parameters. Let's assume the form of call is limited

to the below:

```
eval(prog_string, some_block)
```

Then, since this makes the actual interface function `rb_f_eval()` almost meaningless, we'll start with the function `eval()` which is one step lower. The function prototype of `eval()` is:

```
static VALUE
eval(VALUE self, VALUE src, VALUE scope, char *file, int line);
```

`scope` is the `Proc` of the second parameter. `file` and `line` is the file name and line number of where a string to `eval` is supposed to be located. Then, let's see the content:

## ▼ `eval()` (simplified)

```
4984 static VALUE
4985 eval(self, src, scope, file, line)
4986     VALUE self, src, scope;
4987     char *file;
4988     int line;
4989 {
4990     struct BLOCK *data = NULL;
4991     volatile VALUE result = Qnil;
4992     struct SCOPE * volatile old_scope;
4993     struct BLOCK * volatile old_block;
4994     struct RVarmap * volatile old_dyna_vars;
4995     VALUE volatile old_cref;
4996     int volatile old_vmode;
4997     volatile VALUE old_wrapper;
4998     struct FRAME frame;
4999     NODE *nodesave = ruby_current_node;
5000     volatile int iter = ruby_frame->iter;
5001     int state;
```

```
5002
5003     if (!NIL_P(scope)) { /* always true now */
5004         Data_Get_Struct(scope, struct BLOCK, data);
5005         /* push BLOCK from data */
5006         frame = data->frame;
5007         frame.tmp = ruby_frame; /* to prevent from GC */
5008         ruby_frame = &(frame);
5009         old_scope = ruby_scope;
5010         ruby_scope = data->scope;
5011         old_block = ruby_block;
5012         ruby_block = data->prev;
5013         old_dyna_vars = ruby_dyna_vars;
5014         ruby_dyna_vars = data->dyna_vars;
5015         old_vmode = scope_vmode;
5016         scope_vmode = data->vmode;
5017         old_cref = (VALUE)ruby_cref;
5018         ruby_cref = (NODE*)ruby_frame->cbase;
5019         old_wrapper = ruby_wrapper;
5020         ruby_wrapper = data->wrapper;
5021         self = data->self;
5022         ruby_frame->iter = data->iter;
5023     }
5024     PUSH_CLASS();
5025     ruby_class = ruby_cbase; /* == ruby_frame->cbase */
5026
5027     ruby_in_eval++;
5028     if (TYPE(ruby_class) == T_ICLASS) {
5029         ruby_class = RBASIC(ruby_class)->klass;
5030     }
5031     PUSH_TAG(PROT_NONE);
5032     if ((state = EXEC_TAG()) == 0) {
5033         NODE *node;
5034
5035         result = ruby_errinfo;
5036         ruby_errinfo = Qnil;
5037         node = compile(src, file, line);
5038         if (ruby_nerrs > 0) {
5039             compile_error(0);
5040         }
5041         if (!NIL_P(result)) ruby_errinfo = result;
5042         result = eval_node(self, node);
5043     }
5044     POP_TAG();
```

```

5066     POP_CLASS();
5067     ruby_in_eval--;
5068     if (!NIL_P(scope)) { /* always true now */
5069         int dont_recycle = ruby_scope->flags & SCOPE_DONT_RE
5070
5071         ruby_wrapper = old_wrapper;
5072         ruby_cref = (NODE*)old_cref;
5073         ruby_frame = frame.tmp;
5074         ruby_scope = old_scope;
5075         ruby_block = old_block;
5076         ruby_dyna_vars = old_dyna_vars;
5077         data->vmode = scope_vmode; /* save the modification
5078         scope_vmode = old_vmode;
5079         if (dont_recycle) {
5080             /* .....copy SCOPE BLOCK VARS..... */
5081         }
5082     }
5083     if (state) {
5084         if (state == TAG_RAISE) {
5085             /* .....prepare an exception object..... */
5086             rb_exc_raise(ruby_errinfo);
5087         }
5088         JUMP_TAG(state);
5089     }
5090
5091     return result;
5092 }

```

(eval.c)

If this function is shown without any preamble, you probably feel “oww!”. But we’ve defeated many functions of eval.c until here, so this is not enough to be an enemy of us. This function is just continuously saving/restoring the stacks. The points we need to care about are only the below three:

- unusually FRAME is also replaced (not copied and pushed)
- ruby\_cref is substituted (?) by ruby\_frame->cbase

- only `scope_vmode` is not simply restored but influences data.

And the main parts are the `compile()` and `eval_node()` located around the middle. Though it's possible that `eval_node()` has already been forgotten, it is the function to start the evaluation of the parameter node. It was also used in `ruby_run()`.

Here is `compile()`.

### ▼ `compile()`

```

4968 static NODE*
4969 compile(src, file, line)
4970     VALUE src;
4971     char *file;
4972     int line;
4973 {
4974     NODE *node;
4975
4976     ruby_nerrs = 0;
4977     Check_Type(src, T_STRING);
4978     node = rb_compile_string(file, src, line);
4979
4980     if (ruby_nerrs == 0) return node;
4981     return 0;
4982 }
```

(eval.c)

`ruby_nerrs` is the variable incremented in `yyerror()`. In other words, if this variable is non-zero, it indicates more than one parse error happened. And, `rb_compile_string()` was already discussed in Part 2. It was a function to compile a Ruby string into a syntax tree.

One thing becomes a problem here is local variable. As we've seen in Chapter 12: Syntax tree construction, local variables are managed by using `lvtbl`. However, since a `SCOPE` (and possibly also `VARS`) already exists, we need to parse in the way of writing over and adding to it. This is in fact the heart of `eval()`, and is the worst difficult part. Let's go back to `parse.y` again and complete this investigation.

## ■ `top_local`

I've mentioned that the functions named `local_push()` `local_pop()` are used when pushing `struct local_vars`, which is the management table of local variables, but actually there's one more pair of functions to push the management table. It is the pair of `top_local_init()` and `top_local_setup()`. They are called in this sort of way.

### ▼ How `top_local_init()` is called

```
program : { top_local_init(); }
compstmt
{ top_local_setup(); }
```

Of course, in actuality various other things are also done, but all of them are cut here because it's not important. And this is the content of it:

### ▼ `top_local_init()`

```
5273 static void
5274 top_local_init()
5275 {
5276     local_push(1);
5277     lvtbl->cnt = ruby_scope->local_tbl?ruby_scope->local_tbl
5278     if (lvtbl->cnt > 0) {
5279         lvtbl->tbl = ALLOC_N(ID, lvtbl->cnt+3);
5280         MEMCPY(lvtbl->tbl, ruby_scope->local_tbl, ID, lvtbl-
5281     }
5282     else {
5283         lvtbl->tbl = 0;
5284     }
5285     if (ruby_dyna_vars)
5286         lvtbl->dlev = 1;
5287     else
5288         lvtbl->dlev = 0;
5289 }
```

(parse.y)

This means that `local_tbl` is copied from `ruby_scope` to `lvtbl`. As for block local variables, since it's better to see them all at once later, we'll focus on ordinary local variables for the time being. Next, here is `top_local_setup()`.

## ▼ `top_local_setup()`

```
5291 static void
5292 top_local_setup()
5293 {
5294     int len = lvtbl->cnt; /* the number of local variables
5295     int i; /* the number of local variables
5296
5297     if (len > 0) {
5298         i = ruby_scope->local_tbl ? ruby_scope->local_tbl[0]
5299
5300         if (i < len) {
5301             if (i == 0 || (ruby_scope->flags & SCPE_MALLOC)
```

```

5302     VALUE *vars = ALLOC_N(VALUE, len+1);
5303     if (ruby_scope->local_vars) {
5304         *vars++ = ruby_scope->local_vars[-1];
5305         MEMCPY(vars, ruby_scope->local_vars, VAL
5306             rb_mem_clear(vars+i, len-i);
5307     }
5308     else {
5309         *vars++ = 0;
5310         rb_mem_clear(vars, len);
5311     }
5312     ruby_scope->local_vars = vars;
5313     ruby_scope->flags |= SCOPE_MALLOC;
5314 }
5315 else {
5316     VALUE *vars = ruby_scope->local_vars-1;
5317     REALLOC_N(vars, VALUE, len+1);
5318     ruby_scope->local_vars = vars+1;
5319     rb_mem_clear(ruby_scope->local_vars+i, len-i
5320 }
5321 if (ruby_scope->local_tbl &&
5322     ruby_scope->local_vars[-1] == 0) {
5323     free(ruby_scope->local_tbl);
5324 }
5325     ruby_scope->local_vars[-1] = 0; /* NODE is not
5326     ruby_scope->local_tbl = local_tbl();
5327 }
5328     local_pop();
5329 }

```

(parse.y)

Since `local_vars` can be either in the stack or in the heap, it makes the code complex to some extent. However, this is just updating `local_tbl` and `local_vars` of `ruby_scope`. (When `SCOPE_MALLOC` was set, `local_vars` was allocated by `malloc()`). And here, because there's no meaning of using `alloca()`, it is forced to change its allocation method to `malloc`.

# Block Local Variable

By the way, how about block local variables? To think about this, we have to go back to the entry point of the parser first, it is `yycompile()`.

## ▼ setting `ruby_dyna_vars` aside

```
static NODE*
yycompile(f, line)
{
    struct RVarmap *vars = ruby_dyna_vars;
    :
    n = yyparse();
    :
    ruby_dyna_vars = vars;
}
```

This looks like a mere save-restore, but the point is that this does not clear the `ruby_dyna_vars`. This means that also in the parser it directly adds elements to the link of `RVarmap` created in the evaluator.

However, according to the previous description, the structure of `ruby_dyna_vars` differs between the parser and the evalutor. How does it deal with the difference in the way of attaching the header (`RVarmap` whose `id=0`)?

What is helpful here is the “1” of `local_push(1)` in `top_local_init()`. When the argument of `local_push()` becomes true, it does not attach the first header of `ruby_dyna_vars`. It means, it would look

like Figure 1. Now, it is assured that we can refer to the block local variables of the outside scope from inside of a string to eval.

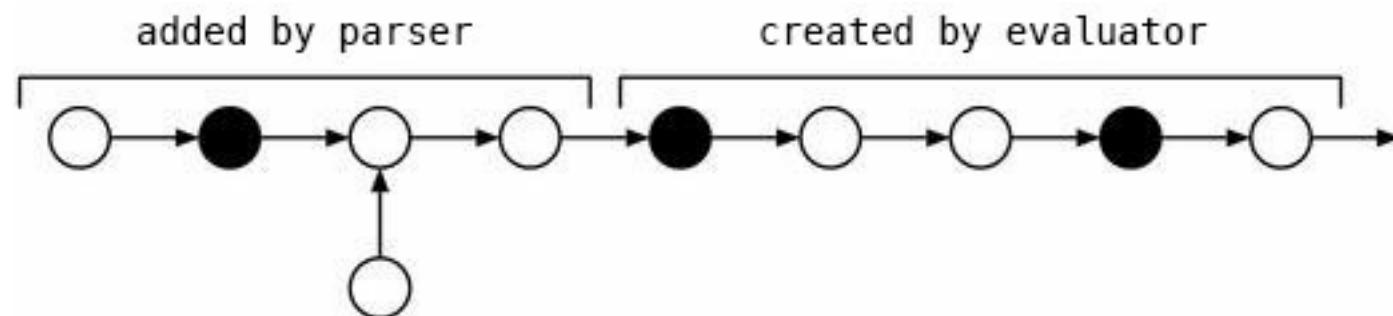


Figure 1: ruby\_dyna\_vars inside eval

Well, it's sure we can refer to, but didn't you say that `ruby_dyna_vars` is entirely freed in the parser? What can we do if the link created at the evaluator will be freed? ... I'd like the readers who noticed this to be relieved by reading the next part.

## ▼ `yycompile()` – freeing `ruby_dyna_vars`

```
2386     vp = ruby_dyna_vars;
2387     ruby_dyna_vars = vars;
2388     lex_strterm = 0;
2389     while (vp && vp != vars) {
2390         struct RVarmap *tmp = vp;
2391         vp = vp->next;
2392         rb_gc_force_recycle((VALUE)tmp);
2393     }
```

(parse.y)

It is designed so that the loop would stop when it reaches the link created at the evaluator (`vars`).

## The Whole Picture

The substance of `Module#module_eval` is `rb_mod_module_eval()`, and the substance of `Object#instance_eval` is `rb_obj_instance_eval()`.

### ▼ `rb_mod_module_eval()` `rb_obj_instance_eval()`

```
5316 VALUE
5317 rb_mod_module_eval(argc, argv, mod)
5318     int argc;
5319     VALUE *argv;
5320     VALUE mod;
5321 {
5322     return specific_eval(argc, argv, mod, mod);
5323 }

5298 VALUE
5299 rb_obj_instance_eval(argc, argv, self)
5300     int argc;
5301     VALUE *argv;
5302     VALUE self;
5303 {
5304     VALUE klass;
5305
5306     if (rb_special_const_p(self)) {
5307         klass = Qnil;
5308     }
5309     else {
5310         klass = rb_singleton_class(self);
5311     }
5312
5313     return specific_eval(argc, argv, klass, self);
5314 }
```

(eval.c)

These two methods have a common part as “a method to replace `self` with `class`”, that part is defined as `specific_eval()`. Figure 2 shows it and also what will be described. What with parentheses are calls by function pointers.

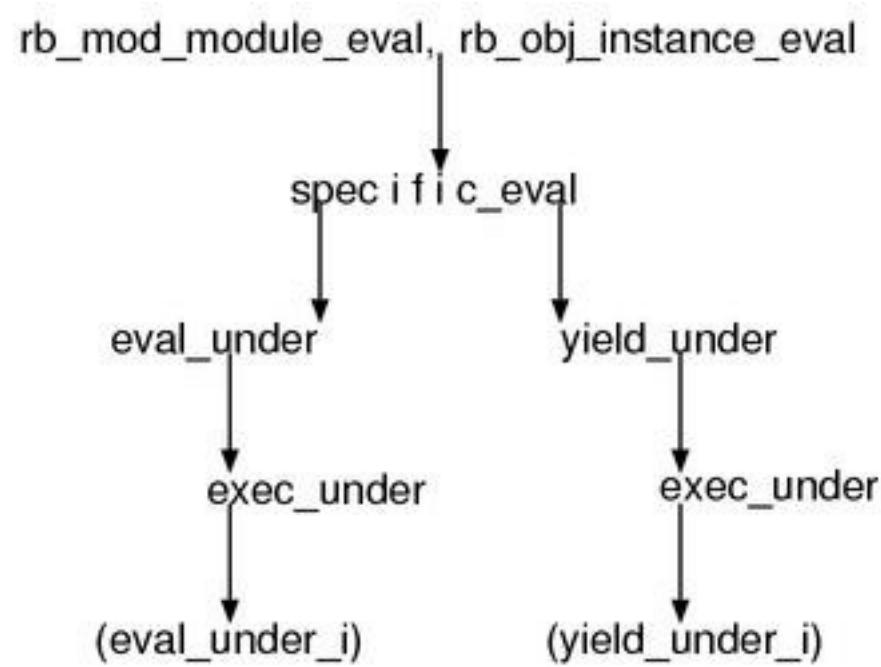


Figure 2: Call Graph

Whichever `instance_eval` or `module_eval`, it can accept both a block and a string, thus it branches for each particular process to `yield` and `eval` respectively. However, most of them are also common again, this part is extracted as `exec_under()`.

But for those who reading, one have to simultaneously face at 2 times  $2 = 4$  ways, it is not a good plan. Therefore, here we assume only the case when

1. it is an `instance_eval`
2. which takes a string as its argument

. And extracting all functions under `rb_obj_instance_eval()` in-line, folding constants, we'll read the result.

## After Absorbed

After all, it becomes very comprehensible in comparison to the one before being absorbed.

### ▼ `specific_eval() - instance_eval, eval, string`

```
static VALUE
instance_eval_string(self, src, file, line)
    VALUE self, src;
    const char *file;
    int line;
{
    VALUE sclass;
    VALUE result;
    int state;
    int mode;

    sclass = rb_singleton_class(self);

    PUSH_CLASS();
    ruby_class = sclass;
    PUSH_FRAME();
    ruby_frame->self      = ruby_frame->prev->self;
    ruby_frame->last_func  = ruby_frame->prev->last_func;
    ruby_frame->last_class = ruby_frame->prev->last_class;
    ruby_frame->argc       = ruby_frame->prev->argc;
    ruby_frame->argv       = ruby_frame->prev->argv;
    if (ruby_frame->cbase != sclass) {
        ruby_frame->cbase = rb_node_newnode(NODE_CREF, sclass, 0,
                                              ruby_frame->cbase);
    }
    PUSH_CREF(sclass);

    mode = scope_vmode;
```

```
SCOPE_SET(SCOPE_PUBLIC);
PUSH_TAG(PROT_NONE);
if ((state = EXEC_TAG()) == 0) {
    result = eval(self, src, Qnil, file, line);
}
POP_TAG();
SCOPE_SET(mode);

POP_CREF();
POP_FRAME();
POP_CLASS();
if (state) JUMP_TAG(state);

return result;
}
```

It seems that this pushes the singleton class of the object to CLASS and CREF and ruby\_frame->cbase. The main process is one-shot of eval(). It is unusual that things such as initializing FRAME by a struct-copy are missing, but this is also not create so much difference.

## ▀ Before being absorbed

Though the author said it becomes more friendly to read, it's possible it has been already simple since it was not absorbed, let's check where is simplified in comparison to the before-absorbed one.

The first one is specific\_eval(). Since this function is to share the code of the interface to Ruby, almost all parts of it is to parse the parameters. Here is the result of cutting them all.

## ▼ `specific_eval()` (simplified)

```
5258 static VALUE
5259 specific_eval(argc, argv, klass, self)
5260     int argc;
5261     VALUE *argv;
5262     VALUE klass, self;
5263 {
5264     if (rb_block_given_p()) {
5265         return yield_under(klass, self);
5266     }
5267     else {
5268         return eval_under(klass, self, argv[0], file, line);
5269     }
5270 }
```

(eval.c)

As you can see, this is perfectly branches in two ways based on whether there's a block or not, and each route would never influence the other. Therefore, when reading, we should read one by one. To begin with, the absorbed version is enhanced in this point.

And `file` and `line` are irrelevant when reading `yield_under()`, thus in the case when the route of `yield` is absorbed by the main body, it might become obvious that we don't have to think about the parse of these parameters at all.

Next, we'll look at `eval_under()` and `eval_under_i()`.

## ▼ `eval_under()`

```

5222 static VALUE
5223 eval_under(under, self, src, file, line)
5224     VALUE under, self, src;
5225     const char *file;
5226     int line;
5227 {
5228     VALUE args[4];
5229
5230     if (ruby_safe_level >= 4) {
5231        StringValue(src);
5232     }
5233     else {
5234         SafeStringValue(src);
5235     }
5236     args[0] = self;
5237     args[1] = src;
5238     args[2] = (VALUE)file;
5239     args[3] = (VALUE)line;
5240     return exec_under(eval_under_i, under, under, args);
5241 }
5214 static VALUE
5215 eval_under_i(args)
5216     VALUE *args;
5217 {
5218     return eval(args[0], args[1], Qnil, (char*)args[2], (int
5219 }
(eval.c)

```

In this function, in order to make its arguments single, it stores them into the `args` array and passes it. We can imagine that this `args` exists as a temporary container to pass from `eval_under()` to `eval_under_i()`, but not sure that it is truly so. It's possible that `args` is modified inside `exec_under()`.

As a way to share a code, this is a very right way to do. But for those who read it, this kind of indirect passing is incomprehensible.

Particularly, because there are extra castings for `file` and `line` to fool the compiler, it is hard to imagine what were their actual types. The parts around this entirely disappeared in the absorbed version, so you don't have to worry about getting lost.

However, it's too much to say that absorbing and extracting always makes things easier to understand. For example, when calling `exec_under()`, `under` is passed as both the second and third arguments, but is it all right if the `exec_under()` side extracts the both parameter variables into `under`? That is to say, the second and third arguments of `exec_under()` are, in fact, indicating `CLASS` and `CREF` that should be pushed. `CLASS` and `CREF` are “different things”, it might be better to use different variables. Also in the previous absorbed version, for only this point,

```
VALUE sclass = .....;  
VALUE cbase = sclass;
```

I thought that I would write this way, but also thought it could give the strange impression if abruptly only these variables are left, thus it was extracted as `sclass`. It means that this is only because of the flow of the texts.

By now, so many times, I've extracted arguments and functions, and for each time I repeatedly explained the reason to extract. They are

- there are only a few possible patterns
- the behavior can slightly change

Definitely, I'm not saying "In whatever ways extracting various things always makes things simpler".

In whatever case, what of the first priority is the comprehensibility for ourself and not keep complying the methodology. When extracting makes things simpler, extract it. When we feel that not extracting or conversely bundling as a procedure makes things easier to understand, let us do it. As for `ruby`, I often extracted them because the original is written properly, but if a source code was written by a poor programmer, aggressively bundling to functions should often become a good choice.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

Translated by Vincent ISAMBART

# Chapter 18: Loading

## Outline

---

### Interface

At the Ruby level, there are two procedures that can be used for loading: `require` and `load`.

```
require 'uri'          # load the uri library
load '/home/foo/.myrc' # read a resource file
```

They are both normal methods, compiled and evaluated exactly like any other code. It means loading occurs after compilation gave control to the evaluation stage.

These two function each have their own use. ‘`require`’ is to load libraries, and `load` is to load an arbitrary file. Let’s see this in more details.

#### `require`

`require` has four features:

- the file is searched for in the load path

- it can load extension libraries
- the `.rb/.so` extension can be omitted
- a given file is never loaded more than once

Ruby's load path is in the global variable `$:`, which contains an array of strings. For example, displaying the content of the `$:` in the environment I usually use would show:

```
% ruby -e 'puts $:'  
/usr/lib/ruby/site_ruby/1.7  
/usr/lib/ruby/site_ruby/1.7/i686-linux  
/usr/lib/ruby/site_ruby  
/usr/lib/ruby/1.7  
/usr/lib/ruby/1.7/i686-linux  
.
```

Calling `puts` on an array displays one element on each line so it's easy to read.

As I ran `configure` using `--prefix=/usr`, the library path is `/usr/lib/ruby` and below, but if you compile it normally from the source code, the libraries will be in `/usr/local/lib/ruby` and below. In a Windows environment, there will also be a drive letter.

Then, let's try to `require` the standard library `nkf.so` from the load path.

```
require 'nkf'
```

If the `required` name has no extension, `require` silently compensates. First, it tries with `.rb`, then with `.so`. On some

platforms it also tries the platform's specific extension for extension libraries, for example `.dll` in a Windows environment or `.bundle` on Mac OS X.

Let's do a simulation on my environment. `ruby` checks the following paths in sequential order.

```
/usr/lib/ruby/site_ruby/1.7/nkf.rb
/usr/lib/ruby/site_ruby/1.7/nkf.so
/usr/lib/ruby/site_ruby/1.7/i686-linux/nkf.rb
/usr/lib/ruby/site_ruby/1.7/i686-linux/nkf.so
/usr/lib/ruby/site_ruby/nkf.rb
/usr/lib/ruby/site_ruby/nkf.so
/usr/lib/ruby/1.7/nkf.rb
/usr/lib/ruby/1.7/nkf.so
/usr/lib/ruby/1.7/i686-linux/nkf.rb
/usr/lib/ruby/1.7/i686-linux/nkf.so      found!
```

`nkf.so` has been found in `/usr/lib/ruby/1.7/i686-linux`. Once the file has been found, `require`'s last feature (not loading the file more than once) locks the file. The locks are strings put in the global variable `$"`. In our case the string `"nkf.so"` has been put there. Even if the extension has been omitted when calling `require`, the file name in `$"` has the extension.

```
require 'nkf'      # after loading nkf...
p $"                # ["nkf.so"]  the file is locked

require 'nkf'      # nothing happens if we require it again
p $"                # ["nkf.so"]  the content of the lock array does
```

There are two reasons for adding the missing extension. The first one is not to load it twice if the same file is later required with its

extension. The second one is to be able to load both `nkf.rb` and `nkf.so`. In fact the extensions are disparate (`.so` `.dll` `.bundle` etc.) depending on the platform, but at locking time they all become `.so`. That's why when writing a Ruby program you can ignore the differences of extensions and consider it's always `so`. So you can say that `ruby` is quite UNIX oriented.

By the way, `$"` can be freely modified even at the Ruby level so we cannot say it's a strong lock. You can for example load an extension library multiple times if you clear `$"`.

## load

`load` is a lot easier than `require`. Like `require`, it searches the file in `$:`. But it can only load Ruby programs. Furthermore, the extension cannot be omitted: the complete file name must always be given.

```
load 'uri.rb'    # load the URI library that is part of the stand
```

In this simple example we try to load a library, but the proper way to use `load` is for example to load a resource file giving its full path.

## Flow of the whole process

If we roughly split it, “loading a file” can be split in:

- finding the file
- reading the file and mapping it to an internal form

- evaluating it

The only difference between `require` and `load` is how to find the file. The rest is the same in both.

We will develop the last evaluation part a little more. Loaded Ruby programs are basically evaluated at the top-level. It means the defined constants will be top-level constants and the defined methods will be function-style methods.

```
### mylib.rb
MY_OBJECT = Object.new
def my_p(obj)
  p obj
end

### first.rb
require 'mylib'
my_p MY_OBJECT  # we can use the constants and methods defined
```

Only the local variable scope of the top-level changes when the file changes. In other words, local variables cannot be shared between different files. You can of course share them using for example `Proc` but this has nothing to do with the load mechanism.

Some people also misunderstand the loading mechanism. Whatever the class you are in when you call `load`, it does not change anything. Even if, like in the following example, you load a file in the `module` statement, it does not serve any purpose, as everything that is at the top-level of the loaded file is put at the Ruby top-level.

```
require 'mylib'      # whatever the place you require from, be it
module SandBox
  require 'mylib'    # or in a module, the result is the same
end
```

## ■ Highlights of this chapter

With the above knowledge in our mind, we are going to read. But because this time its specification is defined very particularly, if we simply read it, it could be just a enumeration of the codes. Therefore, in this chapter, we are going to reduce the target to the following 3 points:

- loading serialisation
- the repartition of the functions in the different source files
- how extension libraries are loaded

Regarding the first point, you will understand it when you see it.

For the second point, the functions that appear in this chapter come from 4 different files, `eval.c` `ruby.c` `file.c` `dln.c`. Why is this in this way? We'll try to think about the realistic situation behind it.

The third point is just like its name says. We will see how the currently popular trend of execution time loading, more commonly referred to as plug-ins, works. This is the most interesting part of this chapter, so I'd like to use as many pages as possible to talk about it.

# Searching the library

## rb\_f\_require()

The body of `require` is `rb_f_require`. First, we will only look at the part concerning the file search. Having many different cases is bothersome so we will limit ourselves to the case when no file extension is given.

### ▼ rb\_f\_require() (simplified version)

```
5527 VALUE
5528 rb_f_require(obj, fname)
5529     VALUE obj, fname;
5530 {
5531     VALUE feature, tmp;
5532     char *ext, *ftptr; /* OK */
5533     int state;
5534     volatile int safe = ruby_safe_level;
5535
5536     SafeStringValue(fname);
5537     ext = strrchr(RSTRING(fname)->ptr, '.');
5538     if (ext) {
5539         /* ...if the file extension has been given... */
5540     }
5541     tmp = fname;
5542     switch (rb_find_file_ext(&tmp, loadable_ext)) {
5543         case 0:
5544             break;
5545
5546         case 1:
5547             feature = fname = tmp;
5548             goto load_rb;
5549
5550         default:
5551             feature = tmp;
5552             fname = rb_find_file(tmp);
```

```

5597         goto load_dyna;
5598     }
5599     if (rb_feature_p(RSTRING(fname)->ptr, Qfalse))
5600         return Qfalse;
5601     rb_raise(rb_eLoadError, "No such file to load -- %s",
5602               RSTRING(fname)->ptr);
5603
5603     load_dyna:
5604         /* ...load an extension library... */
5605         return Qtrue;
5606
5607     load_rb:
5608         /* ...load a Ruby program... */
5609         return Qtrue;
5610 }

5491 static const char *const loadable_ext[] = {
5492     ".rb", DLEXT,      /* DLEXT=".so", ".dll", ".bundle"... */
5493 #ifdef DLEXT2
5494     DLEXT2,           /* DLEXT2=".dll" on Cygwin, MinGW */
5495 #endif
5496     0
5497 };

```

(eval.c)

In this function the `goto` labels `load_rb` and `load_dyna` are actually like subroutines, and the two variables `feature` and `fname` are more or less their parameters. These variables have the following meaning.

<b>variable</b>	<b>meaning</b>	<b>example</b>
<code>feature</code>	the library file name that will be put in <code>\$"</code>	<code>uri.rb</code> 、 <code>nkf.so</code>
<code>fname</code>	the full path to the library	<code>/usr/lib/ruby/1.7/uri.rb</code>

The name `feature` can be found in the function `rb_feature_p()`. This

function checks if a file has been locked (we will look at it just after).

The functions actually searching for the library are `rb_find_file()` and `rb_find_file_ext()`. `rb_find_file()` searches a file in the load path `$'`. `rb_find_file_ext()` does the same but the difference is that it takes as a second parameter a list of extensions (i.e. `loadable_ext`) and tries them in sequential order.

Below we will first look entirely at the file searching code, then we will look at the code of the `require` lock in `load_rb`.

## ■ `rb_find_file()`

First the file search continues in `rb_find_file()`. This function searches the file `path` in the global load path `$'` (`rb_load_path`). The string contamination check is tiresome so we'll only look at the main part.

### ▼ `rb_find_file()` (simplified version)

```
2494 VALUE
2495 rb_find_file(path)
2496     VALUE path;
2497 {
2498     VALUE tmp;
2499     char *f = RSTRING(path)->ptr;
2500     char *lpath;

2530     if (rb_load_path) {
2531         long i;
```

```

2533     Check_Type(rb_load_path, T_ARRAY);
2534     tmp = rb_ary_new();
2535     for (i=0;i<RARRAY(rb_load_path)->len;i++) {
2536         VALUE str = RARRAY(rb_load_path)->ptr[i];
2537         SafeStringValue(str);
2538         if (RSTRING(str)->len > 0) {
2539             rb_ary_push(tmp, str);
2540         }
2541     }
2542     tmp = rb_ary_join(tmp, rb_str_new2(PATH_SEP));
2543     if (RSTRING(tmp)->len == 0) {
2544         lpath = 0;
2545     }
2546     else {
2547         lpath = RSTRING(tmp)->ptr;
2548     }
2549 }
2550
2551     f = dln_find_file(f, lpath);
2552     if (file_load_ok(f)) {
2553         return rb_str_new2(f);
2554     }
2555     return 0;
2556 }

```

(file.c)

If we write what happens in Ruby we get the following:

```

tmp = []                      # make an array
$:each do |path|              # repeat on each element of the load
  tmp.push path if path.length > 0 # check the path and push it
end
lpath = tmp.join(PATH_SEP)     # concatenate all elements in one string
dln_find_file(f, lpath)        # main processing

```

PATH\_SEP is the path separator: ':' under UNIX, ';' under Windows.  
 rb\_ary\_join() creates a string by putting it between the different

elements. In other words, the load path that had become an array is back to a string with a separator.

Why? It's only because `dln_find_file()` takes the paths as a string with `PATH_SEP` as a separator. But why is `dln_find_file()` implemented like that? It's just because `dln.c` is not a library for ruby. Even if it has been written by the same author, it's a general purpose library. That's precisely for this reason that when I sorted the files by category in the Introduction I put this file in the Utility category. General purpose libraries cannot receive Ruby objects as parameters or read ruby global variables.

`dln_find_file()` also expands for example `~` to the home directory, but in fact this is already done in the omitted part of `rb_find_file()`. So in ruby's case it's not necessary.

## ■ Loading wait

Here, file search is finished quickly. Then comes is the loading code. Or more accurately, it is “up to just before the load”. The code of `rb_f_require()`'s `load_rb` has been put below.

### ▼ `rb_f_require():load_rb`

```
5625  load_rb:
5626      if (rb_feature_p(RSTRING(feature)->ptr, Qtrue))
5627          return Qfalse;
5628      ruby_safe_level = 0;
5629      rb_provide_feature(feature);
5630      /* the loading of Ruby programs is serialised */
```

```
5631     if (!loading_tbl) {  
5632         loading_tbl = st_init_strtable();  
5633     }  
5634     /* partial state */  
5635     ftptr = ruby_strdup(RSTRING(feature)->ptr);  
5636     st_insert(loading_tbl, ftptr, curr_thread);  
5637     /* ...load the Ruby program and evaluate it... */  
5638     st_delete(loading_tbl, &ftptr, 0); /* loading done */  
5639     free(ftptr);  
5640     ruby_safe_level = safe;
```

(eval.c)

Like mentioned above, `rb_feature_p()` checks if a lock has been put in `$".` And `rb_provide_feature()` pushes a string in `$",` in other words locks the file.

The problem comes after. Like the comment says “the loading of Ruby programs is serialised”. In other words, a file can only be loaded from one thread, and if during the loading another thread tries to load the same file, that thread will wait for the first loading to be finished. If it were not the case:

```
Thread.fork {  
    require 'foo'    # At the beginning of require, foo.rb is adde  
}                      # However the thread changes during the eval  
require 'foo'    # foo.rb is already in $" so the function return  
# (A) the classes of foo are used...
```

By doing something like this, even though the `foo` library is not really loaded, the code at (A) ends up being executed.

The process to enter the waiting state is simple. A `st_table` is created in `loading_tbl`, the association “`feature=>waiting thread`” is

recorded in it. `curr_thread` is in `eval.c`'s functions, its value is the current running thread.

The mechanism to enter the waiting state is very simple. A `st_table` is created in the `loading_tbl` global variable, and a “`feature=>loading_thread`” association is created. `curr_thread` is a variable from `eval.c`, and its value is the currently running thread. That makes an exclusive lock. And in `rb_feature_p()`, we wait for the loading thread to end like the following.

### ▼ `rb_feature_p()` (second half)

```
5477 rb_thread_t th;
5478
5479 while (st_lookup(loading_tbl, f, &th)) {
5480     if (th == curr_thread) {
5481         return Qtrue;
5482     }
5483     CHECK_INTS;
5484     rb_thread_schedule();
5485 }
```

(`eval.c`)

When `rb_thread_schedule()` is called, the control is transferred to an other thread, and this function only returns after the control returned back to the thread where it was called. When the file name disappears from `loading_tbl`, the loading is finished so the function can end. The `curr_thread` check is not to lock itself (figure 1).

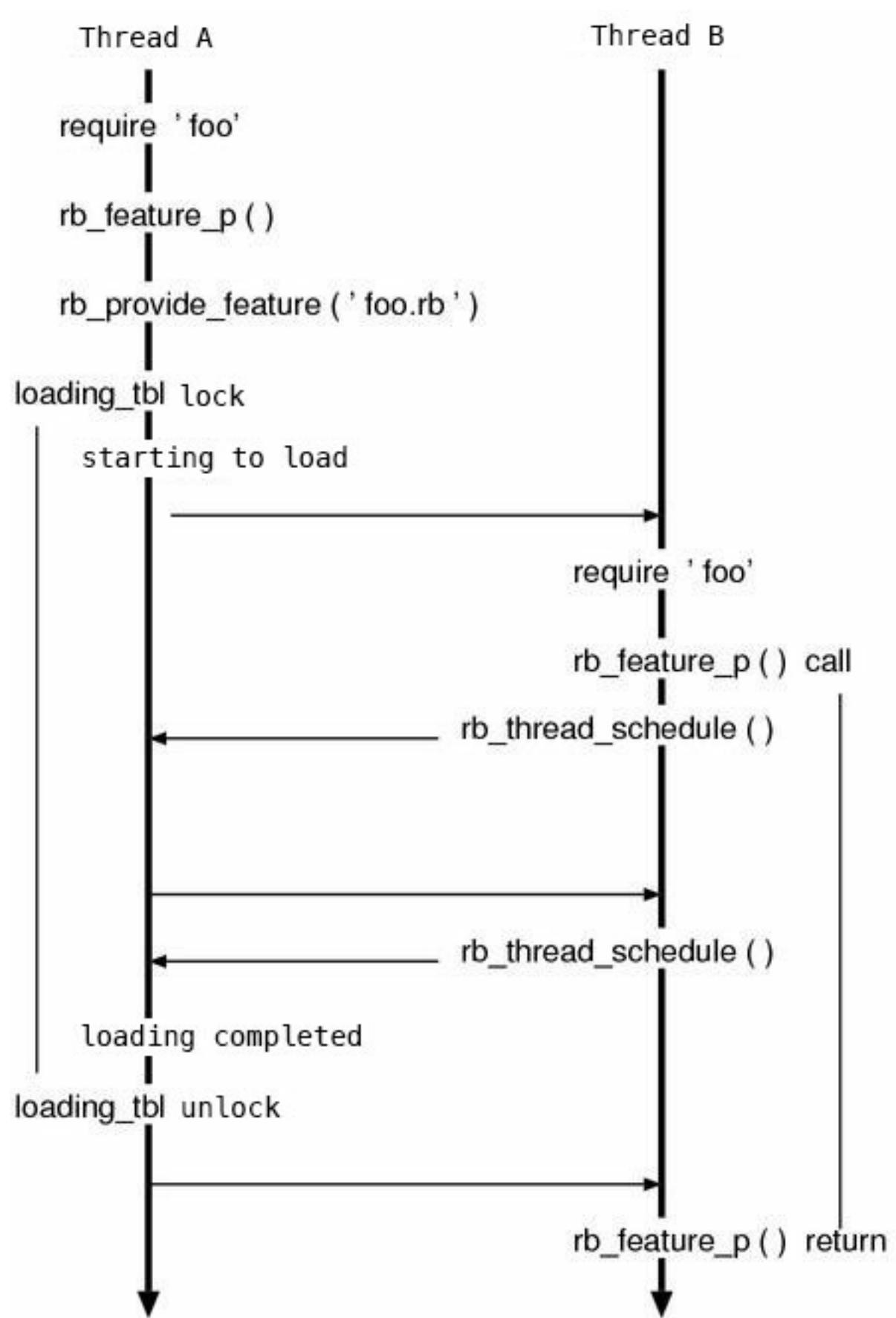


Figure 1: Serialisation of loads

# Loading of Ruby programs

## rb\_load()

We will now look at the loading process itself. Let's start by the part inside `rb_f_require()`'s `load_rb` loading Ruby programs.

### ▼ `rb_f_require()-load_rb-` loading

```
5638     PUSH_TAG(prot_none);
5639     if ((state = EXEC_TAG()) == 0) {
5640         rb_load(fname, 0);
5641     }
5642     POP_TAG();  
  
(eval.c)
```

The `rb_load()` which is called here is actually the “meat” of the Ruby-level `load`. This means it needs to search once again, but looking at the same procedure once again is too much trouble. Therefore, that part is omitted in the below codes.

And the second argument `wrap` is folded with 0 because it is 0 in the above calling code.

### ▼ `rb_load()` (simplified edition)

```
void
rb_load(fname, /* wrap=0 */
         VALUE fname;
{
    int state;
    volatile ID last_func;
```

```
volatile VALUE wrapper = 0;
volatile VALUE self = ruby_top_self;
NODE *saved_cref = ruby_cref;

PUSH_VARS();
PUSH_CLASS();
ruby_class = rb_cObject;
ruby_cref = top_cref; /* (A-1) change CREF */
wrapper = ruby_wrapper;
ruby_wrapper = 0;
PUSH_FRAME();
ruby_frame->last_func = 0;
ruby_frame->last_class = 0;
ruby_frame->self = self; /* (A-2) change ruby_frame->cb
ruby_frame->cbase = (VALUE)rb_node_newnode(NODE_CREF, ruby_clas
PUSH_SCOPE();
/* at the top-level the visibility is private by default */
SCOPE_SET(SCOPE_PRIVATE);
PUSH_TAG(prot_NONE);
ruby_errinfo = Qnil; /* make sure it's nil */
state = EXEC_TAG();
last_func = ruby_frame->last_func;
if (state == 0) {
    NODE *node;

    /* (B) this is dealt with as eval for some reasons */
    ruby_in_eval++;
    rb_load_file(RSTRING(fname)->ptr);
    ruby_in_eval--;
    node = ruby_eval_tree;
    if (ruby_nerrs == 0) { /* no parse error occurred */
        eval_node(self, node);
    }
}
ruby_frame->last_func = last_func;
POP_TAG();
ruby_cref = saved_cref;
POP_SCOPE();
POP_FRAME();
POP_CLASS();
POP_VARS();
ruby_wrapper = wrapper;
if (ruby_nerrs > 0) { /* a parse error occurred */
```

```
    ruby_nerrs = 0;
    rb_exc_raise(ruby_errinfo);
}
if (state) jump_tag_but_local_jump(state);
if (!NIL_P(ruby_errinfo)) /* an exception was raised during
    rb_exc_raise(ruby_errinfo);
}
```

Just after we thought we've been through the storm of stack manipulations we entered again. Although this is tough, let's cheer up and read it.

As the long functions usually are, almost all of the code are occupied by the idioms. PUSH/POP, tag protecting and re-jumping. Among them, what we want to focus on is the things on (A) which relate to CREF. Since a loaded program is always executed on the top-level, it sets aside (not push) `ruby_cref` and brings back `top_cref`. `ruby_frame->cbase` also becomes a new one.

And one more place, at (B) somehow `ruby_in_eval` is turned on. What is the part influenced by this variable? I investigated it and it turned out that it seems only `rb_compile_error()`. When `ruby_in_eval` is true, the message is stored in the exception object, but when it is not true, the message is printed to `stderr`. In other words, when it is a parse error of the main program of the command, it wants to print directly to `stderr`, but when inside of the evaluator, it is not appropriate so it stops to do it. It seems the “eval” of `ruby_in_eval` means neither the `eval` method nor the `eval()` function but “evaluate” as a general noun. Or, it's possible it indicates `eval.c`.

## ■ rb\_load\_file()

Then, all of a sudden, the source file is `ruby.c` here. Or to put it more accurately, essentially it is favorable if the entire loading code was put in `ruby.c`, but `rb_load()` has no choice but to use `PUSH_TAG` and such. Therefore, putting it in `eval.c` is inevitable. If it were not the case, all of them would be put in `eval.c` in the first place.

Then, it is `rb_load_file()`.

### ▼ rb\_load\_file()

```
865 void
866 rb_load_file(fname)
867     char *fname;
868 {
869     load_file(fname, 0);
870 }
```

(`ruby.c`)

Delegated entirely. The second argument `script` of `load_file()` is a boolean value and it indicates whether it is loading the file of the argument of the `ruby` command. Now, because we'd like to assume we are loading a library, let's fold it by replacing it with `script=0`. Furthermore, in the below code, also thinking about the meanings, non essential things have already been removed.

### ▼ load\_file() (simplified edition)

```
static void
```

```

load_file(fname, /* script=0 */)
    char *fname;
{
    VALUE f;
{
    FILE *fp = fopen(fname, "r");      (A)
    if (fp == NULL) {
        rb_load_fail(fname);
    }
    fclose(fp);
}
f = rb_file_open(fname, "r");          (B)
rb_compile_file(fname, f, 1);         (C)
rb_io_close(f);
}

```

- (A) The call to `fopen()` is to check if the file can be opened. If there is no problem, it's immediately closed. It may seem a little useless but it's an extremely simple and yet highly portable and reliable way to do it.
- (B) The file is opened once again, this time using the Ruby level library `File.open`. The file was not opened with `File.open` from the beginning so as not to raise any Ruby exception. Here if any exception occurred we would like to have a loading error, but getting the errors related to `open`, for example `Errno::ENOENT`, `Errno::EACCESS...`, would be problematic. We are in `ruby.c` so we cannot stop a tag jump.
- (C) Using the parser interface `rb_compile_file()`, the program is read from an `IO` object, and compiled in a syntax tree. The syntax tree is added to `ruby_eval_tree` so there is no need to get the result.

That's all for the loading code. Finally, the calls were quite deep so the callgraph of `rb_f_require()` is shown bellow.

```
rb_f_require      ....eval.c
  rb_find_file    ....file.c
    dln_find_file  ....dln.c
      dln_find_file_1
rb_load
  rb_load_file    ....ruby.c
    load_file
      rb_compile_file ....parse.y
eval_node
```

You must bring callgraphs on a long trip. It's common knowledge.

## The number of open required for loading

Previously, there was `open` used just to check if a file can be open, but in fact, during the loading process of `ruby`, additionally other functions such as `rb_find_file_ext()` also internally do checks by using `open`. How many times is `open()` called in the whole process?

If you're wondering that, just actually counting it is the right attitude as a programmer. We can easily count it by using a system call tracer. The tool to use would be `strace` on Linux, `truss` on Solaris, `ktrace` or `truss` on BSD. Like this, for each OS, the name is different and there's no consistency, but you can find them by googling.

If you're using Windows, probably your IDE will have a tracer built in. Well, as my main environment is Linux, I looked using `strace`.

The output is done on stderr so it was redirected using 2>&1.

```
% strace ruby -e 'require "rational"' 2>&1 | grep '^open'
open("/etc/ld.so.preload", 0_RDONLY)      = -1 ENOENT
open("/etc/ld.so.cache", 0_RDONLY)        = 3
open("/usr/lib/libruby-1.7.so.1.7", 0_RDONLY) = 3
open("/lib/libdl.so.2", 0_RDONLY)        = 3
open("/lib/libcrypt.so.1", 0_RDONLY)       = 3
open("/lib/libc.so.6", 0_RDONLY)          = 3
open("/usr/lib/ruby/1.7/rational.rb", 0_RDONLY|0_LARGEFILE) = 3
open("/usr/lib/ruby/1.7/rational.rb", 0_RDONLY|0_LARGEFILE) = 3
open("/usr/lib/ruby/1.7/rational.rb", 0_RDONLY|0_LARGEFILE) = 3
open("/usr/lib/ruby/1.7/rational.rb", 0_RDONLY|0_LARGEFILE) = 3
```

Until the open of `libc.so.6`, it is the `open` used in the implementation of dynamic links, and there are the other four `opens`. Thus it seems the three of them are useless.

## Loading of extension libraries

### rb\_f\_require()-load\_dyna

This time we will see the loading of extension libraries. We will start with `rb_f_require()`'s `load_dyna`. However, we do not need the part about locking anymore so it was removed.

### ▼ rb\_f\_require()-load\_dyna

```
5607  {
5608      int volatile old_vmode = scope_vmode;
```

```
5609
5610     PUSH_TAG(prot_none);
5611     if ((state = EXEC_TAG()) == 0) {
5612         void *handle;
5613
5614         SCOPE_SET(SCOPE_PUBLIC);
5615         handle = dln_load(RSTRING(fname)->ptr);
5616         rb_ary_push(ruby_dln_librefs, LONG2NUM((long)handle)
5617     }
5618     POP_TAG();
5619     SCOPE_SET(old_vmode);
5620 }
5621 if (state) JUMP_TAG(state);

(eval.c)
```

By now, there is very little here which is novel. The tags are used only in the way of the idiom, and to save/restore the visibility scope is done in the way we get used to see. All that remains is `dln_load()`. What on earth is that for? For the answer, continue to the next section.

## Brush up about links

`dln_load()` is loading an extension library, but what does loading an extension library mean? To talk about it, we need to dramatically roll back the talk to the physical world, and start with about links.

I think compiling C programs is, of course, not a new thing for you. Since I'm using `gcc` on Linux, I can create a runnable program in the following manner.

```
% gcc hello.c
```

According to the file name, this is probably an “Hello, World!” program. In UNIX, `gcc` outputs a program into a file named `a.out` by default, so you can subsequently execute it in the following way:

```
% ./a.out
Hello, World!
```

It is created properly.

By the way, what is `gcc` actually doing here? Usually we just say “compile” or “compile”, but actually

1. preprocess (`cpp`)
2. compile C into assembly (`cc`)
3. assemble the assembly language into machine code (`as`)
4. link (`ld`)

there are these four steps. Among them, preprocessing and compiling and assembling are described in a lot of places, but the description often ends without clearly describing about the linking phase. It is like a history class in school which would never reach “modern age”. Therefore, in this book, trying to provide the extinguished part, I’ll briefly summarize what is linking.

A program finished the assembling phase becomes an “object file” in somewhat format. The following formats are some of such formats which are major.

- ELF, Executable and Linking Format (recent UNIX)

- a.out, assembler output (relatively old UNIX)
- COFF, Common Object File Format (Win32)

It might go without saying that the a.out as an object file format and the a.out as a default output file name of cc are totally different things. For example, on modern Linux, when we create it ordinarily, the a.out file in ELF format is created.

And, how these object file formats differ each other is not important now. What we have to recognize now is, all of these object files can be considered as “a set of names”. For example, the function names and the variable names which exist in this file.

And, sets of names contained in the object file have two types.

- set of necessary names (for instance, the external functions called internally. e.g. printf)
- set of providing names (for instance, the functions defined internally. e.g. hello)

And linking is, when gathering multiple object files, checking if “the set of providing names” contains “the set of necessary names” entirely, and connecting them each other. In other words, pulling the lines from all of “the necessary names”, each line must be connected to one of “the providing names” of a particular object file. (Figure. 2) To put this in technical terms, it is resolving undefined symbols.

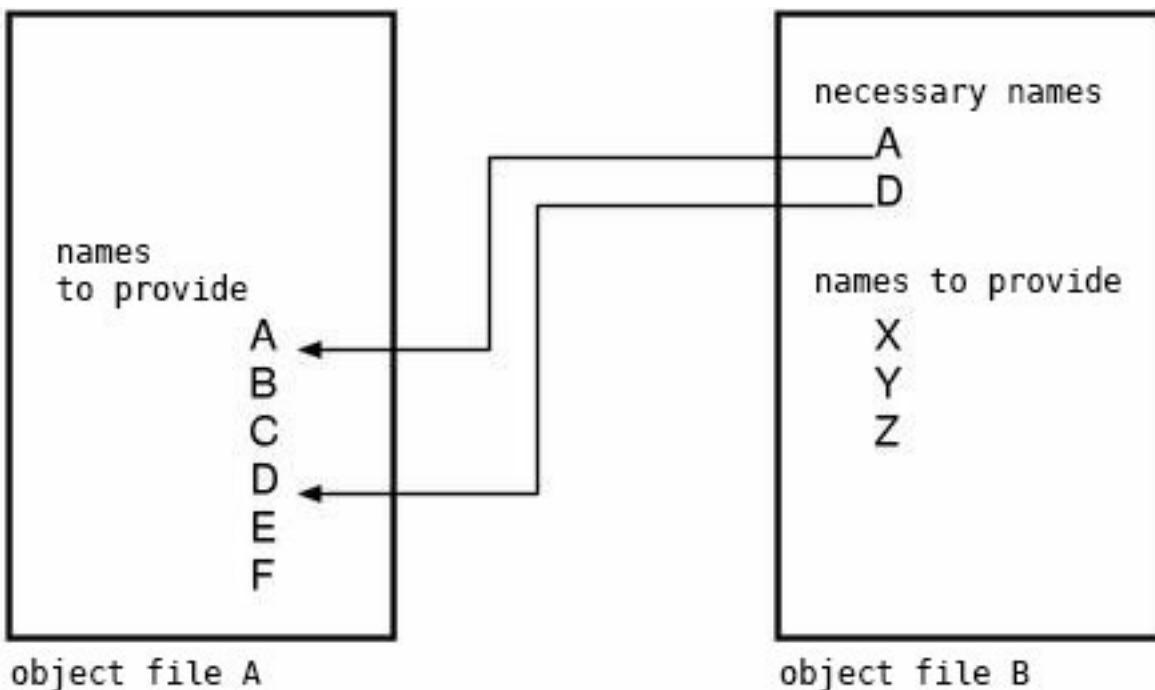


Figure 2: object files and linking

Logically this is how it is, but in reality a program can't run only because of this. At least, C programs cannot run without converting the names to the addresses (numbers).

So, after the logical conjunctions, the physical conjunctions become necessary. We have to map object files into the real memory space and substitute the all names with numbers. Concretely speaking, for instance, the addresses to jump to on function calls are adjusted here.

And, based on the timing when to do these two conjunctions, linking is divided into two types: static linking and dynamic linking. Static linking finishes the all phases during the compile time. On the other hand, dynamic linking defers some of the conjunctions to the executing time. And linking is finally completed when executing.

However, what explained here is a very simple idealistic model, and it has an aspect distorting the reality a lot. Logical conjunctions and physical conjunctions are not so completely separated, and “an object file is a set of names” is too naive. But the behavior around this considerably differs depending on each platform, describing seriously would end up with one more book. To obtain the realistic level knowledge, additionally, “Expert C Programming: Deep C Secrets” by Peter van der Linden, “Linkers and Loaders” by John R. Levine I recommend to read these books.

## ■ **Linking that is truly dynamic**

And finally we get into our main topic. The “dynamic” in “dynamic linking” naturally means it “occurs at execution time”, but what people usually refer to as “dynamic linking” is pretty much decided already at compile time. For example, the names of the needed functions, and which library they can be found in, are already known. For instance, if you need `cos()`, you know it’s in `libm`, so you use `gcc -lm`. If you didn’t specify the correct library at compile time, you’d get a link error.

But extension libraries are different. Neither the names of the needed functions, or the name of the library which defines them are known at compile time. We need to construct a string at execution time and load and link. It means that even “the logical conjunctions” in the sense of the previous words should be done entirely at execution time. In order to do it, another mechanism that is a little different from the ordinal dynamic linkings is

required.

This manipulation, linking that is entirely decided at runtime, is usually called “dynamic load”.

## ■ **Dynamic load API**

I've finished to explain the concept. The rest is how to do that dynamic loading. This is not a difficult thing. Usually there's a specific API prepared in the system, we can accomplish it by merely calling it.

For example, what is relatively broad for UNIX is the API named `dlopen`. However, I can't say “It is always available on UNIX”. For example, for a little previous HP-UX has a totally different interface, and a NeXT-flavor API is used on Mac OS X. And even if it is the same `dlopen`, it is included in `libc` on BSD-derived OS, and it is attached from outside as `libdl` on Linux. Therefore, it is desperately not portable. It differs even among UNIX-based platforms, it is obvious to be completely different in the other Operating Systems. It is unlikely that the same API is used.

Then, how `ruby` is doing is, in order to absorb the totally different interfaces, the file named `dln.c` is prepared. `dln` is probably the abbreviation of “dynamic link”. `dln_load()` is one of functions of `dln.c`.

Where dynamic loading APIs are totally different each other, the

only saving is the usage pattern of API is completely the same. Whichever platform you are on,

1. map the library to the address space of the process
2. take the pointers to the functions contained in the library
3. unmap the library

it consists of these there steps. For example, if it is `dlopen`-based API,

1. `dlopen`
2. `dlsym`
3. `dlclose`

are the correspondences. If it is Win32 API,

1. `LoadLibrary` (or `LoadLibraryEx`)
2. `GetProcAddress`
3. `FreeLibrary`

are the correspondences.

At last, I'll talk about what `dln_load()` is doing by using these APIs. It is, in fact, calling `Init_xxxx()`. By reaching here, we finally become to be able to illustrate the entire process of `ruby` from the invocation to the completion without any lacks. In other words, when `ruby` is invoked, it initializes the evaluator and starts evaluating a program passed in somewhat way. If `require` or `load` occurs during the process, it loads the library and transfers the

control. Transferring the control means parsing and evaluating if it is a Ruby library and it means loading and linking and finally calling `Init_xxxx()` if it is an extension library.

## ■ `dln_load()`

Finally, we've reached the content of `dln_load()`. `dln_load()` is also a long function, but its structure is simple because of some reasons. Take a look at the outline first.

### ▼ `dln_load()` (outline)

```
void*
dln_load(file)
    const char *file;
{
#if defined __WIN32 && !defined __CYGWIN__
    load with Win32 API
#else
    initialization depending on each platform
#endif each platform
    .....routines for each platform.....
#endif
#endif
#if !defined(_AIX) && !defined(NeXT)
    failed:
    rb_loaderror("%s - %s", error, file);
#endif
    return 0;                      /* dummy return */
}
```

This way, the part connecting to the main is completely separated based on each platform. When thinking, we only have to think about one platform at a time. Supported APIs are as follows:

- `dlopen` (Most of UNIX)
- `LoadLibrary` (Win32)
- `shl_load` (a bit old HP-UX)
- `a.out` (very old UNIX)
- `rld_load` (before NeXT4)
- `dyld` (NeXT or Mac OS X)
- `get_image_symbol` (BeOS)
- `GetDiskFragment` (Mac Os 9 and before)
- `load` (a bit old AIX)

## ■ `dln_load()`-`dlopen()`

First, let's start with the API code for the `dlopen` series.

### ▼ `dln_load()`-`dlopen()`

```

1254 void*
1255 dln_load(file)
1256     const char *file;
1257 {
1258     const char *error = 0;
1259 #define DLN_ERROR() (error = dln_strerror(), \
1260                         strcpy(ALLOC_N(char, strlen(error) + 1
1261                         buf, \
1262                         /* write a string "Init_xxxx" to buf (the space is alloc
1263                         init_funcname(&buf, file);
1264
1265     {
1266         void *handle;
1267         void (*init_fct)();
1268
1269 #ifndef RTLD_LAZY
1270 # define RTLD_LAZY 1

```

```

1310 #endif
1311 #ifndef RTLD_GLOBAL
1312 # define RTLD_GLOBAL 0
1313 #endif
1314
1315     /* (A) load the library */
1316     if ((handle = (void*)dlopen(file, RTLD_LAZY | RTLD_G
1317
1318         error = dln_strerror();
1319         goto failed;
1320     }
1321
1322     /* (B) get the pointer to Init_xxxx() */
1323     init_fct = (void(*)())dlsym(handle, buf);
1324     if (init_fct == NULL) {
1325         error = DLN_ERROR();
1326         dlclose(handle);
1327         goto failed;
1328     }
1329
1330     /* (C) call Init_xxxx() */
1331     (*init_fct)();
1332
1333     return handle;
1334 }
1576 failed:
1577     rb_loaderror("%s - %s", error, file);
1580 }

```

(dln.c)

(A) the `RTLD_LAZY` as the argument of `dlopen()` indicates “resolving the undefined symbols when the functions are actually demanded” The return value is the mark (handle) to distinguish the library and we always need to pass it when using `dl*`().

(B) `dlsym()` gets the function pointer from the library specified by the handle. If the return value is `NULL`, it means failure. Here,

getting the pointer to `Init_xxxx()` If the return value is `NULL`, it means failure. Here, the pointer to `Init_xxxx()` is obtained and called.

`dlclose()` is not called here. Since the pointers to the functions of the loaded library are possibly returned inside `Init_xxx()`, it is troublesome if `dlclose()` is done because the entire library would be disabled to use. Thus, we can't call `dlclose()` until the process will be finished.

## ■ `dln_load()` — Win32

As for Win32, `LoadLibrary()` and `GetProcAddress()` are used. It is very general Win32 API which also appears on MSDN.

### ▼ `dln_load()`-Win32

```
1254 void*
1255 dln_load(file)
1256     const char *file;
1257 {
1264     HINSTANCE handle;
1265     char winfile[MAXPATHLEN];
1266     void (*init_fct)();
1267     char *buf;
1268
1269     if (strlen(file) >= MAXPATHLEN) rb_loaderror("filename t
1270
1271     /* write the "Init_xxxx" string to buf (the space is all
1272     init_funcname(&buf, file);
1273
1274     strcpy(winfile, file);
1275
```

```
1276     /* load the library */
1277     if ((handle = LoadLibrary(winfile)) == NULL) {
1278         error = dln_strerror();
1279         goto failed;
1280     }
1281
1282     if ((init_fct = (void(*)())GetProcAddress(handle, buf))
1283         rb_loaderror("%s - %s\n%s", dln_strerror(), buf, file);
1284     }
1285
1286     /* call Init_xxxx() */
1287     (*init_fct)();
1288     return handle;
1289
1576 failed:
1577     rb_loaderror("%s - %s", error, file);
1580 }
```

(dln.c)

Doing `LoadLibrary()` then `GetProcAddress()`. The pattern is so equivalent that nothing is left to say, I decided to end this chapter.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License

# Ruby Hacking Guide

# Chapter 19: Threads

## Outline

---

### Ruby Interface

Come to think of it, I feel I have not introduced an actual code to use Ruby threads. This is not so special, but here I'll introduce it just in case.

```
Thread.fork {  
  while true  
    puts 'forked thread'  
  end  
}  
while true  
  puts 'main thread'  
end
```

When executing this program, a lot of "forked thread" and "main thread" are printed in the properly mixed state.

Of course, other than just creating multiple threads, there are also various ways to control. There's not the `synchronize` as a reserved word like Java, common primitives such as `Mutex` or `Queue` or `Monitor` are of course available, and the below APIs can be used to control a

thread itself.

## ▼ Thread API

<code>Thread.pass</code>	transfer the execution to any other thread
<code>Thread.kill(th)</code>	terminates the <code>th</code> thread
<code>Thread.exit</code>	terminates the thread itself
<code>Thread.stop</code>	temporarily stop the thread itself
<code>Thread#join</code>	waiting for the thread to finish
<code>Thread#wakeup</code>	to wake up the temporarily stopped thread

## ruby Thread

Threads are supposed to “run all together”, but actually they are running for a little time in turns. To be precise, by making some efforts on a machine of multi CPU, it’s possible that, for instance, two of them are running at the same time. But still, if there are more threads than the number of CPU, they have to run in turns.

In other words, in order to create threads, someone has to switch the threads in somewhere. There are roughly two ways to do it: kernel-level threads and user-level threads. They are respectively, as the names suggest, to create a thread in kernel or at user-level. If it is kernel-level, by making use of multi-CPU, multiple threads can run at the same time.

Then, how about the thread of `ruby`? It is user-level thread. And (Therefore), the number of threads that are runnable at the same time is limited to one.

# Is it preemptive?

I'll describe about the traits of ruby threads in more detail. As an alternative point of view of threads, there's the point that is "is it preemptive?".

When we say "thread (system) is preemptive", the threads will automatically be switched without being explicitly switched by its user. Looking this from the opposite direction, the user can't control the timing of switching threads.

On the other hand, in a non-preemptive thread system, until the user will explicitly say "I can pass the control right to the next thread", threads will never be switched. Looking this from the opposite direction, when and where there's the possibility of switching threads is obvious.

This distinction is also for processes, in that case, preemptive is considered as "superior". For example, if a program had a bug and it entered an infinite loop, the processes would never be able to switch. This means a user program can halt the whole system and is not good. And, switching processes was non-preemptive on Windows 3.1 because its base was MS-DOS, but Windows 95 is preemptive. Thus, the system is more robust. Hence, it is said that Windows 95 is "superior" to 3.1.

Then, how about the ruby thread? It is preemptive at Ruby-level, and non-preemptive at C level. In other words, when you are writing C code, you can determine almost certainly the timings of

switching threads.

Why is this designed in this way? Threads are indeed convenient, but its user also need to prepare certain minds. It means that it is necessary the code is compatible to the threads. (It must be multi-thread safe). In other words, in order to make it preemptive also in C level, the all C libraries have to be thread safe.

But in reality, there are also a lot of C libraries that are still not thread safe. A lot of efforts were made to ease to write extension libraries, but it would be brown if the number of usable libraries is decreased by requiring thread safety. Therefore, non-preemptive at C level is a reasonable choice for ruby.

## ■ Management System

We've understand ruby thread is non-preemptive at C level. It means after it runs for a while, it voluntarily let go of the controlling right. Then, I'd like you to suppose that now a currently being executed thread is about to quit the execution. Who will next receive the control right? But before that, it's impossible to guess it without knowing how threads are expressed inside ruby in the first place. Let's look at the variables and the data types to manage threads.

### ▼ the structure to manage threads

```
864  typedef struct thread * rb_thread_t;  
865  static rb_thread_t curr_thread = 0;
```

```
866 static rb_thread_t main_thread;
```

```
7301 struct thread {  
7302     struct thread *next, *prev;
```

(eval.c)

Since `struct thread` is very huge for some reason, this time I narrowed it down to the only important part. It is why there are only the two. These `next` and `prev` are member names, and their types are `rb_thread_t`, thus we can expect `rb_thread_t` is connected by a dual-directional link list. And actually it is not an ordinary dual-directional list, the both ends are connected. It means, it is circular. This is a big point. Adding the static `main_thread` and `curr_thread` variables to it, the whole data structure would look like Figure 1.

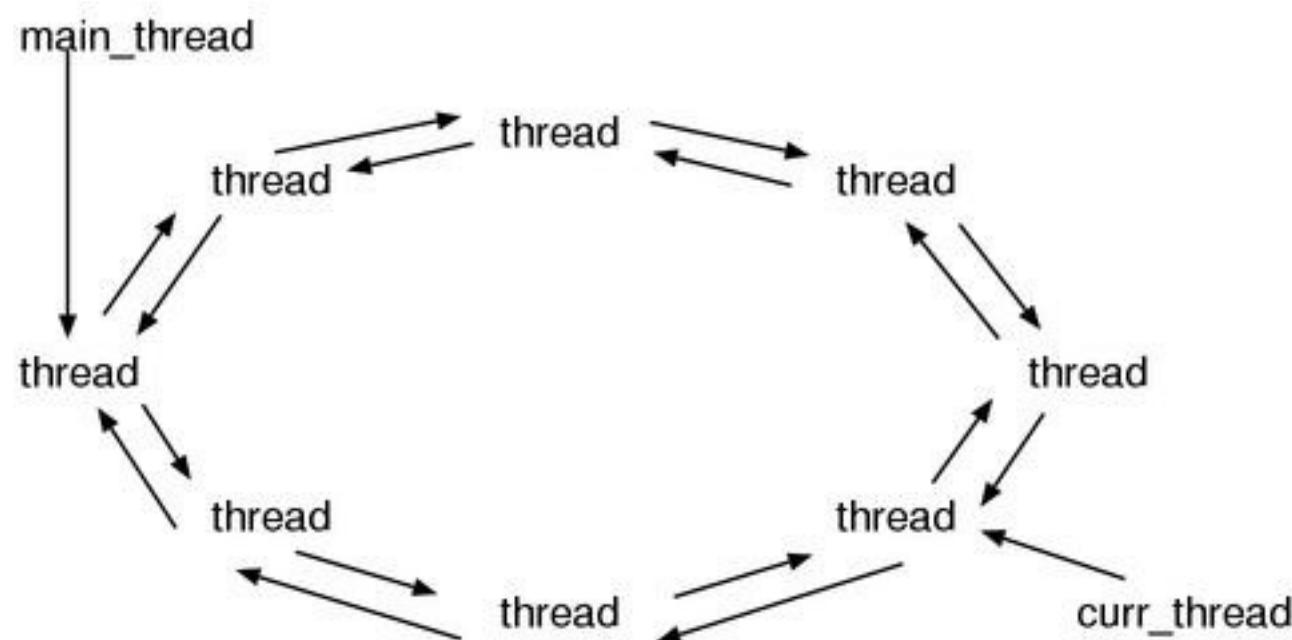


Figure 1: the data structures to manage threads

`main_thread` (main thread) means the thread existed at the time

when a program started, meaning the “first” thread. `curr_thread` is obviously current thread, meaning the thread currently running. The value of `main_thread` will never change while the process is running, but the value of `curr_thread` will change frequently.

In this way, because the list is being a circle, the procedure to chose “the next thread” becomes easy. It can be done by merely following the next link. Only by this, we can run all threads equally to some extent.

## What does switching threads mean?

By the way, what is a thread in the first place? Or, what makes us to say threads are switched?

These are very difficult questions. Similar to what a program is or what an object is, when asked about what are usually understood by feelings, it’s hard to answer clearly. Especially, “what is the difference between threads and processes?” is a good question.

Still, in a realistic range, we can describe it to some extent. What necessary for threads is the context of executing. As for the context of `ruby`, as we’ve seen by now, it consists of `ruby_frame` and `ruby_scope` and `ruby_class` and so on. And `ruby` allocates the substance of `ruby_frame` on the machine stack, and there are also the stack space used by extension libraries, therefore the machine stack is also necessary as a context of a Ruby program. And finally, the CPU registers are indispensable. These various contexts are the

elements to enable threads, and switching them means switching threads. Or, it is called “context-switch”.

## The way of context-switching

The rest talk is how to switch contexts. `ruby_scope` and `ruby_class` are easy to replace: allocate spaces for them somewhere such as the heap and set them aside one by one. For the CPU registers, we can make it because we can save and write back them by using `setjmp()`. The spaces for both purposes are respectively prepared in `rb_thread_t`.

### ▼ struct thread (partial)

```
7301 struct thread {
7302     struct thread *next, *prev;
7303     jmp_buf context;

7315     struct FRAME *frame;          /* ruby_frame */
7316     struct SCOPE *scope;          /* ruby_scope */
7317     struct RVarmap *dyna_vars;    /* ruby_dyna_vars */
7318     struct BLOCK *block;          /* ruby_block */
7319     struct iter *iter;            /* ruby_iter */
7320     struct tag *tag;              /* prot_tag */
7321     VALUE klass;                /* ruby_class */
7322     VALUE wrapper;              /* ruby_wrapper */
7323     NODE *cref;                 /* ruby_cref */

7324     int flags; /* scope_vmode / rb_trap_immediate / raised
7326
7327     NODE *node;                  /* rb_current_node */

7328     int tracing;                /* tracing */
7329     VALUE errinfo;              /* $! */
7330     VALUE last_status;          /* $? */
7331     VALUE last_line;            /* $_ */
```

```
7333     VALUE last_match;          /* $~ */
7334
7335     int safe;                  /* ruby_safe_level */
  
(eval.c)
```

As shown above, there are the members that seem to correspond to `ruby_frame` and `ruby_scope`. There's also a `jmp_buf` to save the registers.

Then, the problem is the machine stack. How can we substitute them?

The way which is the most straightforward for the mechanism is directly writing over the pointer to the position (end) of the stack. Usually, it is in the CPU registers. Sometimes it is a specific register, and it is also possible that a general-purpose register is allocated for it. Anyway, it is in somewhere. For convenience, we'll call it the stack pointer from now on. It is obvious that the different space can be used as the stack by modifying it. But it is also obvious in this way we have to deal with it for each CPU and for each OS, thus it is really hard to serve the portability.

Therefore, ruby uses a very violent way to implement the substitution of the machine stack. That is, if we can't modify the stack pointer, let's modify the place the stack pointer points to. We know the stack can be directly modified as we've seen in the description about the garbage collection, the rest is slightly changing what to do. The place to store the stack properly exists in `struct thread`.

## ▼ struct thread (partial)

```
7310     int     stk_len;          /* the stack length */  
7311     int     stk_max;          /* the size of memory allocated for  
7312     VALUE*stk_ptr;          /* the copy of the stack */  
7313     VALUE*stk_pos;          /* the position of the stack */
```

(eval.c)

## ■ How the explanation goes

So far, I've talked about various things, but the important points can be summarized to the three:

- When
- To which thread
- How

to switch context. These are also the points of this chapter. Below, I'll describe them using a section for each of the three points respectively.

## Trigger

---

To begin with, it's the first point, when to switch threads. In other words, what is the cause of switching threads.

# Waiting I/O

For example, when trying to read in something by calling `I0#gets` or `I0#read`, since we can expect it will take a lot of time to read, it's better to run the other threads in the meantime. In other words, a forcible switch becomes necessary here. Below is the interface of `getc`.

## ▼ `rb_getc()`

```
1185 int
1186 rb_getc(f)
1187     FILE *f;
1188 {
1189     int c;
1190
1191     if (!READ_DATA_PENDING(f)) {
1192         rb_thread_wait_fd(fileno(f));
1193     }
1194     TRAP_BEG;
1195     c = getc(f);
1196     TRAP_END;
1197
1198     return c;
1199 }
```

(io.c)

`READ_DATA_PENDING(f)` is a macro to check if the content of the buffer of the file is still there. If there's the content of the buffer, it means it can move without any waiting time, thus it would read it immediately. If it was empty, it means it would take some time, thus it would `rb_thread_wait_fd()`. This is an indirect cause of switching threads.

If `rb_thread_wait_fd()` is “indirect”, there also should be a “direct” cause. What is it? Let’s see the inside of `rb_thread_wait_fd()`.

## ▼ `rb_thread_wait_fd()`

```
8047 void
8048 rb_thread_wait_fd(fd)
8049     int fd;
8050 {
8051     if (rb_thread_critical) return;
8052     if (curr_thread == curr_thread->next) return;
8053     if (curr_thread->status == THREAD_TO_KILL) return;
8054
8055     curr_thread->status = THREAD_STOPPED;
8056     curr_thread->fd = fd;
8057     curr_thread->wait_for = WAIT_FD;
8058     rb_thread_schedule();
8059 }
```

(eval.c)

There’s `rb_thread_schedule()` at the last line. This function is the “direct cause”. It is the heart of the implementation of the ruby threads, and does select and switch to the next thread.

What makes us understand this function has such role is, in my case, I knew the word “scheduling” of threads beforehand. Even if you didn’t know, because you remembers now, you’ll be able to notice it at the next time.

And, in this case, it does not merely pass the control to the other thread, but it also stops itself. Moreover, it has an explicit deadline that is “by the time when it becomes readable”. Therefore, this

request should be told to `rb_thread_schedule()`. This is the part to assign various things to the members of `curr_thread`. The reason to stop is stored in `wait_for`, the information to be used when waking up is stored in `fd`, respectively.

## Waiting the other thread

After understanding threads are switched at the timing of `rb_thread_schedule()`, this time, conversely, from the place where `rb_thread_schedule()` appears, we can find the places where threads are switched. Then by scanning, I found it in the function named `rb_thread_join()`.

### ▼ `rb_thread_join()` (partial)

```
8227 static int
8228 rb_thread_join(th, limit)
8229     rb_thread_t th;
8230     double limit;
8231 {
8243     curr_thread->status = THREAD_STOPPED;
8244     curr_thread->join = th;
8245     curr_thread->wait_for = WAIT_JOIN;
8246     curr_thread->delay = timeofday() + limit;
8247     if (limit < DELAY_INFTY) curr_thread->wait_for |= WA
8248     rb_thread_schedule();
  
(eval.c)
```

This function is the substance of `Thread#join`, and `Thread#join` is a method to wait until the receiver thread will end. Indeed, since

there's time to wait, running the other threads is economy. Because of this, the second reason to switch is found.

## Waiting For Time

Moreover, also in the function named `rb_thread_wait_for()`, `rb_thread_schedule()` was found. This is the substance of (Ruby's) `sleep` and such.

### ▼ `rb_thread_wait_for` (simplified)

```
8080 void
8081 rb_thread_wait_for(time)
8082     struct timeval time;
8083 {
8084     double date;
8124     date = timeofday() +
8125         (double)time.tv_sec + (double)time.tv_usec*1e-6;
8126     curr_thread->status = THREAD_STOPPED;
8127     curr_thread->delay = date;
8128     curr_thread->wait_for = WAIT_TIME;
8129     rb_thread_schedule();
8129 }
```

(eval.c)

`timeofday()` returns the current time. Because the value of `time` is added to it, `date` indicates the time when the waiting time is over. In other words, this is the order “I'd like to stop until it will be the specific time”.

## Switch by expirations

In the above all cases, because some manipulations are done from Ruby level, consequently it causes to switch threads. In other words, by now, the Ruby-level is also non-preemptive. Only by this, if a program was to single-mindedly keep calculating, a particular thread would continue to run eternally. Therefore, we need to let it voluntary dispose the control right after running for a while. Then, how long a thread can run by the time when it will have to stop, is what I'll talk about next.

## setitimer

Since it is the same every now and then, I feel like lacking the skill to entertain, but I searched the places where calling `rb_thread_schedule()` further. And this time it was found in the strange place. It is here.

### ▼ `catch_timer()`

```
8574 static void
8575 catch_timer(sig)
8576     int sig;
8577 {
8578 #if !defined(POSIX_SIGNAL) && !defined(BSD_SIGNAL)
8579     signal(sig, catch_timer);
8580 #endif
8581     if (!rb_thread_critical) {
8582         if (rb_trap_immediate) {
8583             rb_thread_schedule();
8584         }
8585         else rb_thread_pending = 1;
8586     }
8587 }
```

This seems something relating to signals. What is this? I followed the place where this `catch_timer()` function is used, then it was used around here:

▼ `rb_thread_start_0()` (partial)

```
8620 static VALUE
8621 rb_thread_start_0(fn, arg, th_arg)
8622     VALUE (*fn)();
8623     void *arg;
8624     rb_thread_t th_arg;
8625 {
8632 #if defined(HAVE_SETITIMER)
8633     if (!thread_init) {
8634 #ifdef POSIX_SIGNAL
8635         posix_signal(SIGVTALRM, catch_timer);
8636 #else
8637         signal(SIGVTALRM, catch_timer);
8638 #endif
8639
8640         thread_init = 1;
8641         rb_thread_start_timer();
8642     }
8643 #endif
8644
8645 (eval.c)
```

This means, `catch_timer` is a signal handler of `SIGVTALRM`.

Here, “what kind of signal `SIGVTALRM` is” becomes the question. This is actually the signal sent when using the system call named `setitimer`. That’s why there’s a check of `HAVE_SETITIMER` just before it. `setitimer` is an abbreviation of “SET Interval TIMER” and a

system call to tell OS to send signals with a certain interval.

Then, where is the place calling `setitimer`? It is the `rb_thread_start_timer()`, which is coincidentally located at the last of this list.

To sum up all, it becomes the following scenario. `setitimer` is used to send signals with a certain interval. The signals are caught by `catch_timer()`. There, `rb_thread_schedule()` is called and threads are switched. Perfect.

However, signals could occur anytime, if it was based on only what described until here, it means it would also be preemptive at C level. Then, I'd like you to see the code of `catch_timer()` again.

```
if (rb_trap_immediate) {
    rb_thread_schedule();
}
else rb_thread_pending = 1;
```

There's a required condition that is doing `rb_thread_schedule()` only when it is `rb_trap_immediate`. This is the point. `rb_trap_immediate` is, as the name suggests, expressing “whether or not immediately process signals”, and it is usually false. It becomes true only while the limited time such as while doing I/O on a single thread. In the source code, it is the part between `TRAP_BEG` and `TRAP_END`.

On the other hand, since `rb_thread_pending` is set when it is false, let's follow this. This variable is used in the following place.

## ▼ CHECK\_INTS – HAVE\_SETITIMER

```
73 #if defined(HAVE_SETITIMER) && !defined(__B0W__)
74 EXTERN int rb_thread_pending;
75 # define CHECK_INTS do { \
76     if (!rb_prohibit_interrupt) { \
77         if (rb_trap_pending) rb_trap_exec(); \
78         if (rb_thread_pending && !rb_thread_critical) \
79             rb_thread_schedule(); \
80     } \
81 } while (0)
```

(rubysig.h)

This way, inside of `CHECK_INTS`, `rb_thread_pending` is checked and `rb_thread_schedule()` is done. It means, when receiving `SIGVTALRM`, `rb_thread_pending` becomes true, then the thread will be switched at the next time going through `CHECK_INTS`.

This `CHECK_INTS` has appeared at various places by now. For example, `rb_eval()` and `rb_call0()` and `rb_yeild_0`. `CHECK_INTS` would be meaningless if it was not located where the place frequently being passed. Therefore, it is natural to exist in the important functions.

## tick

We understood the case when there's `setitimer`. But what if `setitimer` does not exist? Actually, the answer is in `CHECK_INTS`, which we've just seen. It is the definition of the `#else` side.

## ▼ CHECK\_INTS – not HAVE\_SETITIMER

```
84 EXTERN int rb_thread_tick;
85 #define THREAD_TICK 500
86 #define CHECK_INTS do {\
87     if (!rb_prohibit_interrupt) {\\
88         if (rb_trap_pending) rb_trap_exec();\
89         if (!rb_thread_critical) {\\
90             if (rb_thread_tick-- <= 0) {\\
91                 rb_thread_tick = THREAD_TICK;\
92                 rb_thread_schedule();\
93             }\
94         }\
95     }\
96 } while (0)
```

(rubysig.h)

Every time going through `CHECK_INTS`, decrement `rb_thread_tick`. When it becomes 0, do `rb_thread_schedule()`. In other words, the mechanism is that the thread will be switched after `THREAD_TICK` (=500) times going through `CHECK_INTS`.

## Scheduling

---

The second point is to which thread to switch. What solely responsible for this decision is `rb_thread_schedule()`.

### rb\_thread\_schedule()

The important functions of ruby are always huge. This

rb\_thread\_schedule() has more than 220 lines. Let's exhaustively divide it into portions.

## ▼ rb\_thread\_schedule() (outline)

```
7819 void
7820 rb_thread_schedule()
7821 {
7822     rb_thread_t next;           /* OK */
7823     rb_thread_t th;
7824     rb_thread_t curr;
7825     int found = 0;
7826
7827     fd_set readfds;
7828     fd_set writefds;
7829     fd_set exceptfds;
7830     struct timeval delay_tv, *delay_ptr;
7831     double delay, now; /* OK */
7832     int n, max;
7833     int need_select = 0;
7834     int select_timeout = 0;
7835
7836     rb_thread_pending = 0;
7837     if (curr_thread == curr_thread->next
7838         && curr_thread->status == THREAD_RUNNABLE)
7839         return;
7840
7841     next = 0;
7842     curr = curr_thread;          /* starting thread */
7843
7844     while (curr->status == THREAD_KILLED) {
7845         curr = curr->prev;
7846     }
7847
7848     /* .....prepare the variables used at select ..... */
7849     /* .....select if necessary ..... */
7850     /* .....decide the thread to invoke next ..... */
7851     /* .....context-switch ..... */
8045 }
```

- (A) When there's only one thread, this does not do anything and returns immediately. Therefore, the talks after this can be thought based on the assumption that there are always multiple threads.
- (B) Subsequently, the initialization of the variables. We can consider the part until and including the `while` is the initialization. Since `cur` is following `prev`, the last alive thread (`status != THREAD_KILLED`) will be set. It is not “the first” one because there are a lot of loops that “start with the next of `curr` then deal with `curr` and end”.

After that, we can see the sentences about `select`. Since the thread switch of `ruby` is considerably depending on `select`, let's first study about `select` in advance here.

## ■ `select`

`select` is a system call to wait until the preparation for reading or writing a certain file will be completed. Its prototype is this:

```
int select(int max,
           fd_set *readset, fd_set *writeset, fd_set *exceptset,
           struct timeval *timeout);
```

In the variable of type `fd_set`, a set of `fd` that we want to check is stored. The first argument `max` is “(the maximum value of `fd` in `fd_set`) + 1”. The `timeout` is the maximum waiting time of `select`. If `timeout` is `NULL`, it would wait eternally. If `timeout` is `0`, without

waiting for even just a second, it would only check and return immediately. As for the return value, I'll talk about it at the moment when using it.

I'll talk about `fd_set` in detail. `fd_set` can be manipulated by using the below macros:

### ▼ `fd_set` manipulation

```
fd_set set;  
  
FD_ZERO(&set)          /* initialize */  
FD_SET(fd, &set)        /* add a file descriptor fd to the set */  
FD_ISSET(fd, &set)      /* true if fd is in the set */
```

`fd_set` is typically a bit array, and when we want to check n-th file descriptor, the n-th bit is set (Figure 2).

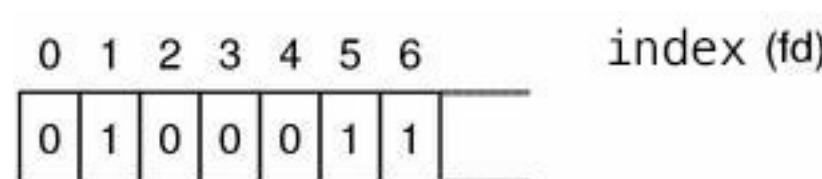


Figure 2: `fd_set`

I'll show a simple usage example of `select`.

### ▼ a usage example of `select`

```
#include <stdio.h>  
#include <sys/types.h>  
#include <sys/time.h>  
#include <unistd.h>
```

```
int
main(int argc, char **argv)
{
    char *buf[1024];
    fd_set readset;

    FD_ZERO(&readset);           /* initialize readset */
    FD_SET(STDIN_FILENO, &readset); /* put stdin into the set */
    select(STDIN_FILENO + 1, &readset, NULL, NULL, NULL);
    read(STDIN_FILENO, buf, 1024); /* success without delay */
    exit(0);
}
```

This code assume the system call is always success, thus there are not any error checks at all. I'd like you to see only the flow that is `FD_ZERO` → `FD_SET` → `select`. Since here the fifth argument `timeout` of `select` is `NULL`, this `select` call waits eternally for reading `stdin`. And since this `select` is completed, the next `read` does not have to wait to read at all. By putting `print` in the middle, you will get further understandings about its behavior. And a little more detailed example code is put in the attached CD-ROM {see also `doc/select.html`}.

## ■ **Preparations for select**

Now, we'll go back to the code of `rb_thread_schedule()`. Since this code branches based on the reason why threads are waiting. I'll show the content in shortened form.

### ▼ `rb_thread_schedule()` – preparations for `select`

```
7848 again:
7849     /* initialize the variables relating to select */
7850     max = -1;
7851     FD_ZERO(&readfds);
7852     FD_ZERO(&writefds);
7853     FD_ZERO(&exceptfds);
7854     delay = DELAY_INFTY;
7855     now = -1.0;
7856
7857     FOREACH_THREAD_FROM(curr, th) {
7858         if (!found && th->status <= THREAD_RUNNABLE) {
7859             found = 1;
7860         }
7861         if (th->status != THREAD_STOPPED) continue;
7862         if (th->wait_for & WAIT_JOIN) {
7863             /* .....join wait..... */
7864         }
7865         if (th->wait_for & WAIT_FD) {
7866             /* .....I/O wait..... */
7867         }
7868         if (th->wait_for & WAIT_SELECT) {
7869             /* .....select wait..... */
7870         }
7871         if (th->wait_for & WAIT_TIME) {
7872             /* .....time wait..... */
7873         }
7874     }
7875     END_FOREACH_FROM(curr, th);
```

(eval.c)

Whether it is supposed to be or not, what stand out are the macros named FOREACH-some. These two are defined as follows:

## ▼ FOREACH\_THREAD\_FROM

```
7360 #define FOREACH_THREAD_FROM(f,x) x = f; do { x = x->next;
7361 #define END_FOREACH_FROM(f,x) } while (x != f)
```

Let's extract them for better understandability.

```
th = curr;
do {
    th = th->next;
    {
        ....
    }
} while (th != curr);
```

This means: follow the circular list of threads from the next of `curr` and process `curr` at last and end, and meanwhile the `th` variable is used. This makes me think about the Ruby's iterators ... is this my too much imagination?

Here, we'll go back to the subsequence of the code, it uses this a bit strange loop and checks if there's any thread which needs `select`. As we've seen previously, since `select` can wait for reading/writing/exception/time all at once, you can probably understand I/O waits and time waits can be centralized by single `select`. And though I didn't describe about it in the previous section, `select` waits are also possible. There's also a method named `IO.select` in the Ruby's library, and you can use `rb_thread_select()` at C level. Therefore, we need to execute that `select` at the same time. By merging `fd_set`, multiple `select` can be done at once.

The rest is only `join` wait. As for its code, let's see it just in case.

## ▼ rb\_thread\_schedule() – select preparation – join wait

```
7861         if (th->wait_for & WAIT_JOIN) {  
7862             if (rb_thread_dead(th->join)) {  
7863                 th->status = THREAD_RUNNABLE;  
7864                 found = 1;  
7865             }  
7866         }  
  
(eval.c)
```

The meaning of `rb_thread_dead()` is obvious because of its name. It determines whether or not the thread of the argument has finished.

## █ Calling select

By now, we've figured out whether `select` is necessary or not, and if it is necessary, its `fd_set` has already prepared. Even if there's a immediately invocable thread (`THREAD_RUNNABLE`), we need to call `select` beforehand. It's possible that there's actually a thread that it has already been while since its I/O wait finished and has the higher priority. But in that case, tell `select` to immediately return and let it only check if I/O was completed.

## ▼ rb\_thread\_schedule() – select

```
7904     if (need_select) {  
7905         /* convert delay into timeval */  
7906         /* if theres immediately invocable threads, do only  
7907             if (found) {  
7908                 delay_tv.tv_sec = 0;
```

```

7909             delay_tv.tv_usec = 0;
7910             delay_ptr = &delay_tv;
7911         }
7912         else if (delay == DELAY_INFTY) {
7913             delay_ptr = 0;
7914         }
7915         else {
7916             delay_tv.tv_sec = delay;
7917             delay_tv.tv_usec = (delay - (double)delay_tv.tv_
7918             delay_ptr = &delay_tv;
7919         }
7920
7921         n = select(max+1, &readfds, &writefds, &exceptfds, d
7922         if (n < 0) {
7944             /* ..... being cut in by signal or something ..... */
7945             }
7946             if (select_timeout && n == 0) {
7947                 /* ..... timeout ..... */
7948             }
7949             if (n > 0) {
7950                 /* ..... properly finished ..... */
7951             }
7952             /* In a somewhere thread, its I/O wait has finished.
7953                 roll the loop again to detect the thread */
7954             if (!found && delay != DELAY_INFTY)
7955                 goto again;
7956         }

```

(eval.c)

The first half of the block is as written in the comment. Since `delay` is the `usec` until the any thread will be next invocable, it is converted into `timeval` form.

In the last half, it actually calls `select` and branches based on its result. Since this code is long, I divided it again. When being cut in by a signal, it either goes back to the beginning then processes again or becomes an error. What are meaningful are the rest two.

# Timeout

When `select` is timeout, a thread of time wait or `select` wait may become invocable. Check about it and search runnable threads. If it is found, set `THREAD_RUNNABLE` to it.

## Completing normally

If `select` is normally completed, it means either the preparation for I/O is completed or `select` wait ends. Search the threads that are no longer waiting by checking `fd_set`. If it is found, set `THREAD_RUNNABLE` to it.

## Decide the next thread

Taking all the information into considerations, eventually decide the next thread to invoke. Since all what was invocable and all what had finished waiting and so on became `RUNNABLE`, you can arbitrary pick up one of them.

### ▼ `rb_thread_schedule()` – decide the next thread

```
7996      FOREACH_THREAD_FROM(curr, th) {  
7997          if (th->status == THREAD_TO_KILL) { /*  
7998              next = th;  
7999              break;  
8000          }  
8001          if (th->status == THREAD_RUNNABLE && th->stk_ptr) {  
8002              if (!next || next->priority < th->priority) /*  
8003                  next = th;  
8004          }
```

```
8005      }
8006      END_FOREACH_FROM(curr, th);
(eval.c)
```

- (A) if there's a thread that is about to finish, give it the high priority and let it finish.
- (B) find out what seems runnable. However it seems to consider the value of `priority`. This member can also be modified from Ruby level by using `Tread#priority` `Thread#priority=`. ruby itself does not especially modify it.

If these are done but the next thread could not be found, in other words if the `next` was not set, what happen? Since `select` has already been done, at least one of threads of time wait or I/O wait should have finished waiting. If it was missing, the rest is only the waits for the other threads, and moreover there's no runnable threads, thus this wait will never end. This is a dead lock.

Of course, for the other reasons, a dead lock can happen, but generally it's very hard to detect a dead lock. Especially in the case of ruby, `Mutex` and such are implemented at Ruby level, the perfect detection is nearly impossible.

## Switching Threads

The next thread to invoke has been determined. I/O and `select` checks has also been done. The rest is transferring the control to the target thread. However, for the last of `rb_thread_schedule()` and

the code to switch threads, I'll start a new section.

## Context Switch

---

The last third point is thread-switch, and it is context-switch. This is the most interesting part of threads of ruby.

### ▀ The Base Line

Then we'll start with the tail of `rb_thread_schedule()`. Since the story of this section is very complex, I'll go with a significantly simplified version.

#### ▼ `rb_thread_schedule()` (context switch)

```
if (THREAD_SAVE_CONTEXT(curr)) {  
    return;  
}  
rb_thread_restore_context(next, RESTORE_NORMAL);
```

As for the part of `THREAD_SAVE_CONTEXT()`, we need to extract the content at several places in order to understand.

#### ▼ `THREAD_SAVE_CONTEXT()`

```
7619 #define THREAD_SAVE_CONTEXT(th) \  
7620     (rb_thread_save_context(th), thread_switch(setjmp((th)->c
```

```
7587 static int
7588 thread_switch(n)
7589     int n;
7590 {
7591     switch (n) {
7592         case 0:
7593             return 0;
7594         case RESTORE_FATAL:
7595             JUMP_TAG(TAG_FATAL);
7596             break;
7597         case RESTORE_INTERRUPT:
7598             rb_interrupt();
7599             break;
7600             /* ..... process various abnormal things ..... */
7601         case RESTORE_NORMAL:
7602             default:
7603                 break;
7604         }
7605     return 1;
7606 }
```

(eval.c)

If I merge the three then extract it, here is the result:

```
rb_thread_save_context(curr);
switch (setjmp(curr->context)) {
    case 0:
        break;
    case RESTORE_FATAL:
        .....
    case RESTORE_INTERRUPT:
        .....
    /* .....process abnormals..... */
    case RESTORE_NORMAL:
        default:
            return;
}
rb_thread_restore_context(next, RESTORE_NORMAL);
```

At both of the return value of `setjmp()` and `rb_thread_restore_context()`, `RESTORE_NORMAL` appears, this is clearly suspicious. Since it does `longjmp()` in `rb_thread_restore_context()`, we can expect the correspondence between `setjmp()` and `longjmp()`. And if we will imagine the meaning also from the function names,

save the context of the current thread

`setjmp`

restore the context of the next thread

`longjmp`

The rough main flow would probably look like this. However what we have to be careful about here is, this pair of `setjmp()` and `longjmp()` is not completed in this thread. `setjmp()` is used to save the context of this thread, `longjmp()` is used to restore the context of the next thread. In other words, there's a chain of `setjmp/longjmp()` as follows. (Figure 3)

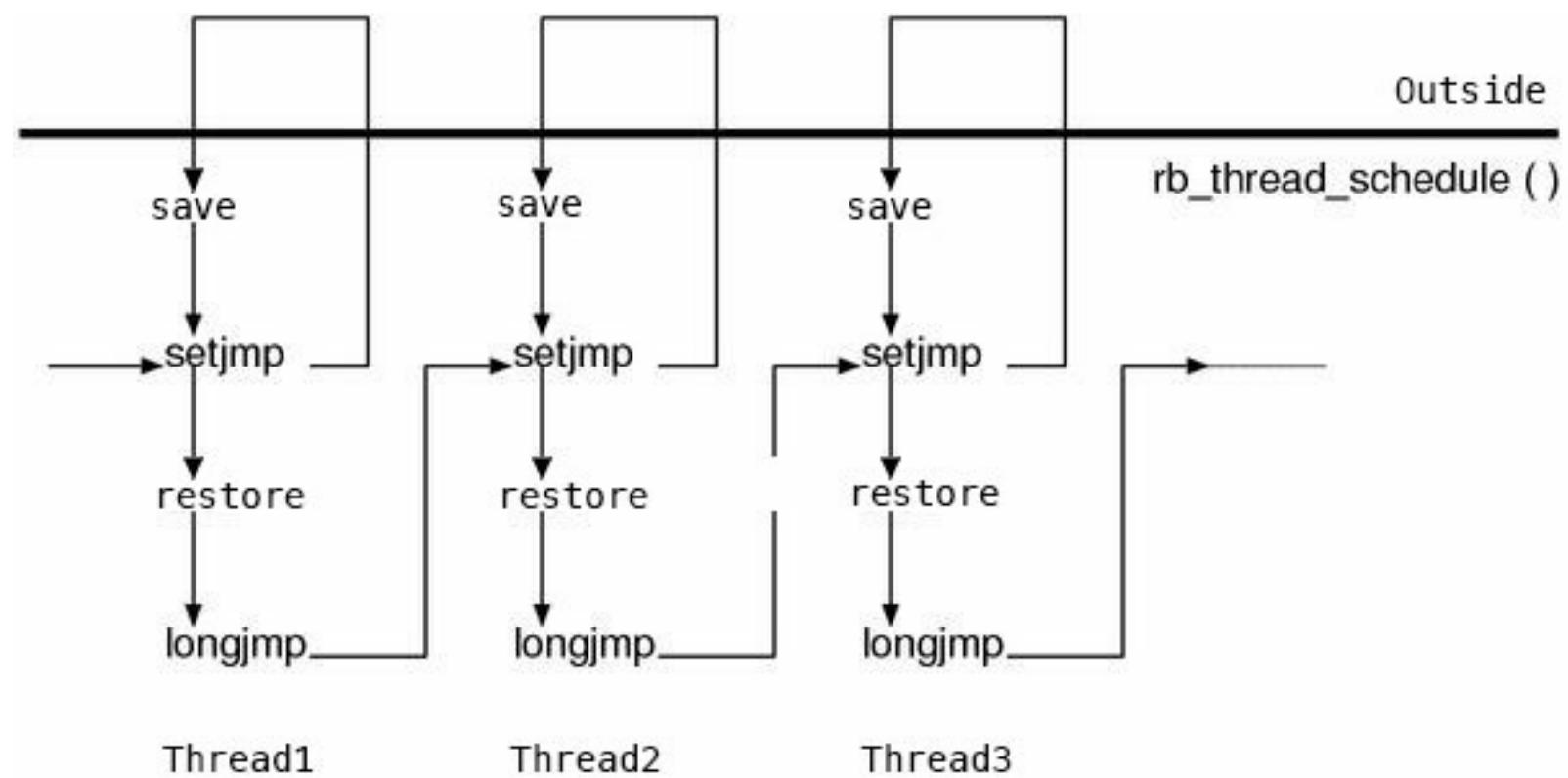


Figure 3: the backstitch by chaining of `setjmp`

We can restore around the CPU registers with `setjmp()`/`longjmp()`, so the remaining context is the Ruby stacks in addition to the machine stack. `rb_thread_save_context()` is to save it, and `rb_thread_restore_context()` is to restore it. Let's look at each of them in sequential order.

## ■ `rb_thread_save_context()`

Now, we'll start with `rb_thread_save_context()`, which saves a context.

### ▼ `rb_thread_save_context()` (simplified)

```
7539 static void
7540 rb_thread_save_context(th)
7541     rb_thread_t th;
```

```
7542  {
7543      VALUE *pos;
7544      int len;
7545      static VALUE tval;
7546
7547      len = ruby_stack_length(&pos);
7548      th->stk_len = 0;
7549      th->stk_pos = (rb_gc_stack_start<pos)?rb_gc_stack_start
7550                                :rb_gc_stack_start
7551      if (len > th->stk_max) {
7552          REALLOC_N(th->stk_ptr, VALUE, len);
7553          th->stk_max = len;
7554      }
7555      th->stk_len = len;
7556      FLUSH_REGISTER_WINDOWS;
7557      MEMCPY(th->stk_ptr, th->stk_pos, VALUE, th->stk_len);

      /* .....omission..... */
}
```

(eval.c)

The last half is just keep assigning the global variables such as `ruby_scope` into `th`, so it is omitted because it is not interesting. The rest, in the part shown above, it attempts to copy the entire machine stack into the place where `th->stk_ptr` points to.

First, it is `ruby_stack_length()` which writes the head address of the stack into the parameter `pos` and returns its length. The range of the stack is determined by using this value and the address of the bottom-end side is set to `th->stk_ptr`. We can see some branches, it is because both a stack extending higher and a stack extending lower are possible. (Figure 4)

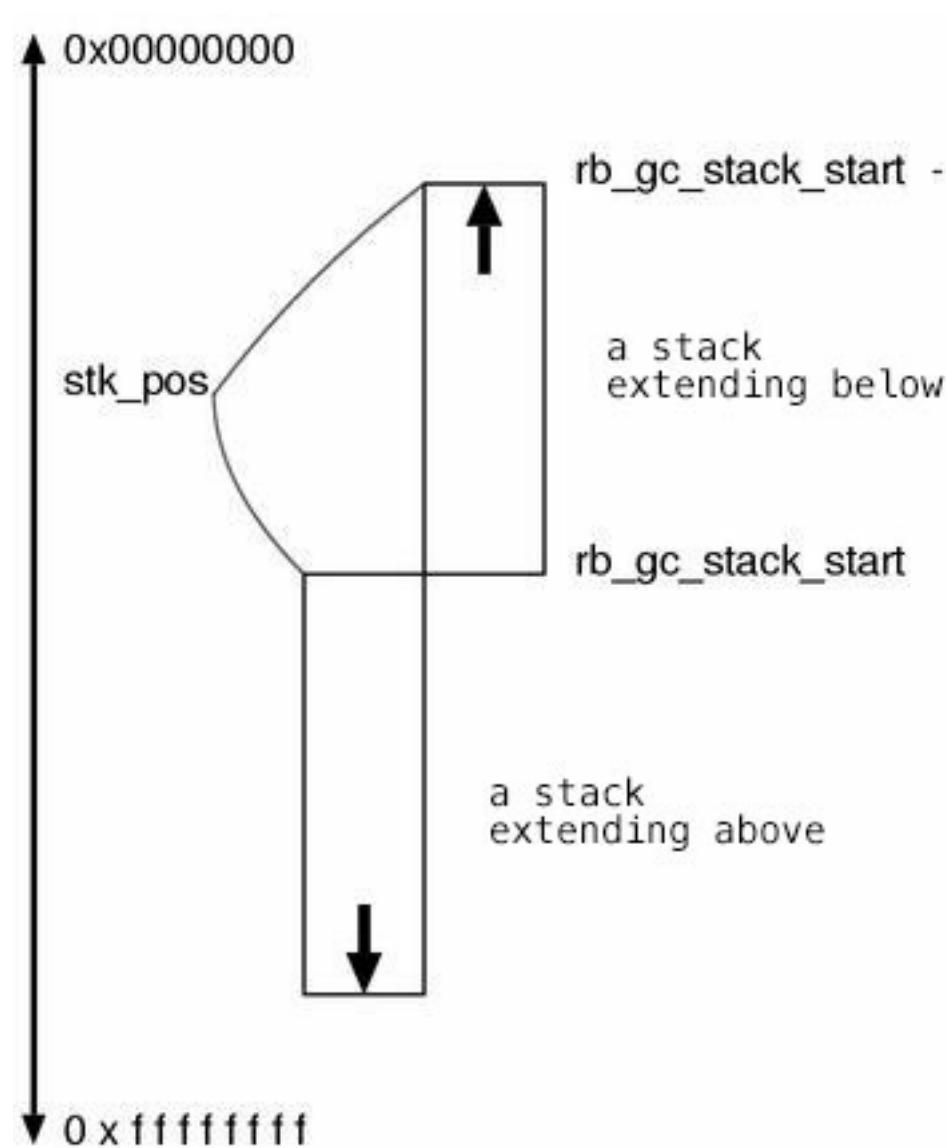


Fig.4: a stack extending above and a stack extending below

After that, the rest is allocating a memory in where `th->stkptr` points to and copying the stack: allocate the memory whose size is `th->stk_max` then copy the stack by the `len` length.

`FLUSH_REGISTER_WINDOWS` was described in Chapter 5: Garbage collection, so its explanation might no longer be necessary. This is a macro (whose substance is written in Assembler) to write down the cache of the stack space to the memory. It must be called when the target is the entire stack.

## rb\_thread\_restore\_context()

And finally, it is `rb_thread_restore_context()`, which is the function to restore a thread.

### ▼ rb\_thread\_restore\_context()

```
7635 static void
7636 rb_thread_restore_context(th, exit)
7637     rb_thread_t th;
7638     int exit;
7639 {
7640     VALUE v;
7641     static rb_thread_t tmp;
7642     static int ex;
7643     static VALUE tval;
7644
7645     if (!th->stk_ptr) rb_bug("unsaved context");
7646
7647     if (&v < rb_gc_stack_start) {
7648         /* the machine stack extending lower */
7649         if (&v > th->stk_pos) stack_extend(th, exit);
7650     }
7651     else {
7652         /* the machine stack extending higher */
7653         if (&v < th->stk_pos + th->stk_len) stack_extend(th,
7654     }
7655
7656     /* omission ..... back the global variables */
7657
7658     tmp = th;
7659     ex = exit;
7660     FLUSH_REGISTER_WINDOWS;
7661     MEMCPY(tmp->stk_pos, tmp->stk_ptr, VALUE, tmp->stk_len);
7662
7663     tval = rb_lastline_get();
7664     rb_lastline_set(tmp->last_line);
7665     tmp->last_line = tval;
7666     tval = rb_backref_get();
7667     rb_backref_set(tmp->last_match);
```

```
7687     tmp->last_match = tval;
7688
7689     longjmp(tmp->context, ex);
7690 }
```

(eval.c)

The `th` parameter is the target to give the execution back. `MEMCPY()` and `longjmp()` in the last half are at the heart. The closer `MEMCPY()` to the last, the better it is, because after this manipulation, the stack is in a destroyed state until `longjmp()`.

Nevertheless, there are `rb_lastline_set()` and `rb_backref_set()`. They are the restorations of `$_` and `$~`. Since these two variables are not only local variables but also thread local variables, even if it is only a single local variable slot, there are its as many slots as the number of threads. This must be here because the place actually being written back is the stack. Because they are local variables, their slot spaces are allocated with `alloca()`.

That's it for the basics. But if we merely write the stack back, in the case when the stack of the current thread is shorter than the stack of the thread to switch to, the stack frame of the very currently executing function (it is `rb_thread_restore_context`) would be overwritten. It means the content of the `th` parameter will be destroyed. Therefore, in order to prevent this from occurring, we first need to extend the stack. This is done by the `stack_extend()` in the first half.

▼ `stack_extend()`

```
7624 static void
7625 stack_extend(rb_thread_t th, int exit)
7626     rb_thread_t th;
7627     int exit;
7628 {
7629     VALUE space[1024];
7630
7631     memset(space, 0, 1); /* prevent array from optimi
7632     rb_thread_restore_context(th, exit);
7633 }
```

(eval.c)

By allocating a local variable (which will be put at the machine stack space) whose size is 1K, forcibly extend the stack. However, though this is a matter of course, doing return from `stack_extend()` means the extended stack will shrink immediately. This is why `rb_thread_restore_context()` is called again immediately in the place.

By the way, the completion of the task of `rb_thread_restore_context()` means it has reached the call of `longjmp()`, and once it is called it will never return back. Obviously, the call of `stack_extend()` will also never return. Therefore, `rb_thread_restore_context()` does not have to think about such as possible procedures after returning from `stack_extend()`.

## Issues

This is the implementation of the ruby thread switch. We can't think it is lightweight. Plenty of `malloc()` `realloc()` and plenty of

`memcpy()` and doing `setjmp()` `longjmp()` then furthermore calling functions to extend the stack. There's no problem to express "It is deadly heavy". But instead, there's not any system call depending on a particular OS, and there are just a few assembly only for the register windows of Sparc. Indeed, this seems to be highly portable.

There's another problem. It is, because the stacks of all threads are allocated to the same address, there's the possibility that the code using the pointer to the stack space is not runnable. Actually, Tcl/Tk excellently matches this situation, in order to bypass, Ruby's Tcl/Tk interface reluctantly chooses to access only from the main thread.

Of course, this does not go along with native threads. It would be necessary to restrict `ruby` threads to run only on a particular native thread in order to let them work properly. In UNIX, there are still a few libraries that use a lot of threads. But in Win32, because threads are running every now and then, we need to be careful about it.

The original work is Copyright © 2002 - 2004 Minero AOKI.  
Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

[Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License](https://creativecommons.org/licenses/by-nc-sa/2.5/)

# Ruby Hacking Guide

## Final Chapter: Ruby's future

### Issues to be addressed

---

ruby isn't 'completely finished software'. It's still being developed, there are still a lot of issues. Firstly, we want to try removing inherent problems in the current interpreter.

The order of the topics is mostly in the same order as the chapters of this book.

#### ■ Performance of GC

The performance of the current GC might be “not notably bad, but not notably good”. “not notably bad” means “it won't cause troubles in our daily life”, and “not notably good” means “its downside will be exposed under heavy load”. For example, if it is an application

which creates plenty of objects and keeps holding them, its speed would slow down radically. Every time doing GC, it needs to mark all of the objects, and furthermore it would becomes to need to invoke GC more often because it can't collect them. To counter this problem, Generational GC, which was mentioned in Chapter 5, must be effective. (At least, it is said so in theory.)

Also regarding its response speed, there are still rooms we can improve. With the current GC, while it is running, the entire interpreter stops. Thus, when the program is an editor or a GUI application, sometimes it freezes and stops to react. Even if it's just 0.1 second, stopping when typing characters would give a very bad impression. Currently, there are few such applications created or, even if exists, its size might be enough small not to expose this problem. However, if such application will actually be created in the future, there might be the necessity to consider Incremental GC.

## ■ Implementation of parser

As we saw in Part 2, the implementation of `ruby` parser has already utilized `@yacc@`'s ability to almost its limit, thus I can't think it can endure further expansions. It's all right if there's nothing planned to expand, but a big name "keyword argument" is planned next and it's sad if we could not express another demanded grammar because of the limitation of yacc.

## ■ Reuse of parser

Ruby's parser is very complex. In particular, dealing with around `lex_state` seriously is very hard. Due to this, embedding a Ruby program or creating a program to deal with a Ruby program itself is quite difficult.

For example, I'm developing a tool named `racc`, which is prefixed with R because it is a Ruby-version `yacc`. With `racc`, the syntax of grammar files are almost the same as `yacc` but we can write actions in Ruby. To do so, it could not determine the end of an action without parsing Ruby code properly, but “properly” is very difficult. Since there's no other choice, currently I've compromised at the level that it can parse “almost all”.

As another example which requires analyzing Ruby program, I can enumerate some tools like `indent` and `lint`, but creating such tool also requires a lot efforts. It would be desperate if it is something complex like a refactoring tool.

Then, what can we do? If we can't recreate the same thing, what if @ruby@'s original parser can be used as a component? In other words, making the parser itself a library. This is a feature we want by all means.

However, what becomes problem here is, as long as `yacc` is used, we cannot make parser reentrant. It means, say, we cannot call `yyparse()` recursively, and we cannot call it from multiple threads. Therefore, it should be implemented in the way of not returning

control to Ruby while parsing.

## Hiding Code

With current ruby, it does not work without the source code of the program to run. Thus, people who don't want others to read their source code might have trouble.

## Interpreter Object

Currently each process cannot have multiple ruby interpreters, this was discussed in Chapter 13. If having multiple interpreters is practically possible, it seems better, but is it possible to implement such thing?

## The structure of evaluator

Current `eval.c` is, above all, too complex. Embedding Ruby's stack frames to machine stack could occasionally become the source of trouble, using `setjmp()` `longjmp()` aggressively makes it less easy to understand and slows down its speed. Particularly with RISC machine, which has many registers, using `setjmp()` aggressively can easily cause slowing down because `setjmp()` set aside all things in registers.

## The performance of evaluator

ruby is already enough fast for ordinary use. But aside from it,

regarding a language processor, definitely the faster is the better. To achieve better performance, in other words to optimize, what can we do? In such case, the first thing we have to do is profiling. So I profiled.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
20.25	1.64	1.64	2638359	0.00	0.00	rb_eval
12.47	2.65	1.01	1113947	0.00	0.00	ruby_re_ma
8.89	3.37	0.72	5519249	0.00	0.00	rb_call0
6.54	3.90	0.53	2156387	0.00	0.00	st_lookup
6.30	4.41	0.51	1599096	0.00	0.00	rb_yield_0
5.43	4.85	0.44	5519249	0.00	0.00	rb_call
5.19	5.27	0.42	388066	0.00	0.00	st_foreach
3.46	5.55	0.28	8605866	0.00	0.00	rb_gc_mark
2.22	5.73	0.18	3819588	0.00	0.00	call_cfunc

This is a profile when running some application but this is approximately the profile of a general Ruby program. `rb_eval()` appeared in the overwhelming percentage being at the top, after that, in addition to functions of GC, evaluator core, functions that are specific to the program are mixed. For example, in the case of this application, it takes a lot of time for regular expression match (`ruby_re_match`).

However, even if we understood this, the question is how to improve it. To think simply, it can be archived by making `rb_eval()` faster. That said, but as for ruby core, there are almost not any room which can be easily optimized. For instance, apparently “tail recursive → goto conversion” used in the place of `NODE_IF` and others has already applied almost all possible places it can be

applied. In other words, without changing the way of thinking fundamentally, there's no room to improve.

## ■ The implementation of thread

This was also discussed in Chapter 19. There are really a lot of issues about the implementation of the current ruby's thread. Particularly, it cannot mix with native threads so badly. The two great advantages of @ruby@'s thread, (1) high portability (2) the same behavior everywhere, are definitely incomparable, but probably that implementation is something we cannot continue to use eternally, isn't it?

## ruby 2

---

Subsequently, on the other hand, I'll introduce the trend of the original ruby, how it is trying to counter these issues.

## ■ Rite

At the present time, ruby's edge is 1.6.7 as the stable version and 1.7.3 as the development version, but perhaps the next stable version 1.8 will come out in the near future. Then at that point, the next development version 1.9.0 will start at the same time. And after that, this is a little irregular but 1.9.1 will be the next stable version.

## stable development

1.6.x	1.7.x
1.8.x	1.9.x
1.9.1~	2.0.0

## when to start

1.6.0 was released on 2000-09-19  
probably it will come out within 6 months  
maybe about 2 years later

And the next-to-next generational development version is `ruby 2`, whose code name is Rite. Apparently this name indicates a respect for the inadequacy that Japanese cannot distinguish the sounds of L and R.

What will be changed in 2.0 is, in short, almost all the entire core. Thread, evaluator, parser, all of them will be changed. However, nothing has been written as a code yet, so things written here is entirely just a “plan”. If you expect so much, it’s possible it will turn out disappointments. Therefore, for now, let’s just expect slightly.

## The language to write

Firstly, the language to use. Definitely it will be C. Mr. Matsumoto said to `ruby-talk`, which is the English mailing list for Ruby,

I hate C++.

So, C++ is most unlikely. Even if all the parts will be recreated, it is reasonable that the object system will remain almost the same, so not to increase extra efforts around this is necessary. However, chances are good that it will be ANSI C next time.

Regarding the implementation of GC, the good start point would be Boehm GC\footnote{Boehm GC [http://www.hpl.hp.com/personal/Hans\\_Boehm/gc](http://www.hpl.hp.com/personal/Hans_Boehm/gc)}. Boehm GC is a conservative and incremental and generational GC, furthermore, it can mark all stack spaces of all threads even while native threads are running. It's really an impressive GC. Even if it is introduced once, it's hard to tell whether it will be used perpetually, but anyway it will proceed for the direction to which we can expect somewhat improvement on speed.

## Parser

Regarding the specification, it's very likely that the nested method calls without parentheses will be forbidden. As we've seen, `command_call` has a great influence on all over the grammar. If this is simplified, both the parser and the scanner will also be simplified a lot. However, the ability to omit parentheses itself will never be disabled.

And regarding its implementation, whether we continue to use yacc is still under discussion. If we won't use, it would mean hand-writing, but is it possible to implement such complex thing by hand? Such anxiety might left. Whichever way we choose, the path must be thorny.

## Evaluator

The evaluator will be completely recreated. Its aims are mainly to improve speed and to simplify the implementation. There are two main viewpoints:

- remove recursive calls like `rb_eval()`
- switch to a bytecode interpreter

First, removing recursive calls of `rb_eval()`. The way to remove is, maybe the most intuitive explanation is that it's like the “tail recursive → goto conversion”. Inside a single `rb_eval()`, circling around by using `goto`. That decreases the number of function calls and removes the necessity of `setjmp()` that is used for `return` or `break`. However, when a function defined in C is called, calling a function is inevitable, and at that point `setjmp()` will still be required.

Bytecode is, in short, something like a program written in machine language. It became famous because of the virtual machine of Smalltalk90, it is called bytecode because each instruction is one-byte. For those who are usually working at more abstract level, byte would seem so natural basis in size to deal with, but in many cases each instruction consists of bits in machine languages. For example, in Alpha, among a 32-bit instruction code, the beginning 6-bit represents the instruction type.

The advantage of bytecode interpreters is mainly for speed. There are two reasons: Firstly, unlike syntax trees, there's no need to

traverse pointers. Secondly, it's easy to do peephole optimization.

And in the case when bytecode is saved and read in later, because there's no need to parse, we can naturally expect better performance. However, parsing is a procedure which is done only once at the beginning of a program and even currently it does not take so much time. Therefore, its influence will not be so much.

If you'd like to know about how the bytecode evaluator could be, `regex.c` is worth to look at. For another example, Python is a bytecode interpreter.

## ■ Thread

Regarding thread, the thing is native thread support. The environment around thread has been significantly improved, comparing with the situation in 1994, the year of Ruby's birth. So it might be judged that we can get along with native thread now.

Using native thread means being preemptive also at C level, thus the interpreter itself must be multi-thread safe, but it seems this point is going to be solved by using a global lock for the time being.

Additionally, that somewhat arcane "continuation", it seems likely to be removed. ruby's continuation highly depends on the implementation of thread, so naturally it will disappear if thread is switched to native thread. The existence of that feature is because "it can be implemented" and it is rarely actually used. Therefore there might be no problem.

In addition, I'd like to mention a few things about class libraries. This is about multi-lingualization (M17N for short). What it means exactly in the context of programming is being able to deal with multiple character encodings.

ruby with Multi-lingualization support has already implemented and you can obtain it from the `ruby_m17n` branch of the CVS repository. It is not absorbed yet because it is judged that its specification is immature. If good interfaces are designed, it will be absorbed at some point in the middle of 1.9.

## IO

The `IO` class in current Ruby is a simple wrapper of `stdio`, but in this approach,

- there are too many but slight differences between various platforms.
- we'd like to have finer control on buffers.

these two points cause complaints. Therefore, it seems Rite will have its own `stdio`.

So far, we've always acted as observers who look at ruby from outside. But, of course, ruby is not a product which displayed in in a showcase. It means we can influence it if we take an action for it. In the last section of this book, I'll introduce the suggestions and activities for ruby from community, as a farewell gift for Ruby Hackers both at present and in the future.

## ■ Generational GC

First, as also mentioned in Chapter 5, the generational GC made by Mr. Kiyama Masato. As described before, with the current patch,

- it is less fast than expected.
- it needs to be updated to fit the edge ruby

these points are problems, but here I'd like to highly value it because, more than anything else, it was the first large non-official patch.

## ■ Oniguruma

The regular expression engine used by current Ruby is a remodeled version of GNU regex. That GNU regex was in the first place written for Emacs. And then it was remodeled so that it can support multi-byte characters. And then Mr. Matsumoto remodeled so that it is compatible with Perl. As we can easily imagine from this history, its construction is really intricate and spooky. Furthermore, due to the LPGL license of this GNU regex,

the license of `ruby` is very complicated, so replacing this engine has been an issue from a long time ago.

What suddenly emerged here is the regular expression engine “Oniguruma” by Mr. K. Kosako. I heard this is written really well, it is likely being absorbed as soon as possible.

You can obtain Oniguruma from the `ruby`’s CVS repository in the following way.

```
% cvs -d :pserver:anonymous@cvs.ruby-lang.org:/src co oniguruma
```

## ripper

Next, ripper is my product. It is an extension library made by remodeling `parse.y`. It is not a change applied to the `ruby`’s main body, but I introduced it here as one possible direction to make the parser a component.

It is implemented with kind of streaming interface and it can pick up things such as token scan or parser’s reduction as events. It is put in the attached CD-ROM \footnote{ripper : archives/ripper-0.0.5.tar.gz} of the attached CD-ROM}, so I’d like you to give it a try. Note that the supported grammar is a little different from the current one because this version is based on `ruby` 1.7 almost half-year ago.

I created this just because “I happened to come up with this idea”, if this is accounted, I think it is constructed well. It took only three

days or so to implement, really just a piece of cake.

## A parser alternative

This product has not yet appeared in a clear form, there's a person who write a Ruby parser in C++ which can be used totally independent of ruby. ([ruby-talk:50497]).

## JRuby

More aggressively, there's an attempt to rewrite entire the interpretor. For example, a Ruby written in Java, Ruby\footnote{JRuby <http://jruby.sourceforge.net>}, has appeared. It seems it is being implemented by a large group of people, Mr. Jan Arne Petersen and many others.

I tried it a little and as my reviews,

- the parser is written really well. It does precisely handle even finer behaviors such as spaces or here document.
- `instance_eval` seems not in effect (probably it couldn't be helped).
- it has just a few built-in libraries yet (couldn't be helped as well).
- we can't use extension libraries with it (naturally).
- because Ruby's UNIX centric is all cut out, there's little possibility that we can run already-existing scripts without any change.
- slow

perhaps I could say at least these things. Regarding the last one “slow”, its degree is, the execution time it takes is 20 times longer than the one of the original ruby. Going this far is too slow. It is not expected running fast because that Ruby VM runs on Java VM. Waiting for the machine to become 20 times faster seems only way.

However, the overall impression I got was, it's way better than I imagined.

## ■ **NETRuby**

If it can run with Java, it should also with C#. Therefore, a Ruby written in C# appeared, “NETRuby\footnote{NETRuby <http://sourceforge.jp/projects/netruby/> }”. The author is Mr. arton.

Because I don't have any .NET environment at hand, I checked only the source code, but according to the author,

- more than anything, it's slow
- it has a few class libraries
- the compatibility of exception handling is not good

such things are the problems. But `instance_eval` is in effect (astounding!).

## ■ **How to join ruby development**

ruby's developer is really Mr. Matsumoto as an individual,

regarding the final decision about the direction `ruby` will take, he has the definitive authority. But at the same time, `ruby` is an open source software, anyone can join the development. Joining means, you can suggest your opinions or send patches. The below is to concretely tell you how to join.

In `ruby`'s case, the mailing list is at the center of the development, so it's good to join the mailing list. The mailing lists currently at the center of the community are three: `ruby-list`, `ruby-dev`, `ruby-talk`. `ruby-list` is a mailing list for “anything relating to Ruby” in Japanese. `ruby-dev` is for the development version `ruby`, this is also in Japanese. `ruby-talk` is an English mailing list. The way to join is shown on the page “mailing lists” at Ruby's official site  
\footnote{Ruby's official site: <http://www.ruby-lang.org/ja/>}. For these mailing lists, read-only people are also welcome, so I recommend just joining first and watching discussions to grasp how it is.

Though Ruby's activity started in Japan, recently sometimes it is said “the main authority now belongs to `ruby-talk`”. But the center of the development is still `ruby-dev`. Because people who has the commit right to `ruby` (e.g. core members) are mostly Japanese, the difficulty and reluctance of using English naturally lead them to `ruby-dev`. If there will be more core members who prefer to use English, the situation could be changed, but meanwhile the core of `ruby`'s development might remain `ruby-dev`.

However, it's bad if people who cannot speak Japanese cannot join the development, so currently the summary of `ruby-dev` is translated once a week and posted to `ruby-talk`. I also help that summarising, but only three people do it in turn now, so the situation is really harsh. The members to help summarize is always in demand. If you think you're the person who can help, I'd like you to state it at `ruby-list`.

And as the last note, only its source code is not enough for a software. It's necessary to prepare various documents and maintain web sites. And people who take care of these kind of things are always in short. There's also a mailing list for the document-related activities, but as the first step you just have to propose "I'd like to do something" to `ruby-list`. I'll answer it as much as possible, and other people would respond to it, too.

## ■ **Finale**

The long journey of this book is going to end now. As there was the limitation of the number of pages, explaining all of the parts comprehensively was impossible, however I told everything I could tell about the ruby's core. I won't add extra things any more here. If you still have things you didn't understand, I'd like you to investigate it by reading the source code by yourself as much as you want.

Translated by Vincent ISAMBART and Clifford Escobar CAOILE  
This work is licensed under a

Creative Commons Attribution-NonCommercial-ShareAlike2.5  
License