



THE INFERNO ADVENTURES

Inspecting the Inferno Operating
System

BY MARVIN JOHANNING

The Inferno Adventures
*Inspecting the Inferno Operating
System*

By Marvin Johanning

TEXT: © Copyright 2020 Marvin Johanning

COVER DESIGN: © Copyright 2020 Marvin Johanning

Copyright (C) 2020 Marvin Johanning.

© 2020 by Marvin Johanning

“The Inferno Adventures: Inspecting the Inferno Operating System” by Marvin Johanning is licensed under CC BY-NC-SA 4.0. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-sa/4.0>.

Inferno® and Limbo® are registered trademarks of Vita Nuova Holdings Limited in the USA and other countries, and in the European Union.

Inferno OS™ is a trademark of Vita Nuova Holdings Limited in the USA and other countries, and in the European Union.

Dis® is a registered trademark of Vita Nuova Holdings Limited in the United Kingdom, the USA and other countries.

PUBLISHING:

Marvin Johanning

Salzuffer Str. 66

33719 Bielefeld

info@marvinjohanning.de

PRINTING: epubli – ein Service der neopubli GmbH, Berlin

*We have persistent objects,
they're called files.*

KEN THOMPSON

Contents

Introduction	i
About me	iii
Book I	2
From the Deepest Depths of Hell	3
Resurrecting the Devil	7
Book II	16
Commencing the Descent	17
Combing through Hell's Inventory	22
Files	23
Charon	26
Edit	27

Contents

Acme	28
Tetris and Bounce	34
Utilities	35
Book III	38
Adrift in Limbo	39
Deliverance from Limbo	44
Mathematical Inquiries	53
Say, what is thy name?	60
In Perpetuum	65
Appendix	71
Addendum	72
Further Reading	73

Introduction

In the following book you will be presented with some information regarding the Inferno operating system created by Bell Labs in the mid 1990s. Its name and somewhat mysterious aura led me to write it in a style akin to the one that can be seen in older books, such as those from the early 20th or late 19th centuries. This applies to both the chapter headings, as well as the text itself.

Furthermore, a rudimentary study of the Limbo programming language is also included, wherein I attempt to explain its basics in a manner which I hope will be easy to understand even if you do not have a lot of programming experience.

I believe that this may also make reading it somewhat more enjoyable than the otherwise rather dryly written literature available on this subject and perhaps help those who are interested in this operating system — and that may possibly not know much about Linux itself either — can have an interestingly written introduction.

It is however to be noted that this is not meant to be a complete introduction to the operating system and its

underlying structure; it is merely intended to be a quick overview of the operating system, how to install it and what you can do with it.

Additionally, should you want further information regarding this operating system, I do promote the reading of the *Addendum*, wherein you will find supplementary information I found unfit for the main portion of this work or which I became aware of only after having finished the main work. In addition to that, the *Further Reading* section will provide you with a list of texts which can be used to delve deeper into this interesting topic.

I highly encourage to to criticise my work — this is, obviously, without having to result to petty insults and the like — and send me an email with your feedback. It is not unlikely that I have made a mistake — or perhaps even several — over the course of this document and I always like to be corrected if there are factual, grammatical or spelling errors.

This work is licensed under a Creative Commons license, which means you are allowed to share it with others free of charge and I do encourage you to do so; you should never have to pay for this document in a digital format and I would like you to report any such occurrences to me, since commercial distribution is strictly prohibited by the license and those who make a profit from it are in violation of it. Physical copies can be obtained solely from me, at least commercially.

About me

My name is Marvin Johanning, I'm twenty-one years old and currently reside in a city many deem to not exist — which, obviously, is untrue for I *do* live here and surely I am real. I like writing things, even though, perhaps, I am not great at it; yet I enjoy doing so and only through practice can you improve, which is why I write as much as possible as frequently as I can.

I have recently switched from writing in LibreOffice to L^AT_EX, as LibreOffice has proven itself to be headache-inducing when working with large amounts of text which you wish to reformat at a later date.

I tend to write about things relating to languages — be it real or programming languages —, computers and, though rarely, politics. All of these can be found on my website. My biggest writing project to-date is *The Intricacies of Ancient Egyptian Hieroglyphics* (ISBN: 978-3-752952-49-0), which incidentally was what lead me to use L^AT_EX, information regarding which can, too, be found on my website.

Book I

From the Deepest Depths of Hell

“Long is the way and
hard, that out of Hell
leads up to light.”

John Milton
Paradise Lost, Book II

I must admit, it was largely the rather intriguing name of this operating system that made me want to know more about it and do more research on it. I had come across it on one of my quite common internet *trips* wherein I simply hop from one hyperlink to the next, trying to discover something interesting — or, perhaps more accurately, something obscure. Frequently these *trips* end unsatisfactorily; yet this time, luck seems to have been on my side, for I was finally able to uncover something interesting — why, even mysterious. I am of course speaking of the *Inferno operating system* created by the renowned Bell Labs in 1996.

They had previously worked on *Plan 9 from Bell Labs* — why they felt the need to add the “from Bell Labs” portion is, unfortunately, beyond me — which, just as Inferno, has largely fallen into obscurity. It is now just to be found in the dark and long forgotten and abandoned corners of the web that seem to be frozen in time, containing information and discussions from days long past; that are held aloft simply by someone continuing to pay for server bills. It’s a pity, really, how large amounts of information can simply vanish from the world by someone — perhaps quite literally — pulling the plug.

Plan 9 — as I will be calling it henceforth, ignoring the *from Bell Labs* portion of its name, for I trust you will know that I am indeed referring to the operating system and not the movie — was actually what I had found prior to Inferno.

I am unsure of the precise circumstances that let me to its discovery, but I distinctly remember discovering a page filled with interesting documents on a wide range of topics which can be reached under the following address: <http://doc.cat-v.org/>. This fascinating archive of old documents contains a myriad of interesting titbits on both Plan 9 and Inferno — and other topics as well.

I thus decided to start reading some of the documents available on Plan 9 and was quickly intrigued — for seeing an operating system whose creators are the same ones that created the beauty that is UNIX was truly remarkable; why had I not heard of it previously?

Yet to my utter dismay, it was quickly revealed that it seems to have been long forgotten and abandoned — but a

mere relic of the past — and that one can count its user base on two hands; and while there appears to remain a small number of people that continue using it, its ultimate failure was being unable to prove much better than what it strove to replace — UNIX itself.

For they realised their old operating system had become *too* old, even by the standards of the mid-1980s. They wished to create a system — from scratch — that would encompass these new-fangled ideas, such as graphical user interfaces; a system that did not have these features haphazardly stitched on — as a surgeon would try to fix a severely wounded fellow — but one that was built with them in mind; a system that would abide by UNIX’ philosophy of everything being a file, but one that was more adapt to modern — at least, for the time — standards.

But alas, as the passage of time has revealed, their efforts have been in vain; for ask any fellow, even those who are learned and know of computers, and chances are high he will not know of their failed efforts to create a superior UNIX.

Yet this is not a tale of Plan 9, it is one of its successor Inferno. Having learnt from their former mistakes with Plan 9, they strove to create a superior Plan 9; one that would be used in network environments; one that was portable and lightweight and could easily run on a myriad of different types of machine. Yet tragically, even Inferno seems to have gone under — perhaps even into hell itself.

But fret not! While abandoned by Bell Labs long ago — what vile traitors they are! —, it has since been acquired by a British company, who have released the latest version in 2015,

known as VITA NUOVA; an Italian word, whose meaning is *new life*. They, too, appear to have obtained the inspiration for their name from one of Dante's most magnificent works and seem to be those whom exclusive rights to the Inferno operating system have been granted — and, most surprisingly, they appear to aid in the development of Bell Lab's former OS, Plan 9, and are providing their services regarding the printing of Plan 9 manuals.

I believe, though, that the time has come to unveil this long forgotten gem.

Resurrecting the Devil

“Soon you will be where
your own eyes will see the
source and cause and give
you their own answer to
the mystery.”

Dante Alighieri
The Divine Comedy,
Inferno

The first question that arises is whence one can obtain a copy of this operating system and it seems the official source code is currently available on GitBucket; but one may also download it from the current maintainers’ website: <http://www.vitanuova.com/index.html>. This appearance of this website, too, reminds one of those long forgotten sites filled with flashing GIFs and strangely formatted text from the late 1990s or early 2000s. Yet it appears that it has yet to be abandoned — why, there seem to exist discussion forums dedicated to Inferno still! What a site to behold they are;

and though posted on only occasionally, it is a pleasure to see discussions being conducted about Inferno — there may be hope still! But let us continue with our journey down into the depths of hell.

Obtaining the latest version of this operating system is a task swiftly completely; indeed, it is a mere 70 megabyte large archive — 'tis truly devilish sorcery worthy of the name “Inferno”.

Yet already we stumble across something strange; something unusual; something you shan't see with an operating system of current times — why are there different downloads for different operating systems. Surely, you cannot imply it is to be run on top of another operating system? Why yes, you can.

Indeed, something I had neglected to mention previously was the fact Inferno is no ordinary operating system — no, it is meant to be installed on top of the one already running on one's machine; and whilst it is indeed possible to install it onto bare hardware, only a madman would do such a thing. Really, its main intent was to be a complementary system to Plan 9 and one was meant to install it on top of Plan 9 to have a more complete experience.

Thus choose wisely; do not download the wrong archive, for it may not work. As I am running Linux, I chose their archive meant to be used with Unix-like systems, of which Linux is one. It is, as previously mentioned, an archive and must thus be extracted, a task easily accomplished with Linux' helpful `tar` command; one must simply type `tar xf` followed by the name of the archive and it is extracted into its own

directory.

Upon entering the directory so thoughtfully created by the `tar` command, one is greeted by files upon files and folders upon folders. Indeed, it may be slightly disorienting — even off-putting — to those uninitiated to Unix-like systems; but, as one is swift to notice, there are two helpful documents to guide us in our installation journey: a `README.md` and an `INSTALL` document. Therein you will find a lot of information that will be imperative for installing Inferno; thus, let us first take a look at the `README`.

Inside the `README` one finds general information regarding the operating system, detailing its creators and purpose. Additionally it highlights what makes it so interesting — yet also similar to Unix: —

The use of a high-level language and virtual machine is sensible but mundane. The interesting thing is the system's representation of services and resources. They are represented in a file-like name hierarchy. Programs access them using only the file operations `open`, `read/write`, and `close`. The 'files' may of course represent stored data, but may also be devices, network and protocol interfaces, dynamic data sources, and services.

It is truly remarkable; a system whereon everything is a file — something that can readily be read and edited — is revolutionary and yet it is simple — why, even mundane! A mere child could think of it. It is, beyond any doubt, an elegant and ingenious idea.

But withal, this helped but little in embarking onto our journey into Inferno; and whilst it was indeed interesting to be given some information regarding its background, we are none the wiser. Thus, let us read the `INSTALL` file, wherein we shall — hopefully — find our answers; and indeed we do! For therein we are presented with actual instructions, but as they may appear cryptic to those who have never — or perhaps only rarely — used a Unix-based operating system, I shall try to elaborate on their instructions; for they merely appear cryptic — or, perchance, even challenging — and are, in actuality, quaint and mundane.

It is, however, to be noted that my instructions henceforth will work only on Linux x86; should you wish to be presented with instructions regarding other systems — systems whose name should not be spoken, for they are vicious and vile, forged somewhere far below hell itself; in a place Beëlzebub himself does not dare to enter — you may cease your reading here, for you may not find the information you desire. The following chapter is akin to a recipe from a cook book; hence follow the instructions with care, or your “meal” may turn out wretched, ugly and inedible.

One must first define the *root* of the installation; this may sound somewhat strange, yet in actuality one is simply required to aid Inferno in knowing where on your system it is located — it cannot know on its own and assumes it is located in `/usr/local/inferno`. Hence, should you not wish Inferno to be installed there, you must alter the `mkconfig` file; therein you can modify the `ROOT` variable to point to your Inferno directory. Yet withal, I would recommend simply

using the directory `Inferno` wishes to be inside of by creating it using `sudo mkdir /usr/local/inferno` and thereafter copying the contents from the old directory to the new one by typing `sudo cp -r * /usr/local/inferno`; I have found this to be much more simple. You must then move into this directory by typing `cd /usr/local/inferno`.

But be aware! Apart from the `ROOT` variable, there exist two others, namely `SYSHOST` and `OBJTYPE`; it is imperative to modify these so that they match your system, for otherwise your installation will fail — indeed, this is akin to letting a scoundrel wreak havoc on your system. Thus — this is, unless you are using one of the aforementioned systems whose names should never be mentioned, for they cause terror and fear far greater than that caused by the devil himself —, pay heed to the values of these two variables, for they ought to be set to `Linux` and `386` respectively. The latter should be modified to equal `arm` should your Linux installation be ARM-based, such as those running on the Raspberry Pi.

Having completed this task, you will thence be able to execute `makemk.sh`, a shell script that will spawn the `mk` executable which shall thenceforth be used for building `Inferno` itself; a infernal tool for certain, yet one that shall prove invaluable.

I implore you to act with as much care as you can permit henceforth, for the following commands utilise `sudo`; an almighty tool whose purpose is to allow you to be the lord of your system; a tool that grants you the permissions to do every conceivable thing on your machine; a tool that is a juxtaposition of both Good and Evil, for it imparts you

with the power of achieving your every desire, but it may also destroy what you have created in an instant. Should your machine contain data you believe invaluable, I urge you create a copy of it elsewhere; in a place where Satan cannot touch it.

Notwithstanding, it must be used, for our machine will not permit us to make changes without it; you may attempt to run `makemk.sh` without `sudo`, but you shall be given an error, as one may not make any changes to the system without having been granted the appropriate privileges.

If you deem yourself to be prepared, execute `sudo ./make mk.sh` whereupon you may be presented with a number of warnings that can, however, be ignored; for if, after a few seconds, you are told that “mk binary built successfully!”, everything has worked and a binary has been concocted and can thence be found inside `/usr/local/inferno/Linux/386/bin`.

But alas, our machine is dim-witted, for running `mk` does not yield anything; nay, it appears the command does not exist — which is undoubtedly wrong, for its existence can be ascertained by looking at the aforementioned directory. So what is one to do? The machine needs to be told where to find it — and the binaries that shall be brought into existence soon —, which means that we will need to modify the `PATH` variable.

The contents of this variable tell our machine where binaries are located and one must thus affix the path to our previously created binary onto the `PATH` variable, a task swiftly accomplished; but first it is imperative to know what Shell is currently being used. This is achieved by typing

`echo $SHELL` into one's terminal, which, in my case, yields `/usr/bin/zsh`, for I use Zsh; on your system, a likely candidate will be `/usr/bin/bash`, since the Bourne Shell, as it is called, is one of the most prominent ones in use today.

Nevertheless, one needs to add the following line to one's `/.bashrc` or `/.zshrc` whose purpose it is to inform the machine of our newly built binary's location, so that by typing its name it will be executed: `export PATH="/usr/local/bin/inferno/Linux/386/bin:$PATH"`.

Should you have neglected to heed my previous warnings regarding the usage of a different operating system, you must change the path accordingly; though I do trust you did no such thing.

Furthermore, the adding of this line to one's `/.profile` file is also recommended, though not a strict requirement.

Upon the completion of this task, please restart your terminal, for this causes it to renew its settings and take those into account you have just added; typing `mk -help` should return some information regarding the command's usage.

Yet you may wonder — why, even be bothered —, for what has all this effort lead to? How has this file aided us in our journey towards Inferno? Fret not, for this file is but the beginning — why, it is akin to the apple that those whom God created first, Adam and Eve, had fallen for; the apple that initiated their fall from Eden; and just as they were cast from Eden, we, too, shall be cast from our current operating system and into Inferno. We, too, have sinned — albeit knowingly — to continue our descent into Inferno; `mk` is our

apple, our forbidden fruit, wherewith we shall enter hell.

Having gained access to such a powerful tool, we must return to our Inferno installation in `/usr/local/inferno` and, upon having arrived, we must commence with the creation of the system itself.

The first command that must be run is `sudo mk nuke` — just a reminder of what is to come; for if we must nuke something, this surely cannot be a system of good. Running this command will erase files which are not necessary for us; thereupon we may begin the actual installation.

Finally, the time has come for us to install Inferno; the final command that will push open the gates that shall unveil the long-forgotten mystery of Inferno — `sudo mk install`; but first, you must wait, for this may take some time, depending on how powerful of machine you own.

Book II

Commencing the Descent

*“Lasciate ogne speranza,
voi ch’intrate”*

*Dante Alighieri
The Divine Comedy*

The gates of hell now ajar; the hellish brute now released; as Bucchianeri once said, “The gates of hell are terrible to behold, are they not?”; but alas, what is to be done now? How are we to enter Inferno? If you remember, we have told our machine where to find its binaries and thus we can simply run Inferno’s emulator by typing `sudo emu` whenceforth one’s Bash or Zsh prompt disappears; instead, a simple semi-colon becomes visible — you have trot onto Inferno’s ground; you have stepped through the gates of hell, set ajar by our previous actions, and entered Beëlzebub’s realm.

Certainly, it is to be wondered how this differs from our previous operating system — or from our previous shell for

that matter —, for it seems like a mere inferior version of what appeared on our screen before. Indeed, it looks to respond to commands in much the same way as our Linux shell. Typing `ls` yields the same results as doing so on Bash; have our efforts been in vain? Are we to believe that Inferno is but a mere clone of Unix? Nay, it is but the first impressions that are the cause of this confusion; for you must keep in mind that Inferno was indeed created by the same geniuses that had once, in days long past, created Unix.

Undoubtedly proud of their previous achievements, they decided to implement — a wise decision, might I add — a large number of the commands that are to be found in Unix-based systems in Inferno. Yet you should not let yourself get fooled, for this is not Unix — 'tis merely hell hiding in plain sight; and whilst it is indeed possible to use Inferno from its shell only, it does provide a window manager; for, as I hope you remember, they longed to create an operating system that was built with a window manager in mind and not have it be, as was the case with Unix, an afterthought, carelessly plastered onto it years later.

Its window manager — simply and rather uncreatively titled `wm` — can be evoked by typing `wm/wm`, whereupon one will be greeted by a small window; indeed, its appearance is somewhat dull, consisting of a grey background, and a grey task bar — truly an atrocity to behold. Indeed, it is puzzling, for what is to be done? What is to be clicked? Whence can programs be started and wherefrom is a poor fellow supposed to receive help? Having been tossed down into this abysmal realm of dread, one is reminded of what, in his book *Paradise*

Lost, John Milton writes: —

Seest thou yon dreary plain, forlorn and wild,
The seat of desolation, void of light,
Save what the glimmering of these livid flames
Casts pale and dreadful?



Figure 1: Inferno's window manager with start menu

you in your journey, you shall henceforth be provided with photographs of the system as I see it, since I believe this will simplify your journey considerably; indeed, those poor souls whom the ability to use Inferno has not been bestowed for they lack a proper system install it onto, are what these photographs will aid most substantially, as they may therefore see the happenings of our doings without needing their own installation.

But withal, those keenly-eyed amongst us may have observed the rather small square icon at the very bottom left of

For indeed it seems what have stridden upon a system wholly encompassed by the colour of despair with no task to accomplish but to gaze upon its grey insipidness. You should not fret, though, for this is but a disguise; hiding beneath its ghastly appearance lie programs who are waiting eagerly to be used. To aid

the screen whose appearance is reminiscent of that of a tree which is situated next to a sickly grey box containing the text “Log”. The latter’s function should be quickly understood by even the more simple-minded — it opens a window containing logs, which tell us about things that have inevitably gone wrong — and the function of the tree-like one, too, should be easily understood; for what hides in this precise spot on most operating system? Why indeed, it is the start menu whose function, in most systems, encompasses the display of programs to start.

Even its appearance is rather colourless and uninspiring, for it consists merely of three sub-menus and six readily executable programs — of which there is one titled “Manual” and that shall be the one we will inspect first, as it may prove invaluable to receive some additional information.

What then appears is a window filled with text — as one would expect a manual to be — and a search bar across the top; this can be used to investigate a particular command’s or program’s function. Thus, if we wished to be presented with information regarding the window manager, simply typing `wm` into its search and pressing the *Enter* key will yield said requested information.

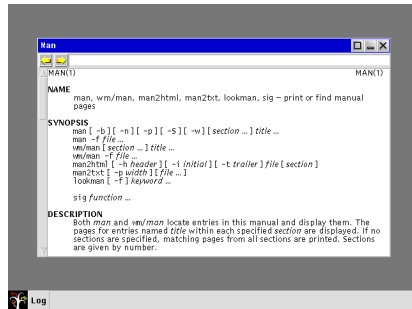


Figure 2: Manual

This program will prove itself invaluable, for it will dispense information regarding practically every single part of this operating system and we shall be referring back to it frequently.

Time has now come to open Pandora's box and examine what programs Inferno provides for us, how they are used and what can be done with them. You shall notice the rather intriguing, yet often mysterious and strange, assortment of programs this operating system has to offer which, unfortunately, is much smaller than I had at first anticipated.

Combing through Hell's Inventory

“O how unlike the place
from whence they fell!”

John Milton
Paradise Lost, Book I

Upon having finally entered and briefly explored Inferno, we shall now begin our more thorough investigations of which kinds of utensils are available to us of which there appear to be numerous. In the following section we shall explore these in brief, for most of them are meagerly equipped with functions; yet there do exist a few that are able to do much and we shall thus look at these in more detail.

I also provide you with photographs of these programs frequently.

Files Its name is, I believe, rather self-explanatory — it merely provides us with a list of files; yet, it appears to struggle doing so, for when one tries to open it, one is pestered by the log, which puts itself into the spotlight to warn us of missing items. Lamentably, its messages are of little help; they speak of the user `root` which, apparently, does not exist; and they also speak of a missing *plumber* — truly tragic that the system seems to be having problems with its plumbing.

However, I believe our newly acquired friend, the *manual* may be of help; let us thus spawn it hither once more and ask it for advice. It tells us the following: —

Plumber provides high-level message-passing between applications.

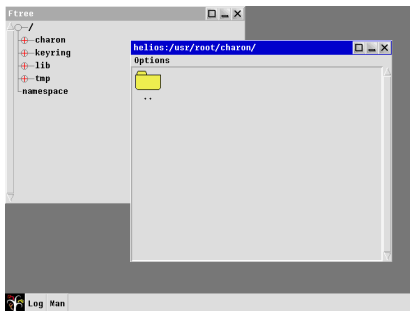


Figure 3: Inferno's *Files* program

This helps us discern what its purpose is; but alas, this helps us but scantily in ascertaining what the error is implying — indeed it appears our installation is missing this plumber altogether, but what is one to do to mitigate this?

In addition to the aforementioned information — and more, for I showed but the main definition of what plumber

does —, the manual page provides us with a file path wherein we are to find its rules for a specific user: `/usr/user/lib/plumbing`, wherein `user` must be replaced by the user who is currently using the system.

Upon entering the `/usr` directory, one immediately notices that there is but one lonely user present, namely `inferno` and the priorly mentioned `root` user is nowhere to be found.

This discovery lead me to believe that a root user must be created, a task that, due to Inferno's philosophy of all things

being files, should be easily accomplished by simply copying the `inferno` folder and renaming it `root`; hence we created a second user, the one which Files was so desperately longing after, whereupon we should have finally gained the ability to execute the File program without it deeming it necessary to throw errors at us for things we did not do.

Indeed, having done so appears to have tempered him who dwells below the surface of Inferno, for Files opens without errors; why, it finally allows us to click folders to open them at inspect their contents freely and without being disturbed.

But withal, one may now ask oneself, has what I have just done really been the correct way? Surely, one should be

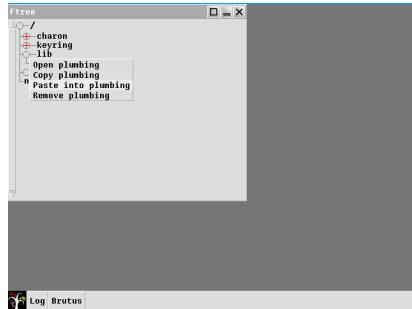


Figure 4: Inferno's *Files* program with menu

able to log in as a different user; yet, unfortunately, I have been unable to ascertain how that is possible, for the manual page of `wm` proved to be of but meager help. There seems to exist a command titled `logon` which, if used with `wm`, is supposed to provide a screen whereon a user can login; but alas, I have been unable to get it working.

Upon having fixed this program, one is now able to open, copy or remove files by clicking and pressing the left mouse button; thereupon, a menu appears where one may select an option. Should you choose the option to open a particular file, a new program will reveal itself titled **Brutus**;

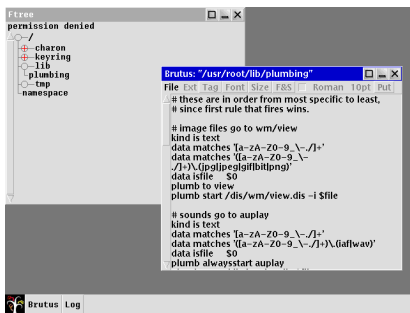


Figure 5: Brutus

yet mysteriously, one is unable to find it inside the start menu, for there it is nowhere to be found — an elusive program indeed, and one that, I must admit, I trust but little; for it appears to have gotten its name from one of those vile politicians who, back in yonder days of the Roman Empire’s glorious rule, stabbed Gaius Julius Caesar to death; why indeed, it was he whom Caesar asked, “Et tu, Brute?” in Shakespeare’s play for Caesar did not believe even Brutus would have betrayed him; and as I have learnt to not trust this name, I shall be weary of it here, too.

Yet we are lucky, for there exists a manual page regarding

Brutus which proclaims that: —

Brutus is a multi-file editor for UTF format text files.

Indeed, it appears to be but a mere text editor wherewith one is able to edit and view files; yet this raises two questions, for why can it not be found within the start menu and what is the reason for the existence of two text editors? Since, apart from vicious Brutus, there is another; one which is much more simple, something reflected in its name too, for it is called — in a truly uncreative Unix-fashion — merely *Edit*. I wonder, whither has their philosophy been tossed; their philosophy of being minimalist, for I cannot grasp the existence of two editors — whereof one being notably inferior — whose functions and purposes are nigh identical.

Charon Continuing Inferno's horrid tendency of naming its programs after beings whose names shall forever be associated with Death, its web browser is titled **Charon**; indeed he, from whom this program has received its name, is the ferryman of Hades; he whom the Greeks believe one must pay a toll for otherwise they mayn't cross the river; the river Styx; the infernal waters, which separate the land of those who live and those who are dead; and he who is unable to pay his dues is forced to wander its shores for æons before Charon shall return to take him to yonder side. Indeed, it was the custom of the Greeks and the Romans to place a coin into the mouths of the Dead, lest they arrive at the shores of the sulphurous waters, impoverished and without a coin

to part with whereupon they would be damned to wander its dismal coasts for years upon years, for Charon would not take aboard those poor souls who did not pay; nay, he would leave them ashore.

Truly, it appears that even we have been left ashore, damned to wander aimlessly, for we are unable to open websites but the one of Inferno itself. Must we, too, wait æons for him to return to us so that he may row us across the Internet river at speeds unfathomable, to allow us to reach other computers, located half-way across the world, in

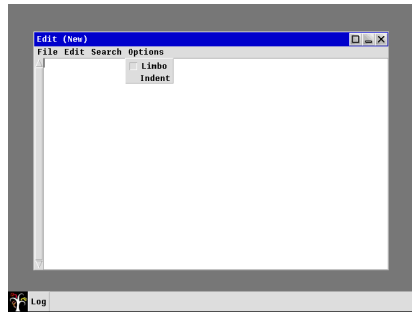


Figure 6: Edit and its options menu

but the blink of an eye? It surely appears to be the case, for I have been unable to ascertain its problem. It complains of a missing `host` file, but the lack of verbosity is truly aggravating; and lest my computer gets thrown through the window I shall continue with the next program.

Edit Having briefly touched upon the subject of editors when discussing Files and the editor wherewith it comes bundled, Brutus, which it opens when attempting to view files, we shall one more comment on the subject of editors. Since, as we have observed, one cannot spawn

Brutus from the start menu; nay, for there one may find but plain *Edit*, a truly bleak program, wholly overshadowed by its seemingly big brother Brutus. Its option menu sparsely populated and its contents cryptic — for what is *Limbo*?

I must confess that I indeed do know of Limbo and its purpose; for it is the language wherein one is supposed to write programs for Inferno, to extent its reach into the world and to expand upon its existing functions. But alas, 'tis a subject much too advanced for us at this stage, for barely do we even know Inferno's core functions; how are we, then, to write programs for it? Fear not of missing out, for we shall examine Limbo, but first we must understand Inferno's inner works; only having done thusly shall we be aptly qualified to use Limbo.

Yon program, which reminds one of those used in bygone days, truly appears to be but a meager editor and whereof not much can be said — it does, indeed, enable one to edit files, but this is all it is able to accomplish.

Acme For once we are presented with a program whose name is rather descriptive; for it derives from the Greek's *ακμῆ* which one can translate into English as *zenith* and truly, it is the most advanced of all programs we have thus far seen; and despite its appearance — which, alas, departs but little from that of Edit or Brutus —, it appears to be the system's peak. Indeed, it strikes one as being feature-rich — that is, in Inferno's terms, for the majority of its programs are rather barren — and rather sophisticated. Therefore, I thought it pertinent to engage with it more thoroughly and

that we shall do.

I was initially confused wherefore the name *Acme* had been chosen, for I believed it to be but a mere acronym; yet this appears not to be the case and I believe my judgement of its name having been chosen for its omnipotence, as I believe one can do but little with Inferno were it not for

Acme, is to be deemed accurate. Perhaps, though, I exaggerate, but I am confident in believing that most tasks would require one to do a much larger amount of work without it.

Ostensibly, one may wonder for what purpose the other programs have been constructed, since, indeed, Acme does what all other programs mentioned thus far — with the exception of Charon — can. The manual page for Acme describes it thusly: —

Acme manages windows of text that may be edited interactively or by external programs. The interactive interface uses the keyboard and mouse; external programs use a set of files served by `acme[.] [...]`

Speaking of its manual page, it is the largest I have seen

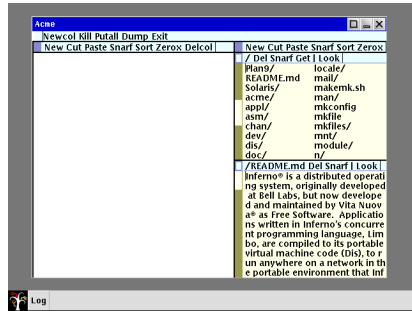


Figure 7: Acme

thus far, containing a great deal of information regarding its functions. Therefore, it shall be impossible for us to cover all its parts in-depth — indeed, we shall leave out a fair portion of it — and instead we shall be focussing on a small number of things I deem to be important or of interest.

The first aspect of this program I wish to illustrate is its mouse support; for, indeed, it may be controlled by one's mouse, yet its controls may seem rather unintuitive to those more accustomed to modern systems. One must also be prudent to differentiate between the title bar Acme has been imparted with by *wm* and its inner title bars, as the one which it has received from *wm* controls but the “outer” window of Acme itself; that is, it controls all of Acme, but not less and the inner parts of Acme cannot be manipulated or controlled using the outer title bar. The outer title bar, should my explanation sound too bizarre, is the title bar which contains the name of the program on its left side; and three buttons which enlarge, minimise or close the window respectively on the right side.

These two title bars have two rather distinct functions and controls, for the outer title bar is controlled in a fashion similar to that which is used on operating systems such as *Windows* wherein you are required to push the left mouse button to activate their functions. Yet the inner title bars depart from this control scheme quite significantly — and wherefore this has been done eludes me greatly —, for one is required to make use of all available mouse buttons; indeed, using the middle mouse button is a frequent occurrence in Inferno.

I believe it pertinent, though, to commence by studying the individual parts of the inner title bars prior to my explanation of their controls — for how may one control something one has but little knowledge of?

As visible on the provided photograph, there exist several windows within Acme, whereof each is in the possession of its own bar; yet there also exists a title may I shall henceforth call the *master title bar*, for therewith one may control all of the inner windows — indeed, one may spawn a new inner window with it. Peculiarly, there appears to exist an *Exit* field on the master title bar, rendering the *X* of the outer title bar redundant; surely, leaving it out would have not impeded the program’s functionality in any way but could have aided in making it appear less complicated.

In addition to this absolutely expendable button and the aforementioned button permitting one to spawn a new column — or inner window —, three other buttons can be observed; namely *Kill*, *Putall* and *Dump* — truly abhorrent names, I must confess, and ones that do not appear to have any discernible function, for clicking them seems to do but little. Thus, consulting the manual would no doubt be advantageous.

Upon having done so, one should have ascertained that *Dump* saves Acme’s current state — which, I believe, encompasses the currently opened inner windows / column and their position within Acme itself — to “acme dump”; it truly appears these names have been chosen by someone of but little good taste, for they are quite dismal beyond any doubt — surely, the choosing of a less vulgar and perhaps more descriptive name would not have been too difficult a task.

The function of the *Kill* button is, to me, rather equivocal, for the manual has but little information regarding its function and merely states the following: —

“Send a kill note to acme — initiated commands named as arguments”

Given this rather tenuous explanation of its function is, unfortunately, insufficient for me to comprehend it; due to that, I shall refrain from commenting on it further and proceed with the last item, namely *Putall*.

Yet, I was forced to uncover, it, too, appears to have a rather cryptic manual entry, for it describes its function thusly: —

“Write all dirty windows whose names indicate existing regular files”

It is truly remarkable how a manual page has the ability to baffle one so greatly that he is thence even more perplexed than he was prior to having read it. For, indeed, why would there be dirty windows on an operating system; something that does not physically exist? Let us therefore continue by examining the other windows’ title bars, even though they, too, appear to contain names I find much too cryptic.

The very first item thereon is a simple square, either suffused entirely by a cerulean colour or merely possessing borders of that colour, in which case the other parts of the square are entirely white; yet the reason wherefore there exist

two distinct types of rectangles escapes me, for they seem to fulfil the exact same functions, namely moving the window around. Indeed, if one wishes to change the location of one of Acme's inner windows, one is required to press and hold the left mouse button and hence move the window whither one desires.

Located to its right, a button titled *New* can be found, wherewith one may spawn additional inner windows.

Thereafter appears *Cut*, which allows one to remove the text which has been selected and, as the manual states, which the computer then places into the "snarf buffer" — horrid naming conventions will, undoubtedly, not disappear henceforth. *Snarf* appears to be the name of the system's clipboard, which also explains the function of the *Snarf* button which can be seen further to the right — it copies selected text into the clipboard.

Zerox has a rather intriguing function, for upon having clicked it, a new windows containing the text which appeared in the old window, emerges; truly a most helpful function to be in the possession of, for it may save one precious time. Yet, it may not behave in the exact manner one expects it to behave, as it not merely spawns a new window; nay, it quasi-mirrors the window whence it was spawned, leading to the existence of two windows which each contain the exact same content. Indeed, changing the content of one window reveals that the cloned window, too, has its contents altered in the same manner, whereas my initial speculation was that *Zerox* would create a window which one could alter separately — but alas, this appears to not be true.

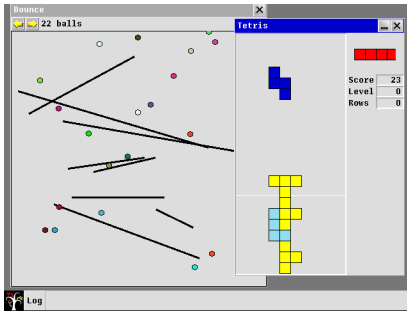


Figure 8: Inferno's games

was able to modify the contents of each window independently, I would understand and even wholly endorse it; yet I fail to comprehend this particular implementation.

There exist a small number of additional functions, yet I found those but of little importance to mention here and we shall thus proceed with other programs.

Tetris and Bounce The system is bundled with two games, one of which one may scantily call a *game*, for it is but a screen permeated by bouncing balls of varying colours and the only controls one has over said balls is the amount appearing on screen, which one may de- or increase, and the ability to draw lines, off which the balls are able to bounce.

In addition to this, an actual game — a reproduction of the acclaimed *Tetris*, a game which appears to have transcended time itself — can be played. Yet, once more, I cannot fathom wherefore this would be included, for this operating system

The reason behind the existence of such a function eludes me somewhat, for I am unfortunately incapable of understanding how one may use this in a meaningful way; having one's window precisely mirrored, to me, is a rather impractical behaviour. If, instead, one

was intended mainly for programmers who, I would like to argue, probably would not spend any time with playing a game, as their host operating system would have a much more capable environment for such an endeavour.

I am wholly convinced by the fact that, had they left these games out, nothing of interest would have been lost and I believe their reason for creating these is to boast about their new programming language's abilities.

Utilities Utilities

In addition to the aforementioned games, the system incorporates a number of utilities, whereof some appear to be of but very little use; indeed, the *Coffee* utility — a program wherewith one is able to control the speed and shakiness of half a dozen coffee making machines, spinning around fruitlessly in front of a depiction of Dante's *Inferno* — is described as, "A whimsical plaything" within the operating systems own manual; a description that I am in complete agreeance with.

Aside from the rather peculiar *coffee* program, there exist a number of other utensils, which are to be found in the start menu's *Misc* and *System* categories respectively; therewith one may perform, amongst other things, system monitoring tasks or receive various kinds of information, such as the current date and time. Hereof, however, I found only a small handful significant enough to include herein, such as the rather simplistic, yet indispensable, *task manger*. It functions in much the same manner as one would expect and is indispensable insofar that, should matters turn awry,

one may terminate a program therewith in a swiftly fashion without having to resort to the *kill* command, saving one precious time to be spent attending to matters more serious and important.



Figure 9: The *Coffee* program

spelling of *Colours* — which allows one to choose a colour of one's liking, whereupon one is shown said colour's value in RGB and Hex. Yet, I must admit, the colours appear to have been arranged in a most peculiar fashion, with no discernible logic therebehind.

Herewith we shall conclude this section, for the remaining programs — of which, I must add, there are but a small amount — are not worthy a mention herein, as they are rather ordinary; examining these in-depth would be of but little use. Instead, I believe we may henceforth begin with the study of the rather alluring Limbo programming language.

Another program I found to be of much use is the rather simplistic, yet quite convenient, *Colours* — although it appears Inferno is indecisive regarding the spelling of this word, for the title bar thereof reads *Col-ors*, whereas the start menu entry uses the much superior British English

Book III

Adrift in Limbo

“Have you ever wondered
what language they speak
in Limbo?”

Lee T. Gallup

We have trot upon foreign ground, onto the lands of a country unknown — Let us begin studying its language. Thus, I believe it is time to shortly investigate the intricacies behind the programming language this operating system has been written in and wherewith one may create one’s own programs for it — Limbo; truly, the name, once more, appears to have been derived from Dante’s works — a matter I should not be too astonished over — and bears strong resemblance to the language used in the creation of the Unix system, namely *C*.

Unfortunately, I must admit with great sorrow, I am but a mere novice regarding this intriguing programming language; indeed, I have but very little knowledge of the C programming language either. Therefore, I found it pertinent to add this disclaimer here, as the forthcoming examples may not have

been written or explained in the best of manners and I hence urge you to consider the study of a book dedicated solely to the topic of programming if you wish to learn more than but the simplest of aspects of this language.

Nevertheless, I shall herein attempt to teach you the very basics of this programming language so that, should you wish to continue your study of it, you will not dive into a more advanced book without any prior knowledge; this, I dearly hope, will greatly decrease the burden of attempting to read a more advanced book on this subject and thenceforth ease your journey in becoming more knowledgeable.

Additionally, I shall be henceforth assume that you possess a very elemental understanding of computer programming; nonetheless, I strive to write everything in a way that even a layman may grasp the majority of the subject by attempting to unravel terms and concepts that may not be known to a large portion of the population. Please be aware, however, that to follow the upcoming chapter properly, you will require a working Inferno installation; it is possible to simply read along — I shall provide everything that is necessary to follow this chapter without the need to have Inferno installed —, yet it would be beneficial to follow along and experiment with the code yourself.

Lastly, I greatly encourage you to glance at the *Further Reading* section of this book, for therein you shall be presented with my personal recommendations for furthering your wisdom regarding both the operating system and the programming language wherewith it was forged.

Firstly, it is paramount to understand a Limbo program's

underlying structure, for it may differ somewhat from other programming languages; yet, Limbo's syntax reminds me greatly of that of the C programming language — whereof, I believe, it was inspired. Nevertheless, there are a number of differences — for, otherwise, it would be but a mere clone of C and not a new language altogether — and should you know C, you may find a large amount of the upcoming code familiar, yet alien at the same time.

Programs written in Limbo are comprised of *modules* which connect with one another to perform tasks. Each module is further partitioned into two separate entities, every one thereof having their own, distinct function, namely the *module implementation* and the *module declaration*, which customarily reside in their own, distinct file; the module declaration being stored inside an `.m` file and the module implementation in a `.b` file. Typically, having an entire module be stored within just one file should be avoided for anything but the simplest of programs.

The module declaration contains the functions and constants that are to be made publicly available, whereafter other modules may access them with ease; the implementation contains the actual code to be executed and is thus, generally, the file comprised of the largest amount of lines. A module is not required to have but one mere implementation and frequently possesses several, each stored inside their own `.b` file.

Additionally, Limbo's ability can be greatly enhanced with the help of *libraries*, whereof a small number of vital ones have been included within the Inferno kernel, such as *Sys*, providing

system-related functions, such as `print`, wherewith one may print text onto the screen; and *Draw* and *Tk* which may be used to create graphical interfaces. Additional libraries can be loaded to further increase the capabilities of Limbo and add features that would otherwise be much to arduous to program oneself.

Let us look at a program written by the majority of programmers when they begin their study of a particular programming language, a *Hello World* program; its intent is to simply print the words *Hello World* on screen. To do so, create a new file titled `hello_world.b` and add the following contents: —

```
1 # hello_world.b
2
3 implement HelloWorld;
4
5 include "sys.m";
6 include "draw.m";
7 sys: Sys;
8
9 HelloWorld: module {
10   init: fn(ctxt: ref Draw->Context, args: list
11     of string);
12 };
13 init(ctxt: ref Draw->Context, args: list of
14   string) {
15   sys = load Sys Sys->PATH;
16   sys->print("Hello, World!\n");
17 }
```

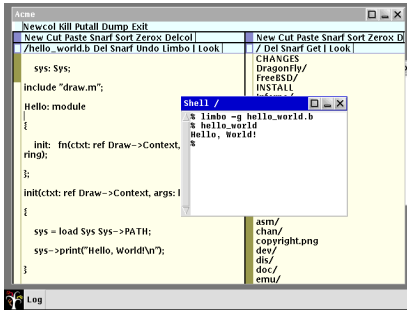


Figure 10: The Hello World program

Thereupon, a new file will be constructed simply titled `hello_world` and to execute it you must type `hello_world` into the shell window; you shall then be greeted by a “Hello, World!”.

We shall now begin examining each line of this program separately, so that we may comprehend how this intriguing language functions; for merely copying and pasting a program’s code will not allow us to fully — or, indeed, at all — grasp its underlying structure.

You may use the previously discussed Edit or Acme programs to create and save this file directly within Inferno itself, whereupon you may save it and open a new shell; therein you must find the directory whence your program has gone and type the following command `limbo -g`

Deliverance from Limbo

“From there we came
outside and saw the stars”

Dante Alighieri
The Divine Comedy,
Inferno

Indeed, we shall henceforth escape from Limbo; from the *edge of Hell* itself and ascend to higher grounds; and we shall do so by furthering our knowledge of Limbo itself, for understanding something frequently yields the knowledge that things are not as complex as they may appear.

I shall herein intermittently refer to the previously discussed code snippet and the numbers printed beside each line; I therefore highly urge you to bookmark page 42, so that you may open the code at a moment’s notice.

The code begins with a comment, a piece of text that the program will ignore and whose sole purpose it is to provide some information to those who read the source code; the comment here is merely stating that the file’s name is

`hello_world.b`. To indicate that a piece of text or code is a comment, one must use the pound (`#`) symbol, whereafter the line it is located on will be ignored by the compiler.

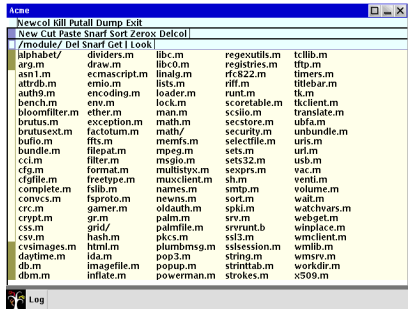


Figure 11: The included libraries

Figure 11: The included libraries

the first actual piece of code is located on line 3, whereon the current file is marked as being the implementation of the HelloWorld module, the name of which can be chosen by the programmer. You may have also observed the semi-colon at the end of this line and it serves a very crucial function; for if you forget the semi-colon, the program will not compile and instead yield the following error: `hello_world.b:3: near 'include': syntax error`. You may wonder why this occurs and the answer is rather straightforward: one must add a semi-colon behind every statement; this can be easily forgotten, especially in the beginning, and can quite easily lead to frustration, especially in longer programs — therefore, take heed, for you surely do not wish your program to throw errors at you, which then requires you to scour the file for a missing semi-colon.

Thereafter we include two of the aforementioned libraries to aid us in creating this program, namely `Sys` and `Draw`. Should you wish to view these modules' code, you can do so by opening the directory wherein your inferno location is

The first actual piece of code is located on line 3, whereon the current file is marked as being the implementation of the HelloWorld module, the name of which can be chosen by the programmer. You may have also observed the semi-colon at the end of this line and it serves a very crucial function; for if you forget the semi-colon, the program will not compile and instead yield the following error: `hello_world.b:3: near 'include': syntax error`. You may wonder why this occurs and the answer is rather straightforward: one must add a semi-colon behind every statement; this can be easily forgotten, especially in the beginning, and can quite easily lead to frustration, especially in longer programs — therefore, take heed, for you surely do not wish your program to throw errors at you, which then requires you to scour the file for a missing semi-colon.

located — which, in my case, is `/usr/local/inferno` — and thereafter open the `modules` directory within. Placed therein are the libraries one can readily load, including both `draw.m` and `sys.m`; as `draw.m` is not actually needed within our Hello World program, we must merely open `sys.m`, wherein we can find a large number of lines, each providing a different functionality. Let us glance briefly at the parts we have used in the creation of the Hello World program: —

```
1 SELF: con "$self";    # Language support for
    loading my instance
2
3 Sys: module
4 {
5     PATH: con "$Sys";
6     #...
7     print:    fn(s: string, *): int;
8     #...
9 }
```

One of the most important functions of a Hello World program is, rather obviously, the printing of “Hello, World!” onto the screen and this functionality is located within `sys.m`. The `print` function is declared therein and it takes a string as its argument — a string is a sequence of letters and symbols, such as “Hello” —, however, in addition to taking a string as its argument, it may take other types of data as well, as indicated by the star. One may also observe the declaration of the `PATH` constant (`con`) — a type of data that cannot be altered, unlike a variable —, to which the value of “`$Sys`” has been assigned; we shall be referring to this again shortly.

Line four of our Hello World program contains the defi-

inition of a variable, namely `sys`; declaring a variable — or, for that matter, functions and other declarable parts of this programming language — adheres to the following formula, wherein `DATA` must be replaced by what the variable shall store: —

```
1  variableName: VariableType DATA;
```

Thus, if you wished to store the “Hello, World!” text within a constant, you could alter the previous program as follows: —

```
1 # hello_world_var.b
2 # ...
3 init(ctxt: ref Draw->Context, args: list of
   string) {
4     text: con "Hello, World!\n";
5     sys = load Sys Sys->PATH;
6
7     sys->print(text);
8 }
```

We thus declared that `text` shall be retaining data of the `con` type with the content of `"Hello, World!\n"` which is then printed using the `print` function from the `sys.m` library.

Let us return, however, to the tenth line of our original code, whereon we find the module declaration; which, I sincerely hope you recall, is what would typically be located in its separate `.m` file — this has, however, been ignored here, for it is unnecessary to split the program, as it us but a mere seventeen lines long — indeed, splitting programs whose scale is as minor as this may actually be detrimental and induce confusion; this is, without a doubt, especially noticeable with

those people who are new to the language and, perhaps, even novices regarding computer programming in general. If, however, you longed to create a separate file for your declaration, you would be required to include it as you did the others as well, by adding a third `include` statement at the top as follows, wherein `module_file_name` must be replaced by the actual file name you gave the declaration file: —

```
1 include "module_file_name.m";
```

The declaration spans three lines in total, namely nine through eleven and is akin to what one may witness in programs written in programming languages such as *Java* or *C*, for those require the creation of a `main` method, which operates in a similar manner to the `init` function created within our Hello World program; in Java, it may be written as follows: —

```
1 class HelloWorld {
2     public static void main(String[] args) {
3         System.out.print("Hello, World!\n");
4     }
5 }
```

One is immediately able to spot the similarities, for the second line of this code snippet serves a function identical to the tenth line of our Limbo program — or rather, almost identical, for in Java, there is no need for a separate declaration as is the case in Limbo. Thus, in Limbo, we are required to both declare and thereafter execute our `init` function, which one may see from line thirteen onwards, for therein the *init* function is executed; and in there, the code that will be

run can encountered.

Should you be unfamiliar with the different types of brackets that are used — or, indeed, the structure and syntax of functions altogether —, I shall attempt to briefly explain the difference between them; let us thus consider merely lines thirteen and seventeen: —

```
1 init(ctxt: ref Draw->Context, args: list of
   string) {
2   #...
3 }
```

Everything placed in-between the round brackets is seen as an argument which can be passed to the function, whereas the code placed between curly brackets is the so-called *body* of the function; if you, for example, wished to create a function that greets people you may do so as follows: —

```
1 # greet_function.b
2 implement Functions;
3
4 include "sys.m";
5 include "draw.m";
6 sys: Sys;
7
8 Functions: module {
9   init: fn(ctxt: ref Draw->Context, args: list
   of string);
10
11 };
12
13 greet: fn(text: string): string;
14
15 greet(myname: string): string {
```

```
16   return "Hello " + myname + "\n";
17 }
18
19 init (ctxt: ref Draw->Context, args: list of
      string) {
20   sys = load Sys Sys->PATH;
21   sys->print("%s", greet("Marvin"));
22 }
```

Herein, we created a new function to which the name `greet` has been appointed; and, as one may hopefully observe, located within the round brackets, the argument one may pass to it is to be seen, namely a string which will thence be saved under the `myname` variable.

Enclosed by curly brackets, however, the aforementioned function *body* can be found, wherein the actual code is situated; in this particular instance, the function's purpose is to return `"Hello " + myname + "\n"`, wherein `myname` will be replaced by the string passed onto the function.

The function is executed on line 21, placed within the `print` statement, where it is run with the `"Marvin"` argument, whereupon the function's output — i.e. `"Hello Marvin\n"` — is printed onto the user's screen.

Let us hence return from this minor divergence and continue on our path towards understanding our Hello World program. Having briefly touched upon the subject of functions, we shall thus begin the dissection of lines 14 and 16, respectively; the statement on line 14 may come across as rather bizarre at first glance: —

```
1 sys = load Sys Sys->PATH
```

Yet, as we have learnt, things frequently but appear to be difficult, whereas, in reality, they can be explained and understood swiftly.

If you recall, on line 7 of our Hello World program, a new variable, namely `sys`, was created without any contents; the contents are what will be added here. Indeed, we may have included `sys.m` at the top of our program, but we have not yet created an instance of it — for now it but eagerly awaits to be used. Thus, to create such an instance, we are required to load `Sys` from where it is located and thereafter store this instance within a variable — which, in this case, is `sys`. To load such an instance, the syntax `load module module _path` is used.

Hence, we can utilise `sys.m`'s inbuilt `PATH` constant — as we saw on page 46 — by simply referencing it using an arrow thusly: `Sys->PATH`; this is the syntax we are required to use whenever we wish to refer to a constant or function within a different module.

Thenceforth, the `sys` variable will be in the possession of a `Sys` instance, whereupon we may use it to access `sys.m` in-built functions, which will be needed on line 16, whereon we refer to the `print` function of `Sys`. This particular function, as we ascertained previously, takes a string as its argument, which we here supply as `"Hello, World!"`. The trailing `\n` is not printed verbatim, as it is a so-called *escape character*, which, in this instance, creates a new line; had this been left out, our printed text would appear in a rather inept position, behind the `%` symbol of our shell instead of being located on its own line.

Upon having analysed the Hello World program, let us continue by learning about functions a bit more in detail; thus, in the following section, we will be creating functions capable of performing certain mathematical operations and returning their result; additionally, we shall be studying the differences between a small number of different data types that can be found in Limbo.

Mathematical Inquiries

“All truths are easy to understand once they are discovered; the point is to discover them”

Galileo Galilei

As we have previously witnessed, one may specify which type of data a particular constant or variable is able to store, such as the `string` type — used to store an aggregation of characters and even numbers — which thence is treated as mere text. Using this type of data is beneficial for text which is to be printed onto the screen or for storing names and addresses, yet it is ineffective for mathematical tasks, such as the subtraction or addition of two numbers. Let us inspect the following example: —

```
1 # string-addition.b
2 # Include statements and declaration have been
   left out, same as previous code snippets
3
```



```
4 init (ctxt: ref Draw->Context, args: list of
    string) {
5     sys = load Sys Sys->PATH;
6
7     num1: con "1";
8     num2: con "2";
9
10    sys->print(num1 + num2);
11 }
```

Herein, two constants, namely `num1` and `num2`, have been defined, each containing a number placed inside quotation marks; hence, both of these are, in actuality, strings. This, in turn, implies that Limbo does not treat them as numbers per se; nay, instead, they get treated as text, which is why the `print` statement outputs *12* instead of *3*.

Thus, it is to be remarked, the `+` operator situated within the `print` statement does not perform a mathematical function in this particular instance, as its function is altered when applied to strings; for there, the operator functions as a *concatenator* of sorts, simply joining two pieces of text with one another. If we wished to add numbers and receive an accurate mathematical result, we must use *integers* — or, indeed, other type of numbers — instead; and, if we desired, we could simply create a new function titled `add`, whose sole purpose it is to accept two integers as its arguments, add them together and return the result. Such a program could be written as follows: —

```
1 # plus-function.b
2 implement IntAddition;
3
```

```
4 include "sys.m";
5 include "draw.m";
6 sys: Sys;
7
8 IntAddition: module {
9   init: fn(ctxt: ref Draw->Context, args: list
10     of string);
11 };
12 add: fn(num1: int, num2: int): int;
13 add(num1: int, num2: int): int {
14   return num1 + num2;
15 }
16
17 init (ctxt: ref Draw->Context, args: list of
18   string) {
19   sys = load Sys Sys->PATH;
20   sys->print("%d", add(2, 4));
21 }
```

Line 12 of this piece of code contains the declaration of the `add` function, taking two integers as its arguments and returning but one integer, namely the sum of the addition. In Limbo, it is imperative to declare the type of data a function may return; this is wherefore the `: int;` part is to be found at the end of the function declaration. The previously studied `greet` function, too, contained such a statement; therein, however, it is supposed to return a string instead of an integer, for the name of a person cannot be stored within an integer.

An integer, should you be unfamiliar with this terminology, is a *whole number*, id est a number which one can depict

without the usage of a fractional component; an example of a number with a fractional component would be *2.2*. Therefore, should you supply the newly created `add` function with a number containing fractions, you shall be greeted by the following error: `'add': argument type mismatch: expected int saw '2.200000047683716' of type real`. Here I provided not an integer, but a number containing a fraction, namely *2.2*.

If you instead yearned to use numbers containing fractional components, you would be required to use *real* numbers, for these types of numbers contain both integers and fractions. We shall consider the changes to be made to our current code so that it may accept real numbers shortly; first, let us continue by examining the current code.

Lines 13 through 15 see the implementation of our function, which shall simply return the sum of both integers it was supplied with.

Lastly, line 19 contains the `print` statement we have used numerous times before; yet herein, its structure appears to deviate slightly from that which was used priorly, for here it contains two statements instead of the single used previously. As I hope you recall, `print` may take only a string as its argument and thus, supplying it with an integer directly will yield an error. For example: —

```
1 sys->print(add(2, 4));
```

This would return the following error upon compilation: `'sys->print': argument type mismatch: expected string saw 'add(2, 4)' of type real`. Therefore, if we

wish for `print` to actually print the sum of our addition, we must do thusly: —

```
1 sys->print("%d", add(2, 4));
```

Here, the first argument supplied to `print` is `"%d"` which, I assume, stands for *digit*; it is used as a reference to the second argument, namely the execution of our `add` function. Please also note that, henceforth, I shall be calling the `"%d"` strings *value reference code*. The same was required in our aforementioned `greet` program, wherein it was written as follows: —

```
1 sys->print("%s", greet("Marvin"));
```

Herein, the `"%d"` has been replaced by an `"%s"` instead, for the `greet` function does not return an integer, but a string instead.

Thus, the first argument receives the value of the second argument and, as it is located within a string, `print` is able to use it. You may append as many argument pairs as you wish, so long as you correctly specify their data type using the `%` parameters. Hence, one may combine both the `greet` and `add` functions within the `print` statement as follows: —

```
1 sys->print("%d%s", add(2, 4), greet("Marvin"));
```

This would return both the sum of our addition and a text which reads "Hello Marvin".

If, instead, we longed to create a function which allowed us to add real numbers, we would be required to change the code accordingly: —

```
1 implement RealAddition;
```

```
2
3 include "sys.m";
4 include "draw.m";
5 sys: Sys;
6
7 RealAddition: module {
8   init: fn(ctxt: ref Draw->Context, args: list
9     of string);
10 };
11 add: fn(num1: real, num2: real): real;
12 add(num1: real, num2: real): real {
13   return num1 + num2;
14 }
15
16 init (ctxt: ref Draw->Context, args: list of
17   string) {
18   sys = load Sys Sys->PATH;
19   sys->print("%f", add(2.2, 4.0));
20 }
```

Herein, we altered the data types our function is able to receive from `int` to `real`, changed the data type it will return to `real` and changed the formatting within the `print` statement to the value reference code `"%f"` instead; this, I believe, stands for *floating point number*. Furthermore, we are required to convert all integers — in this example, 4 — to a real number by adding a `.0` behind it. The reason behind this is that henceforth, our function may only accept real numbers; and even though integers are technically part of the real numbers, we must nonetheless supply our function with a number containing a fraction — even if said fraction is zero.

Withal, it would be possible to convert an integer into a real number by simply adding `real` in front of it; we could thus revise the `print` statement thusly: —

```
1 sys->print("%f", add(2.2, real 4));
```

In addition to the aforementioned data types, there exists another one titled `big`, which is, essentially, a regular integer which can hold larger numbers; its value reference code is `%bd` which can be remembered as meaning *big digit*. There are, however, even further data types to be found within this language, but we shall be discussing them only when we are required to use them.

Let us now examine a simple way of handling user input via command line arguments, wherewith we may improve our previously created `greet` program.

Say, what is thy name?

“The secret of getting things done is to act!”

Dante Alighieri

We shall now be improving upon our `greet` program, for the version we created is rather unwieldy; indeed, we are required to edit the source code and thereafter recompile the entirety of the program whenever we wish to alter the name of the person being greeted — truly, a most ineffective way of handling this matter. Instead, it would be much more effective to supply the name of the person whom we wish to greet when running the compiled program and one possible way of doing this is using *command line arguments*.

Command line arguments are commonly seen with programs one is meant to execute from the command line, for they add a convenient way of allowing user input; for example, a command line argument of the Linux `tar` command would be `car` — yielding `tar caf` — wherewith one may create a new tar archive.

Let us hence glance at one possible implementation of a `greet` program one can supply with a name using a command line argument: —

```
1 # greet.b
2 implement GreetImproved;
3
4 include "arg.m";
5 include "sys.m";
6 include "draw.m";
7 sys: Sys;
8 arg: Arg;
9
10 GreetImproved: module {
11   init: fn(ctxt: ref Draw->Context, args: list
12     of string);
13 };
14 greet: fn(name: string): string;
15
16 greet(name: string): string {
17   return "Hello, " + name + "\n";
18 }
19
20 init (ctxt: ref Draw->Context, args: list of
21   string) {
22   sys = load Sys Sys->PATH;
23   arg = load Arg Arg->PATH;
24
25   arg->init(args);
26   input := arg->arg();
27
28   sys->print("%s", greet(input));
29 }
```


As you may have noticed, we are calling upon the help of a library we had not used previously, namely `arg.m`; this library possesses the code which allows us to easily handle command line arguments. On line 8, just as we were required to do with `sys.m`, we see the creation of a new variable titled `arg` of the `Arg` type, which shall shortly be filled with an instance of the `Arg` library.

The `greet` function itself has not been altered and one can copy it verbatim from our previously created `greet` program.

Lines 22 through 25 contain all the declarations and references we are required to use in order for us to be able to properly employ the `arg.m` library, beginning with storing an `Arg` instance within the priorly made `arg` variable on line 22.

Thereafter, on line 24, we execute the `init` function within the `Arg` module, to which we supply the `args` variable, which we have defined within the `init` function of our current `greet.b` program. Let us briefly view the `Arg` module's code:

```
1 Arg : module
2 {
3   PATH: con "/dis/lib/arg.dis";
4
5   init: fn(argv: list of string);
6   setusage: fn(usage: string);
7   usage: fn();
8   opt: fn(): int;
9   arg: fn(): string;
10  earg: fn(): string;
11
12  progname: fn(): string;
```

```
13  argv: fn(): list of string;  
14  };
```

The `init` function herein is rather similar to the one we are using within our `greet` program, with the sole differences being the omission of the `ctext` variable and the renaming of `args` to `argv`.

Additionally, let us decipher the declaration of the `args` variable within the `init` function, for it creates a variable which may store `list of string`; as we have discussed antecedently, a string's function, in essence, is to store text. Thus if we wished to store a larger number of strings within a list, we can do so by utilising a `list of string` data type, which allows one to save strings in a numbered list, wherefrom we can retrieve particular entries. The entries stored within the `args` variable is the data we enter after typing our program's name.

Thus, taking our current `greet` program as an example, if we typed `greet Marvin` into the shell, the first argument will be the program itself — namely `greet` —, whereas the second argument will be *Marvin*. The `arg` function will thus return the *second* argument of our program and ignore the first argument, or those following the second one. If you wished to retrieve the first argument, the function `progrname` should be used instead.

On line 25, the declaration of the `input` variable can be found with a rather strange looking sign, namely `:=`; this is used whenever you wish to write both the declaration of a variable and its functions on one line; in essence, it is the

amalgamation of `:` — used to declare a variable — and `=` — which one uses to confer a particular value to the variable.

The value we herein assign to it is `arg->arg()`, id est, the output of the `arg` function within the `Arg` module — which, as I hope you recall, outputs the first argument provided following the name of the program itself.

Then, lastly, line 27 concludes the program by printing the output of the `greet` function, whom we provide with the user input stored within the `input` variable.

However, a large portion of names tend to consist of more than but one word and as our improved `greet` function only returns the value of merely one of the appended command line arguments, it may not work with scores of different names. Thus, if we wished to work with all affixed arguments, we are required to work with so-called loops.

In Perpetuum

“The secret of getting
things done is to act!”

Dante Alighieri

Indeed, loops, as the name may suggest, perform a certain task — that which is placed within them — for a certain duration; one generally assigns said duration to the loop as a mathematical condition, for example: so long as statement `x` returns the value `true`, execute the following code. This type of loop is titled a `while` loop.

In addition to `while` loops, another type of loop, namely the `for` loop, is also to be found. Therein you define a variable within the *head* of the function, add a condition and, lastly, provide an operation that is to be conducted upon the priorly defined variable. For example, one may construct a `for` loop in whose *head* the variable `x` is defined as an integer with a value of 0; thereafter specify that the loop is to be run as long as `x` is less than 10; and thereupon instruct the loop to increment the value of `x` by 1. This would result in the code

placed within the functions `body` to be executed ten times. Such looks thusly in Limbo: —

```

1 # for_loop.b
2
3 # ...
4 init (ctxt: ref Draw->Context, args: list of
      string) {
5   sys = load Sys Sys->PATH;
6   for (i := 0; i < 10; i++)
7     sys->print("%d, ", i);
8 }
```

This program would return the numbers 0 through 9 separated by a comma.

As heretofore discussed, the arguments appended to our program are of the type `list of string`; this allows us to iterate over each entry, store it in a regular string and print the results — printing lists on their own is, unfortunately, irrealisable. Let us thus improve our greet program even further, by using our newly acquired knowledge of loops so that a user may append more than one name: —

```

1 implement GreetEvenMoreImproved;
2
3 include "sys.m";
4 include "draw.m";
5 sys: Sys;
6
7 GreetEvenMoreImproved: module {
8   init: fn (ctxt: ref Draw->Context, args: list
9     of string);
10 }
```

```
11 init(ctxt: ref Draw->Context, args: list of
    string) {
12     sys = load Sys Sys->PATH;
13     args = tl args;
14     for (s := ""; args != nil; args = tl args)
15         s += " " + hd args;
16
17     if (s != "")
18         sys->print("Hello %s \n", s[1:]);
19
20     if (s == "")
21         sys->print("Enter your name.\n");
22 }
```

We shall commence by briefly glancing at line 13 of this program which contains the following line.

```
1 args = tl args;
```

Lists can be addressed using the `tl` — tail — and `hd` — head — commands, wherein the *head* returns the first argument as a `string`, and *tail* returns all other arguments as a `list of string`. Thus, on this line, we effectively remove the head from the `args` variable — and, as I hope you remember, since the first argument is the name of the program itself, it is not needed anyhow.

The remaining lines consist of the `for` loop itself and two `if` statements, whereof the latter should be simple enough to grasp quickly and we shall thus concentrate on lines 14 and 15 primarily: —

```
1 for (s := ""; args != nil; args = tl args)
2     s += " " + hd args;
```

The `for` loop's initial statement is the declaration of the `s` variable as an empty string; thereafter follows the condition of `args` not equalling `nil`; and lastly the command `args = tl args` is supposed to be executed, videlicet discarding the head of the `args` variable; this is to be done until `args` no longer contains any data — id est when its value has become `nil`.

The successive line then attaches a space — " " — onto the `s` variable, plucks the head from the `args` variable and, lastly, adjoins it to the `s` variable; thereupon the loop is executed again, yet this time, the `args` variable has had one of its arguments expunged.

Appendix

Addendum

A BRIEF WORD ON COMPILING LIMBO PROGRAMS

Further Reading

I applaud you greatly if you have achieved reading this far and wish to learn even more about this most fascinating operating system. While there do appear to exist quite a number of different manuals, documents, documentation, tutorials etc., the majority of those seem to have been written quite a while ago; and whilst the Inferno community is yet to die — as you hopefully remember from the text itself —, it does not look as if a large amount of scripture is published on this subject.

Nevertheless, the documents that *do* exist are of great quality and a large number thereof have been written by the creators themselves.

A great book regarding programming with Limbo is Phillip Stanley-Marbell's *INFERNO PROGRAMMING WITH LIMBO* (ISBN 0 470 84352 7). In addition, the renowned Brian W. Kernighan has written a paper on the Limbo programming language titled *DESCENT INTO LIMBO*, a copy whereof can be obtained from VITA NUOVA's website under the following link: <http://www.vitanuova.com/inferno/papers/descent.html>. An archived copy thereof can be found on archive.is under the

following URL: <https://archive.is/wip/eQJ05>.

Additionally, as briefly mentioned at the beginning of the book, VITA NUOVA print their own Plan 9 and Inferno manuals, both of which can be ordered on their website.

Another tremendous archive of documents regarding both Inferno and Plan 9 can be found on this website: <http://doc.catv.org/inferno/>