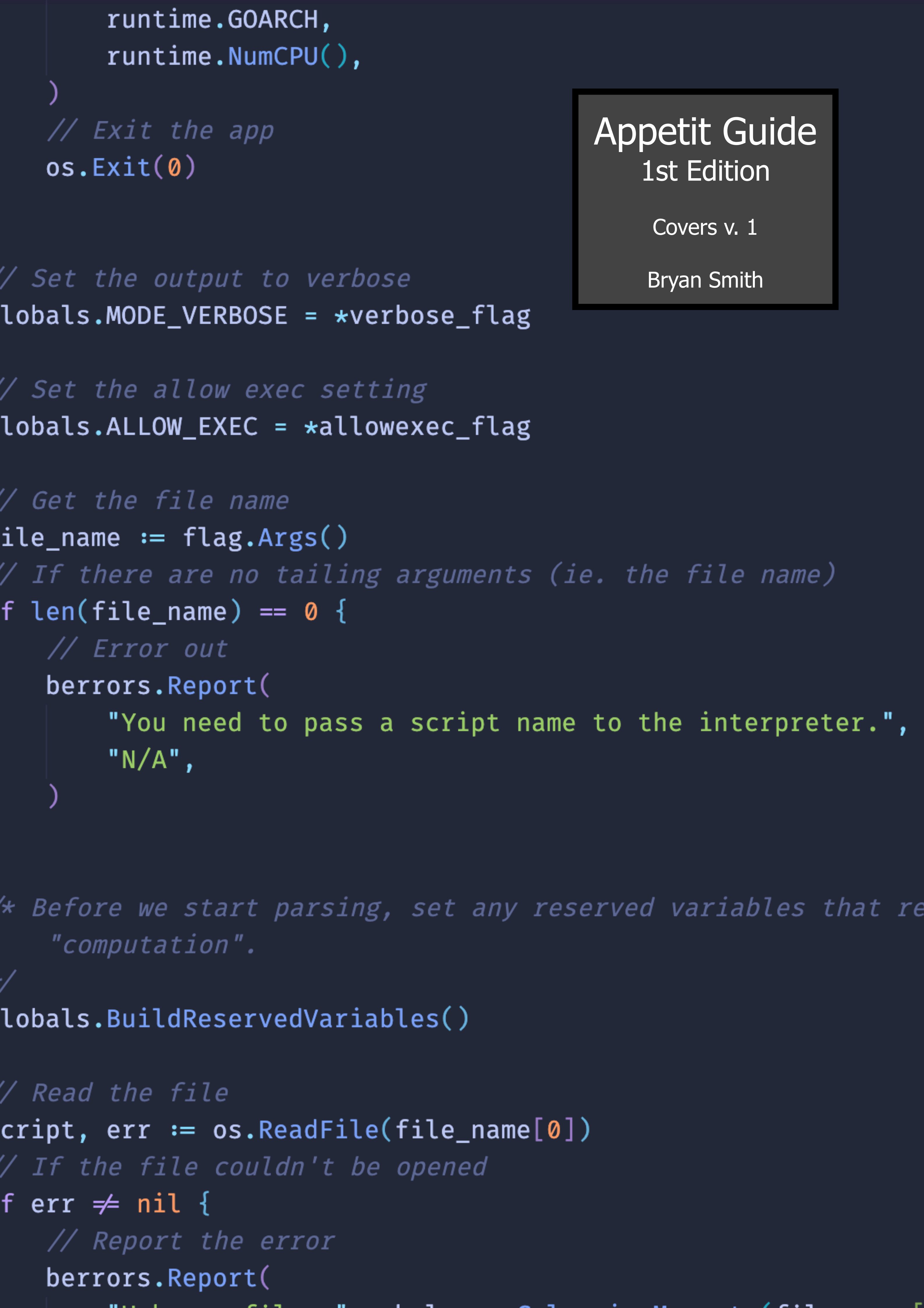
Appetit Guide

1st Edition

Covers v. 1

Bryan Smith



```
       runtime.GOARCH,
       runtime.NumCPU(),
   )
   // Exit the app
   os.Exit(0)


/ Set the output to verbose
lobals.MODE_VERBOSE = *verbose_flag

/ Set the allow exec setting
lobals.ALLOW_EXEC = *allowexec_flag

/ Get the file name
ile_name := flag.Args()
/ If there are no tailing arguments (ie. the file name)
f len(file_name) == 0 {
   // Error out
   berrors.Report(
       "You need to pass a script name to the interpreter.",
       "N/A",
   )


* Before we start parsing, set any reserved variables that re
   "computation".
/
lobals.BuildReservedVariables()

/ Read the file
cript, err := os.ReadFile(file_name[0])
/ If the file couldn't be opened
f err ≠ nil {
   // Report the error
   berrors.Report(
```

Table of Contents

© 2025 by Bryan Smith, 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/.

Introduction and Getting Started

Welcome to Appetit, the simple systems administration and management scripting language. Purpose built largely so that I could learn programming in Go and so that I could learn how to parse text (while taking into account any edge cases), the language is made available for anyone else who either wants to:

A. learn coding and/or;
B. use this to do simple systems admin work.

I'll start as with a simple question: should you use this for any serious work? Probably not. Honestly, there's a good chance that this might explode into digital bits of glory. That's not to say that this is intentional but principle A above — using this project to learn coding — means that this is very likely to have bugs. Operate on the assumption that the language is broken and do not use in any way that is mission critical. That said, the code is regularly tested and should[1] work. Version 1 is considered an "under heavy development" build.

If you're still here and interested, great!

Guiding Philosophy

The language is built around a series of simple philosophical principles that guide what the language ought to look like when you write scripts and then read them.

I. English like. The language should be one that looks like a series of instructions that could pass as workable English. This will never be perfect but it should be close. For instance, some statements (eg. ask) will never look like a coherent English statement. That said, this language is a response to scripts such as those that you might write in Bash which won't read as English because they depend on arcane syntax and/or calling tools that also have short but non-obvious purposes (eg. mv and ls are not obvious at first glance).

II. Recipe like structure. Each line is it's own instruction. The script should read like a set of synchronous instructions much like a recipe has a sequential list of rules to follow. Indeed, this is why the language is called Appetit (you're writing a "recipe" to accomplish a task).

III. Unconventional version numbering. The language will likely move quickly in version numbering to help integrate new features. In light of this, the version number of the interpreter is not particularly meaningful in much the same way that browsers like Firefox and Chrome have moved to major version numbers that move quickly. To accommodate this, the minver statement works to ensure that your script is being run on a new enough version of the interpreter.

[1] Did I emphasise that there is a realistic chance that this won't work? In case this wasn't clear, assume that it won't work (for now).

IV. Abstraction from underlying operating system. The language is designed to work, as much as possible, in a way that allows you to write a script in one place and have it work regardless of the underlying operating system. This is not particularly novel per se but worth mentioning as a key guiding principle. The language is tested (both the interpreter and scripts) to work without change on the following platforms: macOS, Linux, FreeBSD, NetBSD, OpenBSD, and Windows[2]. Currently, the language is tested on the following platforms: macOS 26, Linux (Raspberry Pi OS, Fedora and Gentoo), FreeBSD 15, NetBSD 10.1, OpenBSD 7.8, and Windows 11.

Setting Up Appetit

There are two main ways to get started with Appetit.

- Prebuilt binaries. This is the easiest option and likely what you want.
- Building from source.

Running Prebuilt Binaries

If you just want to get started and have no use for working with the source code, a variety of binaries are available that you can simple download and run. Binaries are available for the following platforms and architectures.

Platform	ARMv6	ARM 64	x86_64
Windows	No	Yes	Yes
macOS	No	Yes	Yes
Linux	Yes	Yes	Yes
FreeBSD	Yes	Yes	Yes
OpenBSD	Yes	Yes	Yes
NetBSD	Yes	Yes	Yes

The only reason that ARMv6 binaries are not built for Windows and macOS is because these platforms have never had commercially available non-ARM64 releases so there is no point. The ARMv6 builds are available for those running something like the original Raspberry Pi Zero (like me!) which is a 32-bit ARM device.

If you run on a platform and/or architecture that is not accounted for above or if you'd like native package support (see more below), building from source is for you.

[2] While all platforms are officially supported, the ordering here of the supported operating systems is indicative of the priority given to each in testing and attention.

Building Appetit from Source

If you want to build Appetit from the source code, all you need is Go. You can find more information here. Getting that installed should be as easy as using your package manager or the official installer.

With Go installed, you have everything you need. To build a binary for just your architecture and platform, simply copy and paste the following from the source code directory to create a build that will be in the src/ directory once you're done.

Build Command (non-Windows)	Build Command (Windows)
cd src/ && go build -ldflags="-s -w -X 'main.BuildDate=$(date)'" -o appetit	cd src/ && go build -ldflags="-s -w -X ‘main.BuildDate=$(Get-Date -Format `”dd/MM/yyyy HH:mm:ss`”)’” -o appetit.exe

Makefile

You will notice that there is a Makefile here. This can help to simplify the build process. If you just want to do what was done above, simply execute the following.

Build Command
make me

That will build a binary tailored for your platform and architecture and put it in dist/. You can also pass a platform name to the make command to make all supported architectures for your platform. For instance, the following will create a macOS binary for ARM64 (M series Macs), x86_64, and a Universal binary[3].

Build Command
make macos

[3] At some point in the future, it's possible that ARM64 builds for macOS will be the only supported build but there's no short-term plan for that to be the case. Given that macOS 26 is the last version to support x86_64, this is a real possibility moving forward in the coming years.

Valid operating system options here include `freebsd`, `linux`, `macos`, `netbsd`, `openbsd`, and `windows`. You can build the binary for all supported platforms and architectures using the following command:

Build Command

```
make all
```

If you just want to get up and going, you can simply use the `make install` option to have a working build ready to go (which will run `make me` first).

Build Command

```
sudo make install
```

That install command will place the single binary in /usr/local/bin/ and call the binary `appetit` for immediate use.

Make.ps1

Since make is not a common tool on Windows, a PowerShell script is available that will serve as a "Makefile" but only for Windows[4]. Simply run that and you'll have your Windows builds.

Build Command

```
./Make.ps1
```

Uninstalling

Uninstalling the interpreter is not anything special since it's just a single binary. Delete that and you're good to go. No extraneous files are created so once that's deleted, you've "uninstalled" the interpreter. If you used the `make install` command, the interpreter is in /usr/local/bin/.

If you're unsure where the binary is, you can check the version information of the interpreter which will tell you where it is installed:

[4] While PowerShell is available for non-Windows platforms, the PowerShell script is really only to accommodate Windows users.

Build Command
`appetit -version`

Using the Interpreter

Note
The following section assumes that the interpreter is in your PATH. If you're on a non-Windows platform and used the Makefile to install the interpreter, you should be good to go assuming that /usr/local/bin is in your PATH. If you're on Windows or downloaded a prebuilt binary, consult your operating system's instructions for adding the binary to your PATH.

To run a script, you can simply pass the name of the script to the interpreter.

Command
`appetit [name of script]`

Flags

Some functionality of the interpreter is accessible via a set of flags, some of which impact execution.

Flag		Version Introduced
`-allowexec`	This is a required flag if you are using the execute statement. Given that the execute statement allows for arbitrary command execution, this is here as a precaution. If this is not provided and an execute call is made, the script will error out when it gets to the first execute call.	1
`-create`	This allows you to create a simple script from the interpreter. Pass this a path to a script and it will be created for you. Example: `appetit -create=~/test.apt`	1
`-dev`	This is a flag that is helpful for people working on the interpreter. If you're trying to diagnose a problem with the code for the interpreter itself, this might be helpful as it spits out a bunch of information that might be useful in parsing errors from tokenisation through to execution.	1
`-docs`	Set up a simple web server on port 8000 that serves an HTML version of the documentation. This can be helpful if you quickly need to look up a command.	1
`-timer`	Time the execution of the script and output the results at the end of execution.	1
`-verbose`	By default, some statements execute without any output because there's no reason to do so. If you want output for everything that happens, pass this flag. This can be helpful if something is happening that you aren't expecting.	1

`-version`	Output information about the interpreter itself and check for updates.	1

Tutorials

This section walks through a series of short tutorials that introduces crucial functionality and introduces you to some basic features and conventions.

Hello World

Let’s start with something simple here: the conventional Hello World example. This is not particularly useful itself but it does allow us to look at three statements: minver, write, and writeln. Additionally, we’ll see how comments work.

Let’s start with a simple three line example.

Code	Output

`minver 1`
`- This is a comment`
`writeln “Hello World”`
	`Hello World`

You’ll notice that there are three lines here, only one of which does anything visible here. With that in mind, let’s consider all the lines together.

1. `minver 1`. This line stipulates the minimum version of the interpreter that will be allowed to execute this script. This is not mandatory but is a nice way to ensure that the script only runs with a guaranteed minimum version interpreter underneath. For instance, let’s say that something was introduced in version 2 of the language and you make use of this; changing this line to minver 2 can force users to run your script with at least version 2. All statements in the Language Features section of this guide include information about when a statement was added to the language.
2. `- This is a comment`. This is, unsurprisingly, a comment. You can think of comments as notes that you leave for yourself or for others that are ignored as part of the execution. These are often used to explain what something is or does.
3. `writeln “Hello World”`. This line uses the writeln statement to output the text that follows to the console. The string `“Hello World”` needs to be in quotation marks.

Let’s try a slightly different example using the write statement to see how this is different. After all, you might rightly be asking why there is a write and writeln statement.

Code	Output
minver 1 – This is a comment write “Hello” write “World”	HelloWorld

You might immediately notice the difference here. Of note, the write statement does not force everything that comes after to a new line. Here, the word ‘Hello’ is written to a line and then the word ‘World’ follows on the same line. This also explains why there is no space between ‘Hello’ and ‘World’ here. To fix this, add a space after Hello (“Hello ”) or before World (“ World”):

Code	Output
minver 1 – This is a comment write “Hello” write “ World”	Hello World

Now, if you want something on a new line, this is where the writeln statement comes in. See the sample below and the output, in particular, to see the effects of using writeln instead of write.

Code	Output
minver 1 – This is a comment writeln “Hello” write “World”	Hello World

Hello User: Getting Input

The above example is rather generic and you likely don’t want to say hello to the world generically. After all, what use is that? Let’s ask the user what they want by making use of the ask statement to produce a personalised greeting.

The ask statement takes a particular form and requires you to provide a variable name to assign it to. If you’re not familiar with what a variable is, they are a named value that resides in memory which you can refer to elsewhere. What this looks like might be best demonstrated in the following example.

Code	Output
minver 1 – Ask the user for their name ask “Your name: ” to name writeln “Hello #name!”	Hello User!
Note	Assuming that the user inputs User as their name, the output above is accurate.

The above requires some explanation, particularly with what’s happening with line 3 and 4. On line 3, the ask statement is asking for a name and saving it to the name variable. Once the third line is done, a space in memory is reserved called name that holds the user’s answer to our question. This can be accessed somewhere else by prepending the variable name with the # sign. You can see this in line 4 (our writeln statement) where name is provided. Anything that is prepended with the # sign that is a valid variable will be replaced with the value. If it’s not a variable value, the actual text will be written (ie. if name was not a variable, the text `Hello #name!` would be written to the screen).

Simple Addition Calculator

Code	Output
minver 1 – Get our first number ask “First Number: ” to first_num – Get our second number ask “Second Number: ” to second_num – Set a variable to hold the sum by asking Appetit to do some arithmetic set sum = “#first_num+#second_num” – Write out the answer writeln “The answer is #sum!”	First Number: [user inputs a number, say 10] Second Number: [user inputs a number, say 2] The answer is 12!

In this example, we’re introduced to three new things: two statements and variable substitution.

- The ask statement will prompt the user to enter in some text;
- The set statement will set a variable and store the value for retrieval later;
- The set statement and the writeln statement will use the value of variables for later usage.

Let’s take each of these in turn. First up, the ask statement will prompt the user for the first number and a second number. These will be held in two variables respectively:

first_num and second_num. Here, we create two variables that store our two numbers that we're going to add.

Next up, we're creating a sum variable that holds the product using the set statement. You'll notice two things here. First, we're including the variables in the assignment and prepending them with a # symbol. The # symbol tells the interpreter that you want to treat the word that it prepends as the variable, replacing it with the value of the variable. You'll also notice that we construct a mathematical expression. If the interpreter can calculate a variable value, it will.

Our final line prints out the answer. That's all there is to it! You see how this might also work, you can try replacing the + sign with / to make this calculator do division.

Download and Backup

Our final example involves two other statements — download and copyfile. Our task here involves downloading a daily version of NetBSD's aarch64 ISO and storing it with a date stamp. In addition to our two new statements, we're going to tap into the reserved variables and add a date stamp to the ISO[5]. As with our previous examples, let's look at some code first:

Code	Output
minver 1 - Set a variable to hold the URL of the ISO set isourl = “https://nycdn.netbsd.org/pub/NetBSD-daily/HEAD/latest/images/NetBSD-11.99.4-evbarm-aarch64.iso” - Write out the answer download “#isourl” to “#b_home/Downloads/netbsd-#b_date_ymd.iso”	Downloading NetBSD-11.99.4-evbarm-aarch64.iso Downloaded 100.00% (301512 KB of 301512 KB) File downloaded to /Users/bryansmith/Downloads/netbsd-2025-12-22.iso

This script will do a few things:

1. First, we set the minimum version to 1.
2. Next, we create a variable called isourl that holds the URL of the NetBSD daily ISO for arm64.
3. Finally, we download it using the download statement which takes in the URL (which we pass as the isourl variable) and the location (here, a Downloads folder in the user's home directory).

[5] The URL in this sample works as of the time of writing. If this doesn't work, check that the isourl variable points to a valid ISO image file.

Language Features

This section provides an overview of the statements in Appetit.

ask	Version Introduced: 1
The ask statement gets input from the user and stores it in a variable. This is the primary way to get input from users at runtime.	
Syntax	`ask “[prompt]” to [variable name]`
Example	`ask “What is your name?” to name`
Note	The ask statement accepts any names for the variable expect for those that share a name with a statement and those that are reserved. See the set statement for more information on reserved variables.

copydirectory	Version Introduced: 1
The copydirectory statement copies a file from one location to another.	
Syntax	`copydirectory “[path]” to “[path]”`
Example	`copydirectory “#b_home/test_path” to “#b_home/Desktop”`

copyfile	Version Introduced: 1
The copyfile statement copies a file from one location to another.	
Syntax	`copyfile “[path]” to “[path]”`
Example	`copyfile “#b_home/test_path” to “#b_home/Desktop”`

deletedirectory	Version Introduced: 1
The deletedirectory statement deletes a specified path.	
Syntax	`deletedirectory “[path]”`
Example	`deletedirectory “#b_home/test_path/”`

deletefile	Version Introduced: 1
The deletefile statement deletes a specified path.	
Syntax	`deletefile “[path]”`
Example	`deletefile “#b_home/test_file.txt”`

download	Version Introduced: 1
The download statement deletes a specified file.	
Syntax	`download “[remote_file]” to “[path]”`
Example	`download “www.internet.com/file.txt” to “#b_home/Desktop”`
Note	The progress of the download is reported back to the user. This includes the file name, a percentage based progress indicator, and a confirmation of the completion. This is, by default, one of the more verbose statements.

execute	Version Introduced: 1
The execute statement allows you to execute system commands. This can be helpful to tap into system tools to do things that you can’t with Appetit.	
Syntax	`execute “[external_tool]”`
Example	`execute “ls”`
Note	The execute statement won’t work if you don’t pass the `-allowexec` flag. This is a security measure to ensure that system commands aren’t executed accidentally. This does not mean, however, that people won’t put malicious system commands in a script so always check your scripts if you’re asked to run with the `-allowexec` flag.

exit	Version Introduced: 1
The exit statement simply exits the script.	
Syntax	`exit`
Example	`exit`

makedirectory	Version Introduced: 1
The makedirectory statement creates a directory.	
Syntax	`makedirectory “[path]”`
Example	`makedirectory “#b_home/test/”`

makefile	Version Introduced: 1
The makedirectory statement creates a directory.	
Syntax	`makefile “[file]”`
Example	`makefile “#b_home/test_file.txt”`

minver	Version Introduced: 1
The minver statement sets a minimum version that the interpreter needs to be for your script to run. This is helpful if you know, for instance, that some functionality that you use was only introduced in a specific version. This is not required but it is convention to include it.	
Syntax	`minver [integer > 0]`
Example	`minver 3`
Note	The minver statement must be the first command in a script. This is to ensure that the check can be done first before trying to execute anything. If it’s not the first line, an error will occur.

movedirectory	Version Introduced: 1
The movedirectory statement moves a file from one location to another.	
Syntax	`movedirectory “[source]” to “[destination]”`
Example	`movedirectory “#b_home/Downloads/test_dir” to “#b_home/Desktop/”`

movefile	Version Introduced: 1

The movefile statement moves a file from one location to another.	
Syntax	movefile “[source]” to “[destination]”
Example	movefile “test_file.txt” to “#b_home/Desktop/“

pause	Version Introduced: 1
The pause statement pauses the execution of a script for a set number of seconds.	
Syntax	pause [integer > 0]
Example	pause 3

set	Version Introduced: 1
The set statement creates a variable.	
Syntax	set [name] = “[value]”
Example	set place = “World”

write & writeln	Version Introduced: 1
The write and writeln statements write content to the terminal/console.	
Syntax	write “[value]” writeln “[value]”
Example	write “Hello” writeln “ World”
Note	Both statements output the content to the screen. The difference here is subtle: • write puts the output on a line but allows any following output to pick up on the same line • writeln puts content on its own line, pushing future output to the next line

zipdirectory

Version Introduced: 1

The zipdirectory statement create a zip archive of a single directory and places it in a specific destination.

Syntax	`zipdirectory "[path]" to "[zipfile]"`
Example	`zipdirectory "#b_home/test/" to "#b_home/test.zip"`

zipfile

Version Introduced: 1

The zipfile statement create a zip archive of a single file and places it in a specific destination.

Syntax	`zipfile "[file]" to "[zipfile]"`
Example	`zipdirectory "#b_home/test.iso" to "#b_home/test.zip"`

Reserved Variables

There are a set of variables that you can use that are preconfigured during runtime that you can access like any other variable. These are called reserved variables. All of them start with the prefix b_ and are listed below.

Reserved Variable		Version Introduced
`b_arch`	The architecture of the current platform.	1
`b_cpu`	The number of CPUs on the current machine.	1
`b_date_dmy`	The date in dd-mm-yyyy format (the dashes are used in case this is used as part of a file name).	1
`b_date_ymd`	The date in yyyy-mm-dd format (the dashes are used in case this is used as part of a file name).	1
`b_home`	The user's home directory.	1
`b_hostname`	The hostname of the current machine.	1
`b_ipv4`	The IPv4 address of the current machine. This may be inaccurate if there are multiple IP addresses on the machine.	1
`b_os`	The operating system of the current machine.	1
`b_tempdir`	The temp directory on the current machine.	1
`b_user`	The current user.	1
`b_wd`	The working (ie. current) directory.	1

Given that these are reserved, you can't overwrite them (ie. they are read only). In addition, you are unable to create a variable using something like the set or ask statements that starts with the b_ prefix. This is to allow for the introduction of reserved variables down the line so you can think of this prefix as effectively claiming a whole set of possibilities

Appetit Scheduler

There is a companion tool called `aptsched` which is a scheduler for running scripts on a timer. Currently, the tool supports running Appetit scripts on a recurring basis according to a timer. All of this is configured in a single JSON file.

Building

Right now, `aptsched` is available in source form and binary form. Given it's simplicity, a build of the source code should involve no more than the following which will also install the app.

Build and Install Command

```
sudo make install
```

The JSON Config File: aptsched.json

All configuration for tasks is done in a file called `aptsched.json` that is in the same directory as the `aptsched` binary. This JSON file is an array of objects each of which has four keys: name, interpreter, path, and time.

- The name key is a name that you can use to label your tasks. This will be included in the standard output and the log for the scheduler.
- The interpreter key is used to set the interpreter. This is helpful if your interpreter is in an unconventional location or if you are using multiple versions.
- The path key is the full path to the script.
- The time key includes a simple string comprised of a single integer and one of h, m, or s for hours, minutes, and seconds respectively. For instance, a valid time value might be something like 1h which would run a task every hour.

Let's see what an aptsched.json file might look like.

Sample aptsched.json File

```
[
    {
        “name”: “Backup”,
        “interpreter”: “/usr/local/bin/appetit”,
        “path”: “/home/user/scripts/backup.apt”,
        “time”: “12h”
    },
    {
        ”name”: ”Download”,
        “interpreter”: “/usr/local/bin/appetit”,
        “path”: “/home/user/scripts/dl.apt/”,
        “time”: “5s”
    }
]
```

Features by Version

Version 1

Release Date: TBD

Interpreter

- Statements: ask, copydirectory, copyfile, create, deletedirectory, deletefile, download, execute, exit, makedirectory, makefile, minver, movedirectory, movefile, pause, set, write, writeln, zipdirectory, zipfile.

Scheduler

- Initial release. Supports running multiple scripts on a schedule.

Visual Studio Code Extension

- Initial version including syntax highlighting and snippet support

Licences

The following licence explanations are based on the directory structure of the source code. Every effort is made to choose the least restrictive licence which is a balancing act between covering the materials with something that protects the copyright and efforts invested in the materials while also making it as open for repurposing as reasonably possible.

The art/icons/ directory

The icon bases here are from the Tango Project which kindly released their icons into the public domain. As a result, the icons for this project are also released into the public domain.

The docs/ directory

Appetit Documentation © 2025 by Bryan Smith 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/.

The samples/ directory

Public domain.

The source code (src/, vscode/, and website/)

Copyright 2025 Bryan Smith.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.