Appetit Guide

Version 1

Bryan Smith

Pre-Release

This is a pre-release version of the handbook. Read as indicative, not definitive.

Table of Contents

Introduction

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 management work.

Before I continue, I need to ask and answer 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 as I learn how to do things. I am not a professional programmer, having opted out of formal learning of computer science in first year of university back in 2005. Moreover, having fun and learning are the primary motivators for this project, less so making something that is battle hardened and ready to use in any production context. It may be production ready some day but ultimately, this is a subordinate goal to learning and having fun. In light of all of this, operate on the assumption that the language is broken and do not use this 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.

Guiding Philosophy

Before we begin, it's worth noting a few "philosophical" principles that guide the work and choices made in Appetit.

I. English like syntax. The language is meant to look like a series of instructions that could pass as workable English. The word "workable" is doing a lot of heavy lifting here; this will never be perfect but it should be close. For instance, some statements (eg. ask) will always read like a statement that would make an English teacher cringe. The choice here to resemble "English like" 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). Ultimately,

[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).

someone who doesn't understand the language should be able to intuit, without much struggle, what is going on in any given script.

II. Recipe like structure. Each line is it's own instruction as the language is entirely statement driven. The result of this is that the script will read like a set of steps to accomplish a task much like a recipe has a sequential list of steps to follow[2]. Because of this, there are no control structures of any kind: no loops, conditionals, or any others.

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 language is not particularly meaningful in much the same way that browsers like Firefox and Chrome have moved to major version numbers that move quickly. Don't expect SemVer or CalVer here. Knowing that this may be annoying, a version checking statement - the minver statement - is provided to ensure that your script is being run on a new enough version of the interpreter (ie. you can force a minimum version). All documentation will make clear when specific functionality is introduced into the language so you can easily set a reasonable minimum version in your script.

IV. Abstraction from the underlying operating system. The language is designed to work, as much as possible, in a way that allows you to write a script on one platform 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[3]. Currently, the language is tested on the following platforms: macOS 26, Linux (Raspberry Pi OS 13 (Trixie), Fedora 43 and Gentoo), FreeBSD 15, NetBSD 10.1, OpenBSD 7.8, and Windows 11.

V. Pragmatics over efficiency. I'm happy to accept criticism of the code and the design choices in the language (keeping in mind that this is a hobby and a learning adventure first and foremost). While that is true, a guiding principle is that pragmatics and ease of implementation will often be chosen over efficiency. If an algorithm can be refactored to be more concise, I'm happy to do it. If a suggestion comes in squabbling over the use of one type over another (eg. choosing an int32 over an int64), I'm unlikely to care enough. The former (refactoring) can be fun

[2] This is why the language is called Appetit (you're writing a "recipe" to accomplish a task).

[3] 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. In particular, the *nix operating systems get priority.

and improve learning and while the latter can improve what I know, discussions like that come at the cost of fun.

VI. Easy to install. By easy to install, I'm referring to the idea that the interpreter and companion tools like aptsched should always be a single binary. Install them is meant less as a sign of an easy installer and more that managing versions should involve nothing more than managing a single file.

VII. Learning and having fun over everything else. At the end of the day, the learning that this project is designed to support means that learning trumps anything else in the choices that are made. This is not to say that functionality isn't important; without any functional, this project is aimless. That said, the goal here very much is to learn and have fun. I don't code for a living and have no intention of entering a computer science space professionally.

Statement on GenAI Usage

Given the world we live in today, it's worth making mention of how this project makes use of code generated by large language models. The TLDR: the only generated code used here is (adapted) code that is created as part of Google's "AI Overview" search result. Crucially, this code is only used if I understand what it does; as a project designed to help me learn, I carefully consider the code produced through those search results and only integrate or adapt the output if I understand it.

The point above about this being a hobby and learning project rings true and guides the use of generated code. I say this fully sympathetic to something that has been offered as a criticism of LLM generated code: that it is code effectively stolen from others and packaged for others to use without credit. In light of this, any code used that is output by Google's AI Overview that is used is code deemed general enough that it would be common knowledge for those that have expertise. I make efforts to reasonably infer how or whether the solution to questions that are answered in the AI Overview is "sufficiently generic" and not a repackaging of someone else's unique algorithm.

Conventions

As one final consideration before we look to how to install and setup Appetit, it's worth noting a few stylistic conventions used throughout the handbook.

These are commands that you can execute in your terminal of choice (eg. Bash, zsh, PowerShell). The command will also be written in the `Courier New` font.	Command
	`Sample command`
These are notes to elaborate on something that is important to keep in mind. If you run into an issue at some point, it's very possible that a note clarifies why and how to address the issue.	Note
	This is a sample note.
These are cautions that detail things to keep in mind as you may be doing something that isn't reversible. For instance, the deletedirectory and deletefile statements don't ask for confirmation so you will see that there is a caution noting this for each statement.	Caution
	This is a sample caution.

Getting Started

In this section, we'll get up and running with the Appetit interpreter. There are some instructions later in this handbook (see here) that will help you set up Visual Studio Code to start writing scripts with editor support.

Installing 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. This also gives you the option of building installers for your platform[4].

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 from the Appetit website. Binaries are available for the following platforms and architectures[5].

Platform	ARM v6	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

[4] Well, possibly. Support is available in the Makefile for macOS pkg installers and deb files for Debian based Linux systems.

[5] 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're running one of these binaries, you are running one of the official builds which are "fully supported" (ie. as supported as much as anything is in the Appetit suite of applications is).

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.

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. The language is often developed, tested, and compiled using either the most recent or closest to most recent version of the language[6].

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. You can also use the Makefile to do this as well with the `make me` command.

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`

[6] As of the writing of this guide, I use the version available via Homebrew for macOS. See here to see what version is currently available via Homebrew.

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[7].

Build Command
`make macos`

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 (with elevated privileges) to have a working build ready to go.

Build Command
`sudo make install`

That install command will compile the interpreter for your platform and architecture, name is `appetit`, and place the interpreter in /usr/local/bin/ for immediate use. Note that this is only really helpful on non-Windows platforms (see Make.ps1 below for a Windows specific "Makefile").

[7] 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 though, this is a real possibility moving forward in the coming years.

If you're interested in all of the Makefile targets, see the table below.

Makefile Target	Description
`all`	Run the `clean`, `freebsd`, `linux`, `macOS`, `netbsd`, `openbsd`, and `windows` targets.
`clean`	Clean up caches and any lingering dist/ folders.
`freebsd`	Build x86_64, arm64, and arm6 binaries for FreeBSD.
`linux`	Build x86_64, arm64, and arm6 binaries for Linux.
`macos`	Build x86_64 and arm64 binaries for macOS.
`netbsd`	Build x86_64, arm64, and arm6 binaries for NetBSD.
`openbsd`	Build x86_64, arm64, and arm6 binaries for OpenBSD.
`windows`	Build x86_64 and arm64 binaries for Windows.
`me`	Build a binary for the current platform and architecture.
`pkg_deb`	On Debian based systems, build a deb file for local install. This builds a binary for the current architecture only.
`pkg_macos`	On macOS systems, build a pkg installer for local install. This builds a binary for the current architecture only.
`install`	Build a binary for the current platform and architecture and copy it to /usr/local/bin.
`release`	Run all target and create OS specific archives for distribution.
`source`	Archive the source code.
`test`	Run Go tests.

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[8]. Run the following from the src/ directory to get going.

[8] While PowerShell is available for non-Windows platforms, the PowerShell script is really only to accommodate Windows users. Indeed, it works more like an installer and less as a Makefile

Build Command
`.\Make.ps1`

Note
If you encounter an error trying to run the `Make.ps1` PowerShell script because you are subject to restrictions, consult the documentation here.

This script will detect what architecture you're running on (ie. ARM64 or x86_64) and then build a release binary for your platform, place it in C:\appetit\ and add that directory to your PATH. In this way, the `Make.ps1` file is very much a poor substitute for the `Makefile` in that it really just serves as an installer for the Windows machine it is being run on. If you're planning to build for multiple platforms, you're better off doing so using the Makefile via something like WSL and cross-compiling.

Uninstalling

Uninstalling the interpreter is not anything special since it's just a single binary. No extraneous files are created so once the interpreter is deleted, you've "uninstalled" the interpreter. If you used the `make install` command, the interpreter is in /usr/local/bin/. If you used the `Make.ps1` file, it will be in C:\appetit\.

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:

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 and used the `Make.ps1` file, you should also be good to go. If 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	Description	Version Introduced
`-about`	Output information about the interpreter itself and check for updates.	1
`-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. Required Parameter: 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. Note that this won't execute the script itself but will, rather, parse it and provide output about both the internal representation of data and other development related information.	1
`-docs`	Set up a simple web server that serves up the handbook that you can view in a web browser. This can be helpful if you quickly need to look up a command. Required Parameter: You need to pass this a port. Example: `appetit -docs=8000`	1
`-timer`	Time the execution of the script and output the results at the end of execution.	1
`-trace`	Create a trace file for development purposes. This is really only helpful if you are doing any work on the interpreter itself.	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

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

1
2
3
	`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 push 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 the user input 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

1
2
3
4
	`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.
- On line 4, we use the writeln statement to write out the value of the variable name to the console. Anything that is prepended with the # sign and 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

Our next example picks up on our last example of asking for user input and introduces two new features: the set statement and how variables are parsed in strings. To do this, we're going to write a simple addition calculator that asks for two numbers, adds them, and outputs the result. As with before, let's start with the code.

Code	Output

```
1 minver 1
2 - Get our first number
3 ask "First Number: " to first_num
4 - Get our second number
5 ask "Second Number: " to second_num
6 - Set a variable to hold the sum by asking
  Appetit to do some arithmetic
7 set sum = "#first_num+#second_num"
8 - Write out the answer
9 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!
```

Some of this will be familiar already: the minver, ask, and writeln statements. On line 7, we have the set statement. This simply creates a variable with a particular value. In this case, we are creating a variable called sum that holds a value. Here, the value for the sum variable is a mathematical expression: #first_num+#second_num. This expression, as you can see, takes the two variable names asked for on lines 3 and 5 and creates an expression that adds the two. Here, we see something built into the language: if a variable can be "calculated," it is and the result becomes the value for the variable. If, on the other hand, the variable assignment is not deemed to be a value that can be "calculated," it isn't and the value is set as-is. In our example above, the value assigned to the sum variable is calculable so it is and it holds the value (eg. the value of first_num + second_num).

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 for instance.

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 what are called reserved variables to add a date stamp to the ISO[9]. As with our previous examples, let's look at some code first:

[9] 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.

Code	Output

1 `minver 1`
2 `- Set a variable to hold the URL of the ISO`
3 `set isourl = "https://nycdn.netbsd.org/pub/NetBSD-daily/HEAD/latest/images/NetBSD-11.99.4-evbarm-aarch64.iso"`
4 `- Write out the answer`
5 `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/user/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).

You may have noticed that there are two variable names in the download statement that we haven't declared: `b_home` and `b_date_ymd`. These are two of the reserved variables in the language. These variables are created by the interpreter at runtime for you that hold values that are both helpful and part of helping to provide some abstraction from the underlying operating system:

- the `b_home` variable holds the value of the current user's home directory;
- the `b_date_ymd` variable holds the date in year-month-date format (specifically as yyyy-mm-dd). This is helpful here to date-stamp the ISO image that we are downloading.

You cannot name a variable that starts with `b_`. This prefix for variable names is prohibited and you can see this for yourself if you trying to create a variable that is named `b_` and then something else.

Language Reference

This section provides an overview of the language itself. Every statement in the language is listed below with a description, syntax, example, and any notes or cautions (where relevant).

Comments	Version Introduced: 1
You can easily add comments to a script to leave yourself or others notes that aren't executed by the interpreter.	
Syntax	`- [Comment text]`
Example	`- This is a comment`
Note	This is a single line comment. There are no multi line comments in Appetit.

Statements

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/"`
Caution	This will not ask for confirmation and will just delete the path.

deletefile	Version Introduced: 1
The deletefile statement deletes a specified path.	
Syntax	`deletefile "[path]"`
Example	`deletefile "#b_home/test_file.txt"`
Caution	This will not ask for confirmation and will just delete the file.

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 makefile 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. This is also helpful if you begin using a statement that only makes an appearance in a version of the language greater than the first version.	
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`

run	Version Introduced: 1
The run statement lets you execute a script external to the one that you are running currently. Effectively, this either allows you to run multiple scripts from one file or break up a complicated script into seperate files.	
Syntax	`run "[value]"`
Example	`run "World"`

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, set automatically 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_time	The time in hh-mm-ss 24 hour format.	1
b_user	The current user.	1
b_wd	The working (ie. current) directory.	1
b_zone	The timezone that the script is executed in.	1
Caution	These are generated at the beginning of execution and only then. So, if you were to write out the time, start something that took a long time, and then write out the time again, the times printed before and after the long task would be the same. This is a known issue and will be resolved.	

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 for future releases.

Appetit Scheduler (aptsched)

Note
Aptsched is very rough around the edges and has not gotten the same kind of attention as the interpreter. It should work but since more established systems exist for scheduling things — eg. cron, systemd, and launchd — you might be better using one of those until this has matured a bit.

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 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.

Key	Value
`name`	This is the name of the task. This is nothing more than a label to help you track what is being executed. This has no functional purpose in that regard.
`interpreter`	This is the path to the interpreter. This does not make an attempt to find it in your PATH in case you need or want to use multiple versions of the interpreter.
`path`	This is the path to the script that you want to run.

`time`	This is a timer for how often you want the script to run. You have one of three intervals available to you. See below.		
	Hours: h Example: 3h (every 3 hours)	Minutes: m Example: 5m (every 5 minutes)	Seconds: s Example: 12s (every 12 seconds)

Example Configuration File

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

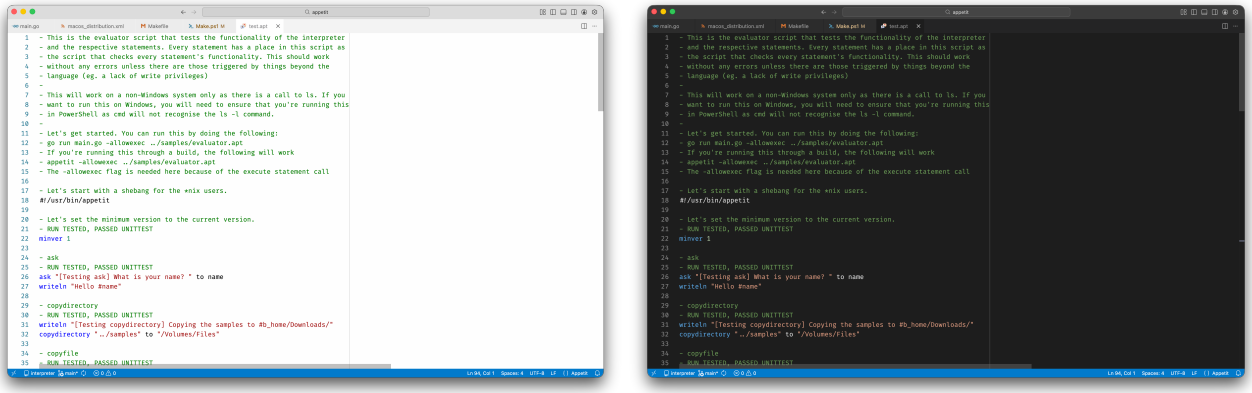
Sample aptsched.json File

```
[
    {
        "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"
    }
]
```

Here, we've got two tasks scheduled. The first is called `Backup` and will run every 12 hours. The script run run every 12 hours is located at `/home/user/scripts/backup.apt` and is run by the interpreter located at `/usr/local/bin/appetit`. The second task is called Download and will use the same interpreter but will run `/home/user/scripts/dl.apt` and do so every 5 seconds.

Visual Studio Code Extension

An extension has been developed for Visual Studio Code to ease the writing of Appetit scripts.



Extensions in light and dark mode

The extension includes the following features:

- Syntax highlighting;
- Snippet support;
- Commenting and uncommenting of lines.

Packaging and Installing

To package the extension, you will need to ensure that vsce is installed first. See here for instructions. Once that's done, you can run the following from the vscode/ directory.

Build Command
`make package`

This will output a vsix file in dist/. To install that in Visual Studio Code, open up the Command Palette and select the "Extensions: Install from VSIX…" option. Select the vsix file just created and you're done. If you downloaded the vsix file, the steps are the same (excluding, of course, the packaging step).

Features

The extension, as noted above, has three features. The syntax highlighting should be fairly obvious; once you start writing valid Appetit statements, you will notice that your code will be appropriately highlighted.

The snippets help ease the writing of statements and provide some guarantees for writing valid statements. Essentially, you write a keyword and a template for that statement will be produced for you. The following table lists the valid snippets and keywords available.

Statement	Keyword	Template Produced
ask	ask	`ask “” to “”`
comment	co	`-`
copydirectory	cd	`copydirectory “” to “”`
copyfile	cp	`copyfile “” to “”`
deletedirectory	dd	`deletedirectory “”`
deletefile	df	`deletefile “”`
download	dl	`download “” to “”`
execute	ex	`execute “”`
makedirectory	mkd	`makedirectory “”`
makefile	mkf	`makefile “”`
minver	mi	`minver`
movedirectory	mvd	`movedirectory “” to “”`
movefile	mvf	`movefile “” to “”`
pause	pa	`pause`
run	run	`run “”`
set	se	`set [space for name] = “”`
write	wr	`write “”`
writeln	wrl	`writeln “”`
zipdirectory	zd	`zipdirectory “” to “”`
zipfile	zf	`zipfile “” to “”`

Additionally, there are some commands in the Command Palette that will allow you to comment and uncomment a line:

- Appetit: Comment Line(s)
- Appetit: Uncomment Line(s)

Developer Details

In this section, we explore the inner workings of the interpreter. The audience here is those who are interested in working on the language itself and not those who are working on writing Appetit scripts.

The Parser

The central "engine" of the interpreter is the parser package. This package can be broken down into three parts.

Engine	Helpers	Statements
The engine, housed in `engine.go`, is a simple module responsible for the tokenisation of the script and execution.	The helpers module, housed in `helpers.go`, is home to a collection of functions that support the engine but are not critical.	The statements, housed in the various `stmt_*.go` modules, includes the functions that handle the functionality of various statements. These functions are called from the engine. Most of the statement modules are called `stmt_[insert statement].go` but some include functions for similar statements (eg. `zipfile` and `zipdirectory` are in the `stmt_compression.go` file.

The Engine

The engine is the key backbone of execution, home to only three functions.

The `Start()` function is the first function called that gets the ball rolling. This is called from the `main()` function entry point. It does a few things first:

- Builds the values for the reserved variables.
- Checks the `minver` statement validity by making sure that it's in the right location and that there is only one.

Once this is all done, this function calls the `Tokenise()` function and then passes the tokenised line to the `Call()` function (unless you're running with the `-dev` flag which avoids calling the `Call()` function and prints out the token information instead).

The `Tokenise()` function which takes a line of code and breaks it into a slice of tokens. A token can be thought of as a significant part of code. For instance, a statement name is

a token as would be a string as a whole (since we aren't interested in each word in the string but the string as a whole). Let's breakdown a simple `writeln` call into its two tokens:

writeln	"Hello World"

Token{
FullLineOfCode: "writeln \"Hello World\"",
LineNumber: 1,
TokenPosition: 1,
TokenValue: writeln,
TokenType: string,
NonCommentLineNumber: 1,
}
	Token{
	FullLineOfCode: "writeln \"Hello World\"",
	LineNumber: 1,
	TokenPosition: 9,
	TokenValue: "Hello World",
	TokenType: string,
	NonCommentLineNumber: 1,
	}

Here, we've got our two tokens for this line. In each, we have some key information:
- FullLineOfCode: this is the full line of code that the token comes from. This is helpful in reporting back any error messages.
- LineNumber: This is the line number that the token is on.
- TokenPosition: If you think of a line of text as having columns, this is where the token begins. You'll see that the second token — the one for `"Hello World"` — has a token position of 9. This is because the beginning of the token is nine characters in (the seven characters in `writeln`, the blank space and the opening quotation mark).
- TokenValue: the value of the token, that is, the part that is actually processed and manipulated.
- TokenType: the type internally of the token.
- NonCommentLineNumber: this holds the line number of the token if we ignore all comments.

If you're interested in seeing the tokens for a whole script, run it with the `-dev` flag.

The `Call()` function does the actual execution of statements now that the `Start()` function has prepped the script for tokenising & done some basic checks and the `Tokenise()` function has tokenised the code. With all of that done, the `Call()` function takes in a line of tokens and then executes each of the statements, throwing errors if non-statements exist. The execution of each of the main statement functions (housed in the seperate `stmt_*.go` modules) is controlled by an elaborate map — `statement_map[]` — that is the basis for making the actual calls to functions that do the work demanded of each line of code.

In short, the `Start()` function calls the `Tokenise()` function and the `Call()` function to get execution rolling.

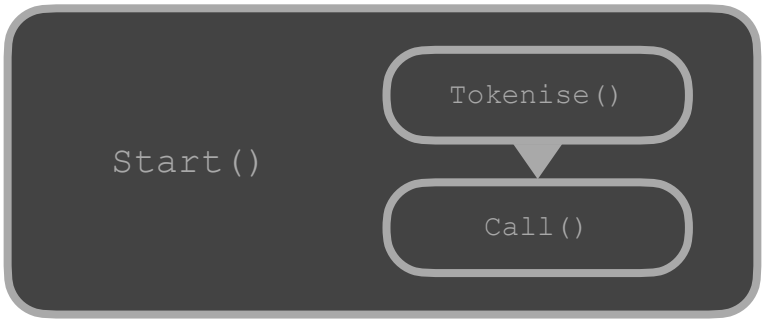


Figure 1. The function call flow in the engine.

Statement Modules

The statement modules, all called from the `Call()` function, follow a similar structure. There is a primary function that shares a name with the statement name and, for some modules, there may be support functions. Let's look at the `stmt_pause.go` module as an example. This module has only one function: `Pause()`. Like many primary statement functions, that takes only one parameter: a slice of tokens for the line of code.

The first thing that a statement function will do is use the first token to get some helpful information about the line of code:

```
full_loc := tokens[0].FullLineOfCode
loc := strconv.Itoa(tokens[0].LineNumber)
```

The first line gets the full line of code and the second gets the line number, both of which are values that are helpful in error reporting.

The second thing that a statement function does is check that there is a valid number of tokens on the line as a quick check that there is the right number of tokens. Doing this is done by way of a call to the investigator package's `ValidNumberOfTokens()` function:

```
investigator.ValidNumberOfTokens(tokens, 2)
```

For the pause statement, this checks that there are two tokens on the line[10]. This is a first check to make sure that there aren't extra tokens and that there isn't only the statement name (ie. just pause).

The next thing that we need to do is deal with the value passed to the pause statement. This is dealt with in two lines:

```
pause_as_string := tokens[2].TokenValue
pause_int, err := strconv.Atoi(pause_as_string)
```

This deals with taking in the token value which is a string and then converts it to an integer. For now, all tokens are stored as strings so the second line converts the value.

The rest of this function deals with doing the actual pausing:

```
time.Sleep(time.Duration(pause_int) * time.Second)
```

That's it! Aside from some error handling, we've unpacked the anatomy of a statement function.

[10] There's actually one token that is "hidden" on a line as one is generated for the line number. The `ValidNumberOfTokens()` function should only validate the visible number of tokens.

Changelog

Version 1 (Canberra)	Release Date: TBD

Interpreter

- Initial release.
- Statements included: ask, copydirectory, copyfile, create, deletedirectory, deletefile, download, execute, exit, makedirectory, makefile, minver, movedirectory, movefile, pause, run, 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.

If anything is not noted below, it is covered by the MIT Licence (ie. the same licence as the source code for the interpreter, aptsched, and the Visual Studio Code extension).

Icons (art/icons/ directory in the interpreter source code)

The icon bases were originally from the Tango Project which kindly released their icons into the public domain. Any versions of the code that you find with the Tango icons as the foundation are released into the public domain.

The current icons are original creations. They are licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International licence. See the details of this here: https://creativecommons.org/licenses/by-nc-sa/4.0/. In short, you are free to share and adapt the icon as long as you provide proper attribution, do not use the icon for commercial purposes, and licence any adaptations under the same CC licence.

Sample Scripts (samples/ directory)

Public domain.

Source Code (interpreter, aptsched, and vscode extension)

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.