THE PHANTASM CRASHCOURSE

A Complete Introduction to the PHANTASM Grammar
for WebAssembly Programmers

IMPORTANT: *The PHANTASM Project is in its absolute infancy. Anything published at this stage is only made available as a preview. The project uses GitHub Issues as a general forum, so feel free to start a discussion there, if you have any questions.*

The *Portable, Hardened, Asynchronous, Natively Typed, Abstract Stack Machine* (*PHANTASM*) is a novel (and opinionated) web assembler that allows you to author WebAssembly modules using a nice, modern syntax, rather than repurposing the WebAssembly Text Format.

This crashcourse summarizes the entire PHANTASM grammar in terms that anyone familiar with WAT should understand. It is pretty terse, but this should be enough to begin programming with PHANTASM (assuming you already have a working knowledge of WAT and WebAssembly).

PHANTASM Terminology

PHANTASM nomenclature is slightly different to WAT and WASM.

Instead of the the WASM terms *truncate* and *saturate* (which use the WAT mnemonics `trunc` and `sat`), PHANTASM uses *lop* and *sop* (`lop` and `sop`), respectively.

Instead of the WAT mnemonics `xchg` and `cmpxchg` (pronounced *exchange* and *compare and exchange*), PHANTASM uses `trade` and `broker`.

PHANTASM also renamed the WAT reftypes, renaming `funcref` to `pointer`, `externref` to `proxy`, and `anyref` to `reference`.

PHANTASM uses the term *identity* to refer to an index within an indexspace, either expressed directly (with a number literal) or (much more commonly) indirectly (with an identifier bound to the index). There are many places in the language where an identity is accepted or required.

PHANTASM refers to *start functions* as *initializer functions* or just *initializers*.

PHANTASM refers to *passive segments* as *banks* (*memory banks* and *table banks*). Active segments are expressed as *segments* within *primers* (*memory primers* and *table primers*), which are indented blocks associated with memory, table, memory bank and table bank definitions.

The above terms are described in detail in the corresponding sections below.

PHANTASM Files

PHANTASM source strings are currently just passed to the assembler (which was designed to run in the browser) as JavaScript Strings. How you author, store and fetch them is ultimately up to you.

PHANTASM binaries are just WebAssembly binaries. They therefore use the `wasm` extension, though the original source URL (when specified) is preserved in the binary (in the *Module Name* subsection of the *Custom Name Section*).

Note: Obviously, running inside the browser is not ideal, but frankly, it is still better than installing Node locally. Other options are being explored.

Character Set

Regular code uses a subset of ASCII, which includes the ASCII printables (including the Space Character), as well as the Newline Character. It *does not* include the Tab Character.

Only string literals and commentary can contain Unicode characters. In those contexts, there are no restrictions on the character set.

Special Characters

Like WAT, PHANTASM uses the longest-match rule, but PHANTASM uses a smaller set of special characters. In PHANTASM, the only special characters are Space, Newline and Comma.

Note: The other WAT special characters (semi-colons, parens, brackets, braces and tabs) are currently unused by the PHANTASM grammar.

The three special characters (Space, Newline and Comma) are the characters that structure the code, while the regular characters are used for spelling the regular tokens.

Significant Whitespace

PHANTASM uses significant indentation and newlines to delimit logical lines and blocks (a bit like Python without the colons).

The Newline Character terminates the logical line, unless the ellipsis-operator (see below) is used to explicitly prevent termination.

The Space Character is used to delimit individual tokens, and to indent blocks. Indentation must contain *exactly* four spaces per level, always.

The Comma Character is used to delimit sequences, and to compound repetitious constructs to minimize redundancy and improve readability in various contexts, each individually described in the relevant sections of the tutorial.

Note: The `end` pseudo-instruction is not used in PHANTASM.

Commentary

Inline-comments start with a bar (`|`), and run to the end of the line. Multiline-comments start and end on an asterisk (`*`). For example:

```
define $addmul of type $binop

    * This is multiline comment, being used as a docstring for a
    small, example function. The comment runs from the previous
    asterisk character to the following one. *

    get 0, get 1, add i32           | this is an inline comment
    get 1, mul i32                  | another inline comment
```

String Literals

Like WAT, PHANTASM uses the Quote Character (`"`) to delimit its (Unicode) string literals:

```
"Hello, World!"
```

Unlike WAT, neither slash character has any special meaning inside PHANTASM string literals. DOS-paths can be written normally:

```
"c:\windows\command"
```

Curly braces are special in a PHANTASM string literal. They are used to wrap one or more space-separated *escape-expressions*, which together form an *escape-sequence*. For example, this text ends with a space, then the grinning-face emoji:

```
"nice! {1F600}"
```

And this string ends by placing a grinning-face emoji below the text, on a newline:

```
"nice!{n 1F600}"
```

To include actual braces in a string literal, they can be doubled-up or expressed with an escape-expression. For example, the following string is valid CSS:

```
"body {{margin: 0}}"
```

Each escape-expression (within a given escape-sequence) expresses exactly one Unicode character, either using its Unicode codepoint (expressed in *uppercase* hexadecimal) or using a *lowercase* name (or a shorter, corresponding alias) defined by the assembler.

See the wiki article *String Literals* for more information, including all of the currently defined character-names, and a note on adding new ones.

Regular Tokens

Every regular token (every token that is not a string literal, comment or special character) follows the longest-match rule, and is then classified as one of ten token types: *keyword*, *prefix*, *mnemonic*, *datatype*, *component type*, *qualifier*, *declarator*, *symbol*, *identifier* or *number literal*.

Keywords

Keywords are used to *begin* directives. There are three types of directive (define-directives, import-directives and export-directives), and correspondingly, there are three keywords:

```
define    import    export
```

Prefixes

Prefixes are similar to keywords, except prefixes appear *within* directives, immediately before constructs that are (lexically) optional (constructs that can either be omitted entirely or replaced by something else, with its own prefix). There are eight prefixes:

```
as        at        from      in        of        sop       to        with
```

Each prefix has a specific association, which allows that prefix to indicate *which* optional thing it precedes (which is important when there is more than one option):

- `as`: Used extensively throughout the language to mean *as follows.* It introduces the *component-descriptors* in the directives for imports (as in `import "init" as $init of f64 → void`), exports (as in `export "result" as register $result`) and definitions (as in `define type $uniop as i32 → i32`), as well as non-default datatypes in instructions (as in `load i32 as s8`). The `as` prefix is also used to inline blocks that would need to be indented otherwise. This includes the bodies of functions and constant-expressions (as in `define function of type $binop as get 0, get 1, add i32`), as well as the *primers* used to populate memories and tables (as in `define memory with 1 as u8 0, 1, 2, 3`).
- `at`: Used to mean *at offset*. It introduces memory-offsets in instructions (as in `load i32 at 2`) and in segment-preambles (as in `@segment at #FF00_0000`).
- `from`: Used to mean *from module*. It introduces the module-name string, when it is not implicitly `"host"` (as in `import "sin" from "math" as $sin of f64 → f64`).
- `in`: Used to mean *in memory* or *in table*. It is used to identify the memory (as in `store i32 in $ram`) or table (as in `invoke i32, i32 → i32 in $opcodes`) that a datum or reference is in.
- `of`: Used to mean *of type*. It introduces function-types and signatures, accepting both type-expressions (as in `function of i32 → i64`) and type-references (as in `$sum of type $binop`).
- `sop`: Used exclusively by the `lop` instruction to *lop and sop* (meaning to *truncate and saturate*) a value (as in `lop f32 sop u32`).
- `to`: Used to mean *to memory* or *to table* (as opposed to *in memory* or *in table*) or *to type* (as in *convert to i32*). It introduces the destination memory or table that some data or references are being moved to (as in `copy memory to $ram`), as well as destination-types during conversions, promotions, demotions *et cetera* (as in `lop f32 to u32`).
- `with`: Used to mean *with initial state*. It introduces memory page limits (as in `define memory with 1 to 10`) and table slot limits (as in `define pointer table with 50 plus`), as well as the initial values of registers (as in `define variable f64 with 0.5`) when a constant expression is not required.

Qualifiers

Qualifiers are simple words that prefix or suffix other words to clarify or modify the semantics (as in `shared memory` or `start function` *et cetera*):

`shared`	`pointer`	`proxy`	`mixed`	`bank`	`atomic`	`variable`	`constant`	`start`
`global`	`local`	`left`	`right`	`equal`	`more`	`less`	`zero`	`plus`

Mnemonics

Mnemonics are simple, lowercase names that are used to begin instructions:

`nop`	`call`	`invoke`	`return`	`select`	`size`	`block`	`loop`	`branch`
`else`	`jump`	`fork`	`exit`	`copy`	`root`	`get`	`set`	`put`
`load`	`store`	`drop`	`push`	`abs`	`neg`	`nearest`	`ceiling`	`floor`
`min`	`max`	`sign`	`wrap`	`lop`	`crash`	`notify`	`wait`	`fence`
`swap`	`broker`	`is`	`not`	`nsa`	`clz`	`ctz`	`grow`	`fill`
`add`	`sub`	`mul`	`div`	`rem`	`and`	`or`	`xor`	`convert`
`promote`	`demote`	`bitcast`	`expand`	`extend`				

Datatypes

Datatypes describe the types of values, and include the pseudo-types used to work with signed integers, 8-bit and 16-bit integers and Unicode data. The following datatypes are used by PHANTASM (though different subsets are valid in all different situations):

`i8`	`s8`	`u8`	`i16`	`s16`	`u16`	`i32`	`s32`	`u32`
`i64`	`s64`	`u64`	`f32`	`f64`	`utf8`	`void`		

Note: The `void` type is only used to represent empty lists of parameters or results when expressing function types. It is not an actual datatype.

Note: The other Unicode types (`utf16` and `utf32`) still need to be implemented.

Component Types

Component-types describe the five types of *native component*:

`function`	`register`	`memory`	`table`	`type`

The `memory` and `table` component-types can be suffixed by the `bank` qualifier to refer to *memory banks* and *table banks* (described later), which are *abstract components*:

```
memory bank
table bank
```

Declarators

Declarators are like keywords, except that they begin with the At Character (`@`), and are used to introduce *preambles* (described later). There are currently two declarators:

```
@segment  @register
```

Symbols

PHANTASM has two symbols: the arrow (`->`) and the ellipsis (`...`).

The arrow-symbol is used to map arities to results when defining the types and signatures of functions. For example:

```
import "sum" as $sum of i32, i32 -> i32
```

The ellipsis-symbol is used to continue a logical line across two or more lines of a source file. The symbol prefixes each newline that should be ignored, and indents each continuation-line one level (relative to the opening line). For example:

```
import "pseudo.namespace.static.thing" from "/static/library/module.wasm" ...
    as function of i32, i32, i32, i32 -> ...
    pointer, pointer, pointer
```

Note: Trailing whitespace after the ellipsis-symbol is insignificant.

Identifiers

PHANTASM identifiers are similar to WAT identifiers, having the same dollar-prefix (`$foo`, `$bar`, `$player::score` *et cetera*), and containing any combination of regular characters.

Note: There is no support for arbitrary identifiers, as it is unclear whether they would be useful.

Semantically, identifiers are always constant. Exploiting that fact, identifier resolution is deferred until all of the indices (within each indexspace) have been assigned. This permits identifiers (and indices) to be referenced before they are bound (though the assembler still ensures they are valid). This ultimately allows you to structure your code in the most natural way.

Numbers

PHANTASM number literals are similar to WAT, except that PHANTASM uses a hash (`#`) prefix (instead of `0x`) for hexadecimals, and PHANTASM has its own syntax for exponentiation (with its own semantics for hexadecimal exponentiation).

Any PHANTASM number literal (integer or float, decimal or hexadecimal) can be suffixed with the raise-operator (`\`) or lower-operator (`/`), followed by any natural number, which causes the value to be raised or lowered by that number of orders of magnitude. For example:

```
1\3           | equal to 1,000
1/3           | equal to 0.001
1.5\6         | equal to 1,500,000
#FF\6         | equal to 4,278,190,080, or #FF000000
#1.F/4        | equal to #0.0001F
```

Note: Unlike WAT (and the C language-family generally), in PHANTASM the number after the operator is alway expressed using both the same base and the same notation as the number before the operator:

```
#FF\A         | equal to #FF0000000000
#1.F/10       | equal to #0.0000000000000001F
10\A          | unrecognized token (`A` is not a decimal digit)
```

The floating-point constants (positive infinity, negative infinity and not-a-number), are expressed exactly like they are in JavaScript (`Infinity` or `+Infinity`, `-Infinity` and `NaN`).

The wiki article *Number Literals* provides more details, including how literals are evaluated and encoded, depending on the context (for example, `i32 1` and `f32 1` use the same literal, but the compiler encodes each one differently). The article also covers edgecases, like using floating point notation with an exponent to express an integer (which affects the range of integers that can be expressed).

The PHANTASM Grammar

Each PHANTASM file (implicitly) defines exactly one WebAssembly module.

At the top level, a module is simply an array of *directives*. Each directive occupies its own logical line (they are never indented, nor grouped together).

Each directive does one of three things to a *component*: A directive can *define*, *import* or *export* a component. Correspondingly, there are three types of directive (*define-directives*, *import-directives*, and *export-directives*).

Directives

Of the three kinds of directive, define-directives have the simplest grammar:

```
define component-definition
```

Import-directives have the most complicated grammar, as they have an optional part that defines the module-name string (that defaults to `"host"` when omitted):

```
import "field" [from "module"] as component-specifier
```

Export-directives use the following grammar:

```
export "field" as component-reference
```

The three directive-grammars describe the entire language at the top level:

```
module : directive*
directive : define component-definition
          | import "field" [from "module"] as component-specifier
          | export "field" as component-reference
```

The three *component-descriptors* (*component-definitions*, *component-specifiers* and *component-references*), which the rest of this document describes, complete the module grammar.

Component References

Of the three kinds of component-descriptor, the component-reference has the simplest grammar:

```
component identity
```

It is simply the component type, followed by its identity. For example:

```
register 1
function $sum
memory 0
table $opcodes
```

Memory-references and table-references can omit their identity (as with memory and table identities generally), and zero will be inferred. For example, `memory 0` can be shortened to just `memory`.

Furthermore, function-references (as with functions generally) can omit the component-type (`function`) when an identity is present, so `function $foo` can be shortened to just `$foo`, and `function 9` can be shortened to just `9`.

Component-references are primarily used in export-directives. For example:

```
export "$foo" as function $foo         | these two statements
export "$foo" as $foo                  | are equivalent
export "five" as register 5
export "DATA" as memory 0              | these two statements
export "DATA" as memory                | are also equivalent
```

While types and banks (function types, memory banks and table banks - see below) cannot be imported or exported (support for importing and exporting types still needs implementing), there are other places in the language where these components are referenced, and the same component-reference grammar is used. For example:

```
type $binop
memory bank 0
table bank $extensions
```

The above component-references are directly used by various directives, as well as the instructions they contain. For example:

```
import "helper" as $helper of type $binop

invoke type 5
copy memory bank $messages
```

Note: Memory banks and table banks *cannot* omit their identities (like memories and tables can).

Component Specifiers

Component-specifiers are used by import-directives, and reused by define-directives (define-directives actually use component-*definitions*, but the grammar for definitions just appends the appropriate type of block to the end of the grammar for specifiers).

Register Specifiers

Register-specifiers begin with a *register-qualifier*, either `variable` or `constant`, followed by a valtype that specifies the type of the register, then an optional identifier, which (when present) will be bound to the newly imported (or defined) register:

```
qualifier valtype [identifier]
```

For example:

```
variable i32
constant pointer $pointer
```

Note: Register-specifiers do not use the `register` component-type. The register-qualifier before the valtype makes the component-type implicit.

Register-specifiers are used in import-statements. For example:

```
import "score" as variable i64
import "PI" from "/std/math" as constant f64 $PI
```

Function Specifiers

Function-specifiers use the `function` component-type, prefixed by the `initializer` qualifier when specifying a initializer function. The `function` component-type is followed by an optional identifier, then an optional type, which (as always) implies `void → void` when omitted:

```
[initializer] function [identifier] [of type]
```

As `function` is essentially the default component type, the `function` component-type becomes optional when the qualifier or identifier is present:

```
[initializer] [function] identifier [of type]
initializer [function] [identifier] [of type]
```

To clarify, one or more of the first three parts (the `initializer` qualifier, the `function` component-type or the `identifier`) is enough to specify a function. The `type` part is always optional, but cannot specify a function on its own.

The type of the specified function can be described with a *type-reference* (as described above, in the *Component References* section), or with a *type-expression* (described next, in the *Type Expressions* section). For example:

```
function of type $binop
$sum of i32, i32 → i32
```

Function-specifiers are primarily used by import-directives. For example:

```
import "sum" from "mathlib" as $sum of type $binop
import "init" as initializer function
```

Type Expressions

Type-expressions express function-types (generally just referred to as *types*) by mapping one list of valtypes (the *arity*) to another (the *results*), with an arrow-symbol between them (→).

```
arity → results
```

Both the arity and results are expressed as a comma-separated lists of valtypes, or `void` when the list is empty. For example:

```
i32, i32 -> i32
pointer -> f64, f64
pointer -> i64
f32, proxy -> void
```

Type-expressions are used in various statements and instructions. For example:

```
import "pointer.get" as $get_pointer of i32 -> pointer
```

```
invoke i32, i32 -> i32
branch of i32, i32 -> i32 ...
```

Note: Function-*definitions* can use a *signature* to define their type (explained below).

Memory Specifiers

Memory-specifiers begin with an optional `shared` qualifier, followed by the component name `memory`. The component name is followed by an optional identifier (to be bound to the newly imported memory), which is in turn followed by the *limits* of the memory:

```
[shared] memory [identifier] limits
```

The limits-construct starts with the `with` prefix, followed by a number-literal that defines the length of the memory (in pages):

```
with number
```

The above grammar defines memories of a fixed length. It can be extended in one of two ways to define memories that are able to grow:

```
with number to number
with number plus
```

In the later case (where the `plus` qualifier is used), there is no maximum length, which makes those limits invalid for shared memories (which must define a maximum length, whether implicitly or explicitly).

Below are some examples of memory specifiers:

```
memory with 1
shared memory $store with #10 to #20
memory $current_track with 16 plus
```

Memory-specifiers are used in import-directives. For example:

```
import "ram" from "sys" as shared memory with #100
import "audio" as memory $audio with 16 plus
```

Table Specifiers

Table-specifiers use a grammar that is almost identical to memory-specifiers. However, in a table-specifier, the qualifier is required, and must be one of `pointer`, `proxy` or `reference` (equivalent to the WAT reftypes `funcref`, `externref` and `anyref`, respectively). For example:

```
pointer table
proxy table
reference table
```

As with memories, the component name is followed by an optional identifier (to be bound to the newly imported table), which is followed by the limits:

```
table-qualifier table [identifier] limits
```

Note: The limits are expressed as the number of slots (not pages) when used to specify a table (as opposed to a memory).

Below are a few examples of table-specifiers:

```
proxy table with 256
pointer table $opcodes with #100 to #200
reference table $extensions with 16 plus
```

Component Definitions

Component-definitions are used by define-directives.

Along with component-references and component-specifiers (covered above), component-definitions make up the all three of the component-descriptors.

Component-definitions reuse the grammar of the component-specifiers (just described above), appending (indented or inline) blocks that define the various components. That said, those blocks include the bodies of functions (with all of the instructions), as well as the primers that are used to populate memories, tables and banks (with all of the data and reference elements they include), so there is still quite a lot of ground to cover.

Register Definitions

Register-definitions append an optional expression to the grammar for register-specifiers, which is used to initialize the register:

```
qualifier valtype [identifier] [register-expression]
```

When the expression is omitted, numtype registers are initialized to `0`, while reftype registers are initialized to `null` (a constant-expression containing the appropriate instruction is generated by the assembler automatically). For example, the following directive defines a writable register for a 32-bit integer, which will be initialized to zero (and identified as `$score`):

```
define variable i32 $score
```

When the expression is present, it can be a regular constant-expression (a block of one or more instructions). For example:

```
define variable f64 $master.gain
    push f64 1
```

Single-line blocks (generally) can be inlined using the `as` prefix. So, for example, the previous example could be written like this:

```
define variable f64 $master.gain as push f64 1
```

Register-definitions also support using the `with` prefix to directly specify the initial value, allowing the previous examples to be shortened to this:

```
define variable f64 $master.gain with 1
```

The `with` prefix can also be used with pointer registers to reference an identity for a function (which gets compiled to a constant expression using the `ref.func` instruction):

```
define variable pointer $foo with $bar
define constant pointer with 1
```

Note: The `with` prefix *cannot* be used to initialize proxy-registers (what would it compile to?).

Function Definitions

Function-definitions append a required block, the *function-body*, to the grammar for function-specifiers, and replace the type-construct with a *signature-construct*, producing the following grammar:

```
[initializer] function [identifier] [of signature] body
```

As described in the *Function Specifiers* section, the presence of the qualifier, the component-type or the identifier renders the other two optional:

```
[initializer] [function] identifier [of signature] body
initializer [function] [identifier] [of signature] body
```

The signature-construct extends the grammar of type-expressions, permitting (though never requiring) an identifier after each valtype in the arity-construct. For example:

```
i32 $x, i32 $y, i32 $z → void
```

The signature grammar also permits the type of a given parameter to be inferred from the previous parameter (recursively). So, the previous example could be abbreviated to the following signature:

```
i32 $x, $y, $z → void
```

The `body` begins with zero or more *register-preambles*, followed by its instructions (see below).

Directives, Preambles & Commands

The grammar is described in terms of three kinds of *statement*: *directives*, *preambles* and *commands*.

Directives were described already (in the *Module Grammar* section above). In short, they define a module at the top level.

Preambles introduce the (one or more) *segments* that blocks are broken into. Preambles always occupy their own logical line (exactly like a directive, except indented), and therefore, blocks containing preambles cannot be inlined. There are (currently) two kinds or preamble: *Register-preambles* (described below) and *segment-preambles* (explained later, in the *Memory Definitions* and *Table Definitions* sections).

Commands include the *instructions* that define the logic of functions and expressions, the *data* that populate memories and memory banks, and the *references* that populate tables and table banks.

Unlike directives and preambles, multiple commands can be grouped on the same logical line, using a comma-separator. For example:

```
get 0, get 1, add i32        | instructions (commands used by functions and expressions)
i8 0, 1, 2, f64 0, 1.5       | data (commands used to populate memories and memory banks)
pointer 0, 1, 2, null, $f    | references (commands used to populate tables and table banks)
```

Register Preambles

Register-preambles begin with the `@register` declarator, followed by the valtype of the register, which is in turn followed by an optional identifier (which is bound to index of the implied local register):

```
@register valtype [identifier]
```

Below are a couple of examples:

```
@register i32
@register pointer $helper
```

Multiple registers can be declared in a compounded preamble. For example:

```
@register i32 $x, i32 $y, i32 $z
```

As with parameters, the type can be inferred from the previous element:

```
@register i32 $x, $y, $z
```

A single register-preamble can also define different types of register. For example:

```
@register i32 $x, $y, $z, i64 $a, $b, $c
```

As with parameters, when an identifier is omitted, the type is required:

```
@register i32, i32, i32, i64, i64, i64
```

The Instructions

After any register-preambles, the body of a function (which is always a single segment) contains one or more instructions (which are grammatically commands).

Note: In practice, you can group the block-instructions `block`, `loop` and `branch` like any other instructions, except that block-instructions (obviously) end with their own block of instructions, so nothing can be grouped *after* a block-instruction (a block-instruction will always follow any instructions it is grouped with). This also implies that `else` with never be grouped with anything.

To avoid the issues that CoffeeScript had with cryptic whitespace, the grammar enforces a pair of complementary rules that state that *block-instructions cannot use inline blocks*, and *inline-blocks cannot use block-instructions*. Otherwise, stuff like this becomes permissible:

```
define function of i32 → void as branch of type 0 as call $foo
else as call $bar
```

The individual instructions are documented in the following sections.

The WAT Mnemonics

The following four instructions all use single-token mnemonics that are copied directly from WAT:

```
nop
return
else
drop
```

The Crash Mnemonic

The WAT `unreachable` mnemonic has simply been renamed to `crash`:

```
crash
```

The Get, Set & Put Mnemonics

PHANTASM uses the mnemonics `get` and `set` for the WAT instructions `global.get`, `global.set`, `local.get`, `local.set`, `table.get` and `table.set`.

Both `get` and `set` take an optional scope-qualifier, which when present, must be one of `global`, `local` or `table`, defaulting to the lexical scope of the instruction, so `local` within functions, and `global` otherwise:

```
mnemonic [scope-qualifier] identity
```

Below are a few examples of the `get` and `set` mnemonics:

```
get $x
set local 6
get table $opcodes
```

PHANTASM uses the `put` mnemonic for the WAT `local.tee` instruction. Unlike `get` and `set`, there is no scope-qualifier in the `put` grammar (`put` is always implicitly local):

```
put identity
```

The Push Mnemonic

PHANTASM has a `push` mnemonic that is used for all of the WAT instructions that push a value to the stack: `i32.const`, `i64.const`, `f32.const`, `f64.const`, `ref.func` and `ref.null`.

The `push` mnemonic uses the following grammar:

```
push [valtype] immediate
```

The valtype defaults to `i32`, and the immediate must be valid for the (given or implied) valtype: When the valtype is a numtype, the immediate must be an appropriate number literal. When the valtype is `pointer`, the immediate must be an identity or `null`, and when the valtype is `proxy`, the immediate must be `null`.

Below are examples of how the `push` mnemonic is used more generally:

```
push #30                  | i32.const 0x30
push f64 0.5              | f64.const 0.5
push pointer $helper      | ref.func $helper
push pointer null         | ref.null func
push proxy null           | ref.null extern
```

Note: The WAT instruction `ref.is_null` is handled by the PHANTASM instruction `is null` (described below).

The Load & Store Mnemonics

PHANTASM uses the `load` and `store` mnemonics for all of the WAT load and store instructions, including those that load or store a *datatype* that differs from the register-type.

The `load` and `store` mnemonics use the following grammar:

```
mnemonic numtype [as datatype] [in identity] [at number]
```

The `as` prefix is used to specify the datatype (when it differs from the register-type). The `in` prefix introduces the identity of the memory (when it is not implicitly `0`). The `at` prefix introduces the offset of the load or store into memory (when it is not implicitly `0`).

Below are some examples of load and store instructions:

```
load i32                          | i32.load
store f64                         | f64.store
store i64 as u32                  | i64.store32_u
load i64 as i8 at 3               | i64.load8 offset=3
store i32 as s8 in $memory        | i32.store8_s $memory
load i64 as i8 in $RAM at 1       | i64.load8 $RAM offset=1
```

Note: PHANTASM never uses signed or unsigned mnemonics (as WAT does). Instead, PHANTASM always indicates whether a given instruction operates on signed, unsigned or agnostic integers using signed, unsigned and agnostic types.

Note: The atomic instructions generally use the same grammar as the `load` and `store` instructions. This is fully documented below.

The Arithmetic & Logic Mnemonics

PHANTASM uses essentially the same grammar for all arithmetic operations:

```
mnemonic ~numtype
```

The valid set of numtypes depends on the individual instruction. Some instructions operate on both integers and floats, while others only work with integers or floats. Furthermore, of those that operate on integers, some require explicitly signed or unsigned integers, while others are type-agnostic.

The following list of mnemonics all use the above grammar, and have names (and semantics) that are copied directly from WAT:

```
add    sub    mul    div    rem    and    or     xor
abs    neg    min    max    clz    ctz    floor  nearest
```

The following mnemonics all use different names to their WAT equivalents, however they are otherwise the same as the mnemonics in the previous list:

```
sign   ceiling          | copysign    ceil
lop    root             | trunc       sqrt
nsa                     | popcnt
```

Below are a handful of examples, showing how the grammar for arithmetic and logic looks in practice:

```
add i64
abs f64
xor i32
div u32
rem s64
```

The Shift & Rotate Mnemonics

PHANTASM uses the `shift` and `rotate` mnemonics for all of the `shl`, `shr`, `rotl` and `rotr` WAT instructions (including gnostic instructions, like `i32.shr_s` *et cetera*).

The grammar is identical to the grammar for arithmetic and logic instructions, except that it appends a shift qualifier, one of `left` or `right`, to the end:

```
mnemonic numtype qualifier
```

Naturally, the numtype must be either `i32` or `i64`, except for shift-right instructions, where the type is gnostic (`u32`, `s32`, `u64` or `s64`). For example:

```
shift i32 left
shift s32 right
rotate i32 right
rotate i64 left
shift u64 right
```

The Call & Invoke Mnemonics

PHANTASM copies the `call` mnemonic from WAT, but uses the `invoke` mnemonic for `call_indirect`.

The grammar for the call-instruction simply appends an identity to the mnemonic:

```
call identity
```

The grammar for the invoke-instruction requires a type, and uses the `in` prefix for the table identity:

```
invoke type [in identity]
```

As always, the type can be expressed with a type-reference or type-expression. For example:

```
call 12
call $helper
invoke type $binop
invoke i32, i32 -> i32
invoke type $check in $checks
invoke i32 -> void in 2
```

The Branch Instructions

PHANTASM renames all three of the WAT branch-instructions, from `br`, `br_if` and `br_table` to `jump`, `fork` and `exit`, respectively.

In PHANTASM terms, a *jump* is unconditional, while a *fork* is a conditional jump. An *exit* always exits the current block, but uses an operand and an (immediate) array of indices to determine which exit to take.

The jump-instruction and fork-instruction use a grammar that directly appends a label for the target block to the mnemonic:

```
mnemonic label
```

Labels and identities are syntactically indistinct. However, there are no indexspaces for labels.

The exit-instruction requires one or more (space-separated) labels (in the same order as WAT):

```
mnemonic identity+
```

For example:

```
jump 0
fork $block
exit 1 0 2 $loop
```

The Block Instructions

PHANTASM copies the `block` and `loop` mnemonics from WAT, and also uses `else`. However, PHANTASM uses the `branch` mnemonic instead of `if` (so `else` blocks optionally follow `branch` blocks).

All three branch instructions (`block`, `loop` and `branch`) use the same grammar, which is similar to the function grammar:

```
mnemonic [label] [of type] block
```

When present, the identifier is bound to the implied label. The type can be expressed with a type-reference or a type-expression (and defaults to `void → void`). For example:

```
loop $loop of type 2
    | instructions...

block of i32, i32 → void
    | instructions...

branch $branch of type $check
    | instructions...
```

When present, the `else` mnemonic is indented to the same level as the preceding `branch` mnemonic:

```
branch $branch of type $check
    | instructions...
else
    | instructions...
```

Note: The blocks must be indented (block-instructions cannot use inline-blocks, and inline-blocks cannot use block-instructions).

Note: PHANTASM does not require whitelines anywhere in the grammar, but you are recommended to use one before and after each block header and docstring, as well as between chunks of related directives, preambles and commands (as in any other language). Adjacent whitelines are never recommended.

The Select Mnemonic

PHANTASM copies the `select` mnemonic from WAT, with the same grammar:

```
select [valtype]
```

For example:

```
select
select i32
select pointer
```

The Memory & Table Instructions

PHANTASM uses the `grow`, `size` and `fill` mnemonics from WAT, followed by a memory-reference or table-reference:

```
mnemonic component-reference
```

For example:

```
grow memory
size memory 1
fill table $opcodes
```

The Drop Mnemonic

PHANTASM uses the `drop` mnemonic from WAT, with the same grammar as the memory and table instructions (above), except that the component-reference must reference a bank:

```
mnemonic component-reference
```

For example (noting that banks cannot use implicit identities):

```
drop memory bank 0
drop table bank $extensions
```

The Copy & Initialization Instructions

PHANTASM uses the `copy` mnemonic for the WAT instructions `copy` and `init`, with the following grammar:

```
mnemonic component-reference [to identity]
```

The component-reference defines the *location* (the component to copy *from*). It can reference a memory or table (implementing the WAT copy-instruction), or a memory bank or table bank (implementing the WAT init-instruction).

The `to` prefix is used to identify the *destination* (the component to copy *to*). As ever, it defaults to `0`. Given that banks are readonly, the destination type can be inferred from the location type (`memory` for memories and memory banks, and `table` for tables and table banks). For example:

```
copy memory to 5
copy table $opcodes
copy table $messages to $messages
copy table bank $extensions to $opcodes
copy memory bank 0 to 5
copy table bank 1
```

The Is & Not Mnemonics

PHANTASM uses the `is` and `not` mnemonics for the WAT mnemonics `eqz`, `eq`, `ne`, `gt`, `lt`, `ge` and `le`, using the following grammar:

```
mnemonic test
```

When the mnemonic is `is`, the test can be one of `null`, `zero`, `equal`, `less` or `more`. When the mnemonic is `not`, the test can only be one of `equal`, `less` or `more`. For example:

```
is zero
is null
not more
is less
not equal
```

The Conversion Instructions

PHANTASM reuses the `wrap`, `convert`, `promote` and `demote` mnemonics with the following grammar:

```
mnemonic numtype to numtype
```

For example:

```
wrap i64 to i32
convert s32 to f64
promote f32 to f64
demote f64 to f32
```

PHANTASM also uses the same grammar for its `bitcast` and `lop` mnemonics, which replaces the WAT mnemonics `reinterpret` and `trunc`, respectively. For example:

```
bitcast i32 to f32
bitcast i64 to f64
bitcast f32 to i32

lop f32 to u32
lop f32 to s64
```

PHANTASM uses the same grammar and mnemonic for its version of the WAT `trunc_sat` instructions, but replaces the `to` prefix with the `sop` prefix. For example:

```
lop f32 sop u32
lop f32 sop s64
```

The PHANTASM `extend` mnemonic is used for the two WAT extend-instructions that sign-extend or zero-extend a 32-bit operand to produce a 64-bit result. For example:

```
extend u32
extend s32
```

Note: The other WAT extend-instructions use the PHANTASM `expand` mnemonic (see below).

The Expand Instructions

PHANTASM uses the `expand` mnemonic for those WAT extend-instructions that sign-extend the least-significant 8, 16 or 32 bits of the operand, filling the most significant bits appropriately. Unlike the `extend` mnemonic, `expand` produces a result that is the same width as its operand (expanding the value in place).

The expand-instructions use the `as` prefix to specify a datatype, similar to the `load` and `store` instructions, and most of the atomic instructions (described below):

```
expand numtype as datatype
```

The numtype describes the the operand (and result), while the datatype describes the smaller sub-value that will be sign-extended (one of `s8` or `s16`, or if the numtype is `i64`, then `s32`). For example:

```
expand i32 as s8
expand i32 as s16
expand i64 as s32
```

The Atomic Instructions

Most of the PHANTASM atomic-instructions (`load`, `store`, `add`, `sub`, `and`, `or`, `xor`, `trade` and `broker`) use the same grammar as the non-atomic `load` and `store` instructions, prefixed by the `atomic` qualifier:

```
atomic mnemonic numtype [as datatype] [in identity] [at number-literal]
```

Note: In practice, the numtype of any atomic-operation will be an integer.

PHANTASM uses the `trade` and `broker` mnemonics for the WAT mnemonics `xchg` and `cmpxchg`, respectively.

The various constructs of the grammar are interpreted in the same way as the `load` and `store` instructions. For example:

```
atomic load i32
atomic store i64 as i8
atomic add i32 in $memory
atomic broker i32 as i16
```

The atomic-wait-instructions use a similar grammar to the regular atomic-instructions, just without the (optional) datatype:

```
atomic wait numtype [in identity] [at number-literal]
```

The atomic-notify-instruction uses the same grammar as the atomic-wait instructions, but without the (required) numtype:

```
atomic notify [in identity] [at number-literal]
```

Below are a few examples of how the `wait` and `notify` mnemonics are used:

```
atomic notify
atomic wait i32
atomic wait i64 at 4
atomic notify in $memory at 2
```

The atomic-fence-instruction is simply written as follows:

```
atomic fence
```

The atomic-fence instruction completes the section on PHANTASM instructions, and with it, the section on function-definitions. The rest of this document describes the component-definitions for types, memories and tables (including banks).

Memory Definitions

Memory-definitions append an optional *memory-primer* to the grammar for memory-specifiers:

```
[shared] memory [identifier] limits [primer]
```

When the primer is omitted, the memory is implicitly empty. When present, the primer must contain one or more *memory-segments*. Segments are defined using a *segment-preamble* (see below), followed by one or more *data-commands* (see after).

Segment Preambles

A segment-preamble specifies an offset, using the following grammar:

```
@segment block
```

The block is a constant expression that returns the offset. For example:

```
@segment as push #100
```

Alternatively, segment-preambles can use the `at` prefix to directly specify an offset:

```
@segment at number-literal
```

For example:

```
@segment at #100
```

Data Commands

Data-commands define the chunks of data that are used to populate memories when they are initialized. All data-commands use the following grammar:

```
mnemonic value
```

The mnemonic must be one of `i8`, `i16`, `i32`, `i64` or `utf8`. When the mnemonic is a numtype, the value must be a valid number literal (for the given type). When the mnemonic is `utf8`, the value must be a string literal.

As you might expect, the mnemonic can be omitted and inferred from the previous command when they use the same mnemonic, and multiple commands can be compounded on the same line. For example:

```
utf8 "Hello, World!"
i8 #10, #20, i16 #30, #40
```

Memory Primers

Memory-primers are blocks that contain one or more segments, each defined by a segment-preamble, followed by one or more data-commands. For example:

```
define shared memory with 16

    @segment #500                   | the first segment preamble
    i8 #10, #20, #30, #40           | four (compounded) data commands
    i16 #10, #20, #30, #40          | some more data commands

    @segment #1000                  | the second segment preamble
    utf8 "Hello, World!”            | a data command using a UTF-8 string
    i8 #10, #20, i16 #30, #40       | a data command with mixed types
```

The first segment in any primer can omit its preamble, and the offset is implicitly zero:

```
define memory with 1 plus

    utf8 "Congratulations... You Won!"
    utf8 "Commiserations... You Lost!"
```

When the preamble (which must be on its own logical line) is omitted from a primer with only one segment, the primer may then be inlined. For example:

```
define memory with 1 plus as i8 #10, #20, #30
```

Memory Bank Definitions

Memory banks are compiled to passive memory segments (within the binary), serving as little ROMs that can be copied from at runtime.

PHANTASM compiles all passive segments ahead of any active segments, so memory banks and table banks each have their own indexspaces (effectively).

Note: The indices of active segments are not accessible (nor useful) to the user. Attempting to access one (using a number literal to specify its index) will result in a compile-time error.

The grammar for memory-bank-definitions is derived from the grammar for memory-definitions. The limits-construct is removed, while the primer becomes required:

```
memory bank [identifier] primer
```

Note: Bank primers cannot contain segment-directives (a bank *is* a segment).

Memory-bank-definitions are used by define-statements. For example:

```
define memory bank $bank

    utf8 "Hello, World!"
    i8 #10, #20, i16 #30, #40
```

As ever, blocks that only contain a single line of commands can be inlined:

```
define memory bank as i8 #10, #20, i16 #30, #40
```

Table Definitions

Table-definitions append an optional *table-primer* to the grammar for table-specifiers:

```
qualifier table [identifier] limits [primer]
```

Most of the parts of the above grammar have been described already (in the *Table Specifiers* and *Memory Definitions* sections). The qualifier must be one of `pointer`, `proxy` or `reference`.

The primer contains *reference-commands* (see below), and table-definitions can only (currently) include primers (at all) if the table-type is `pointer`.

Reference Commands

The reference-commands used to populate tables use the same grammar as the data-commands used to populate memories:

```
mnemonic value
```

The mnemonic must (currently) be `pointer`. More reference-commands will be added soon.

The value is a function identity or `null`. As usual, the mnemonic can be inferred from the previous command and compounded. For example:

```
pointer 14, 9, null, $helper, null
```

Table Primers

Table-primers (when present) are blocks that contain one or more *table-segments*. Each segment is defined by a segment-preamble (exactly like a memory-segment, except that the offset is slotwise, instead of bytewise), followed by one or more reference-commands.

The preamble can be omitted from the first segment (and implicitly zero). For example:

```
define pointer table $opcodes with #1000

    pointer $nop, $jsr, $rts

    @segment #100
    pointer 1, 2, 3, null, null, null
    pointer $foo, $bar, $spam, $eggs

define pointer table with 256 plus as pointer $nop, $jsr, $rts
```

Table Bank Definitions

Table banks are compiled to passive table-segments (within the binary), just like memory banks, and also have their own indexspace.

The grammar for table-bank-definitions is the same as for memory banks, except that the component name is replaced by the qualifier (one of `pointer`, `proxy` or `reference`):

```
qualifier bank [identifier] primer
```

In fact, table banks must (currently) use the `pointer` type, so in practice, the effective grammar is more restrictive than above:

```
pointer bank [identifier] primer
```

Bank primers cannot contain segment-preambles, so the primer will consist of one or more reference-commands, which must each use the `pointer` mnemonic (at the moment).

Table-bank-definitions are used by define-directives. For example:

```
define pointer bank $bank as pointer $nop, $jsr, $rts
```

Type Definitions

Type-definitions define function-types. The type-expression is not a block, so it is always inline, and prefixed by `as`:

```
type [identifier] as type-expression
```

Naturally, type-definitions are used by define-directives. For example:

```
define type $binop as i32, i32 → i32
```

There are six places in the grammar where a type can be referenced (function-specifiers, function-definitions, the three block-instructions and the invoke-instruction). In all six cases, the type can either be referenced with a type-reference (like `type $binop`) or expressed with a type-expression (like `i32, i32 → i32`).

Explicit & Implicit Types

Every function type is unique (within its module). Types are either defined explicitly (using a type-definition, as above) or implicitly (by using them in one or more type-expressions).

Explicitly defined types are always indexed ahead of any implicitly defined types. Furthermore, implicitly defined indices are treated as out of bounds, effectively providing explicitly defined types with their own indexspace.

Type-definitions will not compile if they define a type that was already explicitly defined by a previous type-definition, preventing explicit duplication. Likewise, type-expressions that express existing types are replaced with references to the existing type, avoiding implicit duplication.

Note: The lack of duplicate types neatly sidesteps an issue WAT has, where a function-definition cannot reference a type *and* bind identifiers to the parameters of the type, without duplicating the type or requiring the author to include a type-reference *and* a signature (which, naturally, must express the same type).

Going Forwards

Now that you have finished reading the Crashcourse, you should grab a copy of the *PHANTASM: Abstract Grammar Cheatsheet*., which summarizes the grammar in BNF-style pseudocode. The project wiki also contains a growing collection of useful articles:

- *Installation*: How to get the code and use it.
- *String Literals*: Details string literals, particularly interpolation.
- *Number Literals*: Details number literals, particularly exponentiation and evaluation.
- *Roadmap & Status*: This sets out the general future direction. Take it with a pinch of salt.

Please feel free to use the project issue tracker as a general forum for any discussion.