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 Text Format.

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

PHANTASM Files

PHANTASM source strings are 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*).

Character Sets

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, though in those contexts, there are no restrictions on the character set.

Special Characters

Like WAT, PHANTASM uses the longest-match rule. PHANTASM also uses the same set of special characters (whitespace characters, commas and semicolons, as well as parens, brackets and braces). However (outside of string literals and commentary), PHANTASM does not actually use semicolons, parens, brackets or braces, nor tabs.

The three special characters that *are* used in regular code (spaces, newlines and commas) are the characters that structure the code, while the regular characters (non-special ASCII printables) are used to spell the regular tokens.

Significant Whitespace

PHANTASM uses significant indentation and significant newlines, along with the Comma Character, to delimit tokens, logical lines and blocks (the result looks 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, a comment or a special character) follows the longest-match rule, and is then classified as belonging to one of six token types: *keyword*, *mnemonic, decorator, operator*, *identifier* or *number*.

Keywords

Keywords are used to begin statements, and to create continuations within statements. They are the most important tokens in the language, and provide the various statements of the language with a consistent structure.

Keywords are always simple English words, spelled in lowercase:

`define`	`import`	`export`		
`with`	`from`	`thus`	`plus`	`sop`
`of`	`to`	`at`	`in`	`as`

Note: A policy on reserved words would be redundant.

Mnemonics

Mnemonics are simple, little names that the language assigns to its instructions, components, types *et cetera*. This includes the names of pseudo types (like `s8` and `utf8`), and the qualifiers used to qualify other mnemonics (as in `pointer table`, `shared memory`, `start function` *et cetera*).

Note: PHANTASM renames the WAT reftypes `funcref` and `externref` to `pointer` and `proxy` respectively.

Mnemonics are lexically similar to keywords, except that mnemonics may also contain digits (but will not begin with one). This is the complete list of mnemonics:

`i8`	`s8`	`u8`	`i16`	`s16`	`u16`	`i32`	`s32`	`u32`
`i64`	`s64`	`u64`	`f32`	`f64`	`utf8`	`void`	`null`	`nop`
`shared`	`memory`	`pointer`	`proxy`	`mixed`	`table`	`bank`	`type`	`atomic`
`variable`	`constant`	`register`	`start`	`function`	`call`	`invoke`	`return`	`select`
`global`	`local`	`left`	`right`	`equal`	`more`	`less`	`zero`	`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`	

Decorators

Decorators are like keywords, except that they begin with the At Character (`@`), and are used to introduce *preambles* (described later).

There are currently two decorators (though at least one more will be added when support for compile-time expressions is implemented):

```
@segment   @register
```

Operators

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

The arrow-operator 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-operator is used to continue a logical line across two or more lines of a source file. The operator 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-operator is insignificant.

Identifiers

PHANTASM identifiers are similar to WAT identifiers, having the same dollar-prefix (`$foo`, `$bar`, `$player::score` *et cetera*). However, while PHANTASM identifiers can contain any combination of regular characters, Unicode characters are unsupported (outside of strings and commentary).

Semantically, identifiers are always constant. Exploiting that, identifier resolution gets deferred until all of the indices (within each indexspace) have been assigned (and the identifiers are bound to their respective indices). This permits identifiers to be used before they are bound (just like explicit indices), in turn allowing statements to be ordered in the most natural, readable way (which is not always possible in WAT code).

Note: PHANTASM uses the term *identity* to refer to an index, expressed either directly (with a number literal) or indirectly (with an identifier bound to the index). There are many constructs in the language that accept or require an identity.

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 #0.00000000000000001F
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).

We have already covered the basics of PHANTASM grammar (significant whitespace, special characters and the longest-match rule), as well as comments and the six regular token types (keywords, mnemonics, decorators, operators, identifiers and numbers).

The rest of the document describes the PHANTASM module grammar, from the top down.

The Module Grammar

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

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

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

The components are divided into two types: *system-components* (*registers*, *functions*, *memories* and *tables*) that can be defined, imported and exported, and *internal-components* (*function types*, *memory banks* and *table banks*) that can only be defined.

Statements

Of the three kinds of statement, define-statements have the simplest grammar:

```
define component-definition
```

Import statements 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 statements use the following grammar:

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

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

```
module : statement*
statement : 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*) complete the statement grammar.

Note: The *component-definition* grammar (used by define-statements) is a superset of the *component-specifier* grammar (used by import-statements). Furthermore, the *component-reference* grammar (used by export-statements) is naturally very simple (just one or two tokens). So, while there are seven kinds of component, and three different contexts, the various component-descriptors this gives rise to all share a relatively simple and consistent syntax.

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 name (`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 used in export-statements. 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) are (at least currently) all internal-components (they cannot be imported or exported), there are other places in the language where these types of components are referenced, and the same component-reference grammar is used. For example:

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

These internal component-references are used in various statements and instructions. 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 in import-statements. As such, there are four kinds, one for each kind of system component (register, function, memory and table).

Register Specifiers

Register-specifiers begin with a *register-qualifier*, either `variable` or `constant`, followed by the valtype, then an optional identifier, which will be bound to the newly imported register:

```
qualifier valtype [identifier]
```

For example:

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

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` keyword, prefixed by the `start` qualifier when specifying a start function. The `function` keyword is followed by an optional identifier, then an optional type, which (as always) implies `void -> void` when omitted:

```
[start] function [identifier] [of type]
```

When the identifier is present, the `function` keyword becomes optional:

```
[start] [function] identifier [of type]
```

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 used in import-statements. For example:

```
import "sum" from "mathlib" as $sum of type $binop
import "percent" as function of f32, f32 -> f32
```

Type Expressions

Type-expressions express function types by mapping one list of reftypes (the *arity*) to another (the *results*), using the arrow-operator (`→`).

```
arity → results
```

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

```
i32, i32 → i32
i32 → f64, f64, pointer
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 keyword `with`, followed by a number literal that defines the length of the memory (in pages):

```
with number
```

The above grammar defines memories with 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 `plus` is used), there is no maximum length, which makes that form of limits-construct invalid with shared memories (which must define a maximum length, whether implicitly or explicitly).

Below are some examples of memory specifiers (each specifying its limits):

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

Memory-specifiers are used in import-statements. 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 `mixed` (equivalent to the WAT reftypes `funcref`, `externref` and `anyref`, respectively). For example:

```
pointer table
proxy table
mixed table
```

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

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

Note: The minimum and maximum lengths (within a limits-construct) 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
mixed table $extensions with 16 plus
```

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

```
import "opcodes" as pointer table $opcodes with #100 to #200
import "extensions" from "sys" as proxy table with 256
```

Component Definitions

Component-definitions are used in define-statements.

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 (used by import-statements), extending it as required. However, those grammatical extensions 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 memory and table elements they include), so there is still quite a lot of ground to cover.

Register Definitions

Register-definitions append an optional initializer to the grammar for register-specifiers:

```
qualifier valtype [identifier] [initializer]
```

The initializer is used to define the constant expression that the engine will use to initialize the register.

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

```
define variable i32 $score
```

When an initializer is provided, it can be expressed with a block of one or more instructions. For example:

```
define variable f64 $master.gain
    push f64 100
```

Single-line blocks can be inlined using the `thus` keyword. For example:

```
define variable f64 $master.gain thus push f64 100
```

Register-definitions also support sugar for blocks that simply push a constant (as in the example above). This grammar uses the `as` keyword to inline the initializer, allowing numtype registers to be initialized like this:

```
define variable f64 $master.gain as 100
```

The `as` keyword can also be used with pointer registers to inline an identity that references a function (which all gets compiled to a constant expression using the `ref.func` instruction):

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

The `as` keyword cannot be used to initialize proxy-registers, as there is no analog of `ref.func` for proxies (externrefs) that the sugar could sensibly compile to.

Function Definitions

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

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

As always, the presence of the identifier makes the `function` component optional:

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

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

```
i32 $x, i32 $y, i32 $z → i32
```

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 compounded to produce the following signature:

```
i32 $x, $y, $z → i32
```

The `body` construct begins with zero or more *local-preambles*, followed by a block of instructions.

Statements, Preambles & Commands

The grammar is primarily described in terms of *statements*, *preambles* and *commands*.

Statements were described already (in the *Statement Grammar* section above). In short, they define a module at the top level. Preambles can appear at any level, and commands are always contained within blocks.

Preambles must occupy their own logical line (like a statement). There are (currently) two kinds: *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, the *data* that populate memories and memory banks, and the *references* that populate tables and table banks.

Unlike statements and preambles, commands can be grouped on the same line, using a comma-separator.

Register Preambles

Register-preambles begin with the `@register` decorator, 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, when present:

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

Below are a couple of examples:

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

Multiple local 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. So, the last example could be shortened to this:

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

A single register-preamble can define different types of register too. 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-definition contains one or more *instructions*.

Instructions are commands. As such, they can be grouped with other instructions, using a comma-separator. This includes the block-instructions (`block`, `loop`, `branch` and `else`), though block-instructions (obviously) end with a block of instructions, so naturally, nothing can be grouped after a block-instruction (only before one).

To avoid the issues that CoffeeScript had with cryptic whitespace, the assembler 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 thus branch of type 0
    call $foo
else thus 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*.

PHANTASM uses the term *datatype* to describe the types that can be written to, or read from, linear memory, that must be extended or truncated (to a proper numtype) in the process of moving them to or from the stack.

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

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

The `as` keyword, when present, prefixes the datatype (when it differs from the given numtype). The `in` keyword, when present, prefixes the identity of the memory (supporting multiple memories). The `at` keyword, when present, prefixes the offset of the load or store into memory. For example:

```
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 expresses whether a given instruction operates on signed, unsigned or agnostic integers using corresponding signed, unsigned and agnostic integer-type names.

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 that are copied directly from WAT:

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

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

```
ceiling     | ceil
lop         | trunc
root        | sqrt
sign        | copysign
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 similar to the grammar for arithmetic and logic instructions, but appends a `left` or `right` qualifier 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 an optional identity, prefixed by the `in` keyword when present (`in` prefixes both memory and table indices):

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

The type can be expressed with a type-reference or type-expression, and the identity specifies the table, defaulting to zero. 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 appends an identity (for the target block) to the mnemonic:

```
mnemonic identity
```

The exit-instruction requires one or more (space-separated) identities (with 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 [identifier] [of type] block
```

When present, the identifier is bound to the implied label, and (just like functions) the type (when present) can be expressed with a type-reference or a type-expression. For example:

```
loop $loop of type 2

    ; instructions...

block of i32, i32 → void

    ; instructions...

branch $branch of type $check

    ; instructions...

else

    ; instructions...
```

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

Note: PHANTASM does not require whitelines anywhere in the grammar, but you are recommended to use (exactly) one before and after each block header, block, directive and docstring, as well as between chunks of related statements, directives and commands (as in the example above). 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, but the component type (`memory` or `table`) follows the mnemonic, with an optional identity for the memory or table:

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

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 (and therefore, the identity is required):

```
mnemonic component-reference identity
```

For example:

```
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 optional identity (prefixed by the `to` keyword when present) is used to identify the *destination* (the component to copy *to*). It defaults to zero. 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 table 1
copy memory to 5
copy table $opcodes
```

Here are a few more examples of copy instructions:

```
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 a very similar grammar for its version of the WAT `trunc_sat` instructions, which retain the `lop` mnemonic, but replace the `to` keyword with `sop`. 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` keyword 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`, `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 initializer, known as a *memory-primer*, to the grammar for memory-specifiers:

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

When omitted, the primer is implicitly empty. When present, the primer is a block (which can be inlined), and must contain one or more *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 (grammatically, just a block of instructions) that defines the offset.

The block can be indented or inlined. For example:

```
@segment thus push #100
```

Segment-preambles also support a shorthand using the following grammar:

```
@segment number-literal
```

The sugar synthesizes a push-instruction, using the number literal as its immediate, so for example, the following code is equivalent to the previous example:

```
@segment #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. For example:

```
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 thus 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 thus i8 #10, #20, i16 #30, #40
```

Table Definitions

Table-definitions append an optional initializer, known as a *table-primer*, to the grammar for table-specifiers:

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

The various constructs of the above grammar have been described already (in the Table Specifiers and Memory Definitions sections).

The qualifier is one of `pointer`, `proxy` or `mixed`.

The primer contains *reference-commands* (instead of the data-commands that memories use), and (currently) table-definitions can only 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 table-commands will be added soon. The value is a reference to a function that can be expressed as an 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 pagewise), 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 thus 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 `mixed`):

```
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-statements. For example:

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

Type Definitions

Type-definitions define function types. They use the inline grammar for component definitions, where an inline initializer is prefixed by the `as` keyword (instead of using a block):

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

Naturally, type-definitions are used by define-statements. 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 (using a type-expression within a larger statement or instruction).

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.

Type-expressions that express existing types are replaced with references to the existing type, avoiding implicit duplication.

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 relevant discussions.