

PHANTASM: The Abstract Grammar Cheatsheet

This cheatsheet summarizes the PHANTASM abstract grammar in loose BNF-style pseudocode. The token grammar is not expanded. Instead, the following names are used to represent the various token types:

- **Number**: A number literal token.
- **String**: A string literal token.
- **Reftype**: One of **pointer** or **proxy**.
- **Valtype**: One of the valtypes (a **Numtype** or a **Reftype**).
- **Primitive**: Any primitive type (including pseudo-types when specified in a comment).
- **Identifier**: A user-assigned identifier (mapped to an index within an indexspace).
- **Identity**: A **Number** or **Identifier** (used to identify a component within its indexspace).

The first two pages of this document cover statements, while the last two pages cover instructions. Comments are used to explain things that would otherwise complicate the abstract grammar:

```
module : statement* ; statements are always onside, each on its own line

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

component-definition : function-definition ; every type of component can be defined ...
                    | register-definition
                    | memory-definition
                    | table-definition
                    | memory-bank-definition
                    | table-bank-definition
                    | type-definition

component-specifier : function-specifier ; only system components can be specified ...
                   | register-specifier
                   | memory-specifier
                   | table-specifier

component-reference : function-reference ; every type of component can be referenced ...
                   | register-reference
                   | memory-reference
                   | table-reference
                   | memory-bank-reference
                   | table-bank-reference
                   | type-reference

function-definition : function-specifier block
register-definition : register-specifier [initializer]
memory-definition : memory-specifier memory-segment*
table-definition : table-specifier table-segment*
memory-bank-definition : memory bank Identity encoding+
table-bank-definition : table bank Identity reference+
type-definition : type as type-expression

function-reference : function Identity
register-reference : register Identity
memory-reference : memory [Identity] ; optional identities always default to zero ...
table-reference : table [Identity]
memory-bank-reference : memory bank Identity
table-bank-reference : table bank Identity
type-reference : type Identity
```

```

function-specifier : function of type
                    | Identity of type
                    | function of signature ; signatures are only valid in function definitions ...
                    | Identity of signature

type : type-expression
     | type-reference

register-specifier : constant Valtype [Identity]
                  | variable Valtype [Identity]

memory-specifier : [shared] memory [Identity] limits

table-specifier : table-qualifier table [Identity] limits

table-qualifier : pointer
                | proxy
                | mixed

limits : with Number to Number
       | with Number max
       | with Number ; not valid with shared memories

memory-segment : segment-directive encoding+ ; must be indented or inlined...
               | encoding+ ; initial segments can omit the segment directive

table-segment : segment-directive reference+ ; must be indented or inlined...
              | reference+ ; initial segments can omit the segment directive

type-expression : types → types

signature : params → types

types : Valtype,+
      | void

params : param,+
       | void

param : Valtype [Identifier] ; sequences can be compounded

block : local-directive* instruction+ ; must be indented or inlined

constant-expression : instruction+ ; must be indented or inlined

local-directive : local Valtype [Identifier] ; sequences can be compounded

initializer : constant-expression
            | as Number ; with numtype registers
            | as Identity ; with pointer registers

segment-directive : segment constant-expression ; must be indented or inlined
                  | segment at Number

encoding : i8 Number ; sequences can be compounded...
         | i16 Number
         | i32 Number
         | i64 Number
         | utf8 String

reference : pointer Identity ; sequences can be compounded...
          | pointer null

```

```

instruction : unreachable
nop
return
drop
get [scope] Identity          ; scope defaults to current scope
set [scope] Identity          ; scope defaults to current scope
put Identity
add Primitive
sub Primitive
mul Primitive
div Primitive                  ; float or gnostic integer
rem Primitive                  ; gnostic integer
and Primitive                  ; integer
or Primitive                   ; integer
xor Primitive                  ; integer
clz Primitive                  ; integer
ctz Primitive                  ; integer
abs Primitive                  ; float
neg Primitive                  ; float
min Primitive                  ; float
max Primitive                  ; float
floor Primitive                ; float
nearest Primitive              ; float
rotate Primitive direction    ; integer
call Identity
invoke type [via Identity]
jump Identity
fork Identity
exit Identity+
block [Identifier] block-type block ; block must be indented
loop [Identifier] block-type block  ; block must be indented
branch [Identifier] block-type block ; block must be indented
else block ; block must be indented, following a branch-block
select [Valtype]
grow writeable-type [Identity]
size writeable-type [Identity]
fill writeable-type [Identity]
drop writeable-type Identity
copy readable-type [to Identity]
is positive-test
not negative-test
wrap i64 to i32
convert Primitive to Primitive    ; gnostic integer → float
promote f32 to f64
demote f64 to f32
cast Primitive to Primitive       ; float → integer or vice versa
lop Primitive to Primitive        ; float → gnostic integer
lop Primitive sop Primitive       ; float → gnostic integer
extend Primitive to i64           ; 32-bit gnostic integer → i64
expand Primitive to Primitive     ; signed integer → larger integer
load Primitive [as datatype] [in Identity] [at Number] ; in memory, at offset ...
store Primitive [as datatype] [in Identity] [at Number]
atomic load Primitive [as datatype] [in Identity] [at Number]
atomic store Primitive [as datatype] [in Identity] [at Number]
atomic add Primitive [as datatype] [in Identity] [at Number]
atomic sub Primitive [as datatype] [in Identity] [at Number]
atomic and Primitive [as datatype] [in Identity] [at Number]
atomic or Primitive [as datatype] [in Identity] [at Number]
atomic xor Primitive [as datatype] [in Identity] [at Number]
atomic swap Primitive [as datatype] [in Identity] [at Number]
atomic broker Primitive [as datatype] [in Identity] [at Number]
atomic wait Primitive [in Identity] [at Number]
atomic notify [in Identity] [at Number]
atomic fence
shift-instruction
push-instruction

```

scope : local | global | table

direction : left | right

datatype	:	i8		i16		i32
		s8		s16		s32
		u8		u16		u32

writeable-type : memory | table

readable-type : writeable-type | memory bank | table bank

negative-test : more | less | equal

positive-test : negative-test | zero | null

block-type : of type
| with Reftype ; `block with i32 ...` is short for `block of i32 → void ...`

shift-instruction	:	shift	Primitive	left		; integer
		shift	Primitive	right		; gnostic integer

push-instruction	:	push	[Numtype]	Number		; numtype defaults to i32
		push	pointer	Identity		
		push	Reftype	null		