

Plof Language Specification

This specification is not yet complete.
It is available in Plof's mercurial repository, at <https://codu.org/plof/hg/>.

Revision 42c310dc4da7 tip

03 September 2010

Contents

1	Overview	2
2	Plof Stack Language	3
2.1	Stack	3
2.2	Objects	3
2.2.1	Raw Data	4
2.2.2	Procedures	4
2.2.3	Exceptions	4
2.2.4	Arrays	4
2.3	Call stack	4
2.4	Operations	4
2.4.1	Bignums	10
2.4.2	File Format	10
2.4.3	ASCII PSL	11
3	Plof Runtime Parser	12
4	Plof User Language	13
4.1	Language	13
4.1.1	Grammar	14
4.2	base.apsl	14
4.3	Old PUL specification	15
4.4	PUL ABI	15
4.4.1	Indirection Objects	16

4.4.2	Helper Procedures	16
4.4.3	Calling Convention	16
4.5	Language	16
4.5.1	Grammar	16
4.5.2	Basic Syntax	19
4.5.3	Variables	19
4.5.4	Objects	19
4.5.5	Operator Overloading	19
4.5.6	Object Syntax	21
4.5.7	Multiple Inheritance	21
4.5.8	Field Ownership	21
4.5.9	Prototypes	21
4.5.10	Functions	22
4.5.11	Function Syntax	22
4.5.12	Parameters	22
4.5.13	Provided Objects	22
A	Primitive Types	23
B	Unsupported Operations and Trap	24
C	Version Specifiers	25
D	C Native Function Interface	26

Chapter 1

Overview

Plof is composed of a user-visible language which is parsed and compiled dynamically to a bytecode. An implementation of Plof must only support the bytecode and the flexible parsing framework; the user language is built entirely from Plof code.

The virtual machine runs a simple bytecode stack language, called the Plof Stack Language, or PSL, and code in the user language (the Plof User Language or simply Plof) is compiled using a runtime-defined grammar into PSL code. This specification will cover the details of PSL first. PSL is intended to be low-level and highly generatable, but also analyzable and optimizable.

The intermediary layer between these two is the Plof Runtime Parser (PRP), which is a flexible parsing and compiling framework, allowing the grammar being parsed and output code to be changed dynamically, at runtime. This allows Plof to be an extremely flexible language, with new features added to the language proper by libraries.

Plof, or the Plof User Language (PUL), is a minimalistic prototype-based object-oriented general-purpose programming language. Its syntax is inspired by C, Java, Tcl, Haskell and JavaScript, but its semantics are unique, although resembling JavaScript in some ways. Although an implementor of Plof must understand this entire document, the average user of Plof needs no knowledge of PSL or the parser.

Chapter 2

Plof Stack Language

The Plof Stack Language is a simple bytecode. That is, every byte is a single operation, with few exceptions (*marker*, *immediate*, *code*, and *raw*) which will be covered below. PSL features dynamic objects and first-class procedures.

2.1 Stack

The PSL stack is initially empty. Trying to pop from an empty stack is a runtime error. The stack may contain any object, integer, or other representable Plof type.

2.2 Objects

Everything is an object in PSL, including both the current execution context and procedures. PSL's object system is fairly simple. Objects are all by-reference, empty objects can be created with the *new* operation, and objects can be combined with the *combine* operation. When combining objects, object members of the right-hand object override object members of the left-hand object. Object members can be accessed with the *member* and *memberset* operations, and their names are completely arbitrary: They can be anything representable as a string of bytes. The object *null* is used as a marker in various situations, but it is otherwise just a normal object. All objects have a parent setting, which usually refers to the context in which the object was created, and can also be *null*. *this* is the name for the object representing the current context of execution at all times, and *global* is a globally-accessible object meant for the (highly-discouraged) use of global data when necessary. *null* is the parent of both *global* and *null* itself.

PSL implementations should garbage collect. They may use whatever form of garbage collector is suitable; this specification does not cover the details of the GC.

2.2.1 Raw Data

PSL can hold raw binary data in objects, to support simple and intrinsic data types. There are no limitations on the data that can be stored in a raw data object.

2.2.2 Procedures

Since PSL is bytecode, PSL instructions are representable within a PSL raw data object. Raw data objects are callable, to allow for procedures. The *call* operation runs a procedure in a new context; the exact details are discussed later.

2.2.3 Exceptions

PSL supports very simple exceptions. Any object can be thrown with the *throw* operation, and caught with the *catch* operation. Because PSL is typeless, *catch* will catch any object thrown. It is up to the programmer to determine whether the caught object can be handled properly.

2.2.4 Arrays

Object in PSL which do not contain raw data can be arrays. The result of using raw data as an array or an array as raw data is undefined. Arrays are solely for the sake of efficiency, as anything that can be done with arrays can be done with objects. Arrays have a *length*, which can be set, and elements within the array default to *null*. Elements can be accessed with *index* and set with *indexset*. The lowest valid index is 0, and the highest valid index is *length* - 1.

2.3 Call stack

Every procedure call at runtime has its own stack and context object. The stack initially contains infinite *nulls* and the object passed as an argument when the procedure was called. When the procedure completes, the object on the top of its stack is returned, and the rest of its stack is discarded.

2.4 Operations

This is the list of all PSL operations. The column “Stack behavior” indicates how many elements are popped from the stack and how many elements are pushed (in that order). In various operations, the function is described such as “pops A and B.” This means that A and B will be on the stack in that order; that is, A was pushed first, then B.

Hex	Name	Stack behavior	Function
-----	------	----------------	----------

00-07	push0-push7	0 → 1	push n pushes the n th element of the stack onto the top of the stack. e.g., for the stack: original stack: a, b, c after push0: a, b, c, c after push2: a, b, c, a																														
08	pop	1 → 0	Pop an object from the stack.																														
09	this	0 → 1	Push the context object.																														
0A	null	0 → 1	Push the null object.																														
0B	global	0 → 1	Push the global object.																														
0C	new	0 → 1	Pushes a new object. The new object has no members set and its parent is set to the current context.																														
0D	combine	2 → 1	Combine two objects. Pops the two objects to be combined, and pushes the new, combined objects. All of the members of both source objects are set in the new object, preferring the second source object when conflicts occur. The parent of the new object is the parent of the second object. The raw or array data is combined like so: <table border="1" data-bbox="669 850 1123 1180"> <thead> <tr> <th>Left</th> <th>Right</th> <th>Result</th> </tr> </thead> <tbody> <tr> <td>none</td> <td>none</td> <td>none</td> </tr> <tr> <td>raw</td> <td>none</td> <td>left raw</td> </tr> <tr> <td>none</td> <td>raw</td> <td>right raw</td> </tr> <tr> <td>raw</td> <td>raw</td> <td>concatenated raw</td> </tr> <tr> <td>array</td> <td>none</td> <td>left array dup</td> </tr> <tr> <td>none</td> <td>array</td> <td>right array dup</td> </tr> <tr> <td>array</td> <td>array</td> <td>concatenated array</td> </tr> <tr> <td>raw</td> <td>array</td> <td>left raw</td> </tr> <tr> <td>array</td> <td>raw</td> <td>right raw</td> </tr> </tbody> </table>	Left	Right	Result	none	none	none	raw	none	left raw	none	raw	right raw	raw	raw	concatenated raw	array	none	left array dup	none	array	right array dup	array	array	concatenated array	raw	array	left raw	array	raw	right raw
Left	Right	Result																															
none	none	none																															
raw	none	left raw																															
none	raw	right raw																															
raw	raw	concatenated raw																															
array	none	left array dup																															
none	array	right array dup																															
array	array	concatenated array																															
raw	array	left raw																															
array	raw	right raw																															
0E	member	2 → 1	Get a member from an object. Pops the object then the name of the member as raw data from the stack, and pushes the member object.																														
0F	memberset	3 → 0	Set a member in an object. Pops the object, the name of the member and the new member object from the stack.																														
10	parent	1 → 1	Get the parent of an object. Pops the object, pushes its parent.																														
11	parentset	2 → 0	Set the parent of an object. Pops two objects, sets the parent of the first to the second.																														
12	call	2 → 1	Call a procedure. Pops an argument object and the object containing the procedure. The procedure is called with a new stack containing only the argument object that was popped. A new context object is created for the procedure, the parent of which is set to the parent of the procedure object. The context contains at least one member, +procedure, the value of which is the procedure object. When the procedure finishes, the top element from its stack (or <i>null</i> if its stack is empty) is pushed onto the calling stack, and the procedure's stack is discarded.																														
14	throw	(special)	Throw an object. Pops the object to be thrown. Everything on the stack which was pushed by contexts which are unwound is discarded.																														

15	catch	3 → 1	Catch an object. Pops two procedures. The first procedure is called. If an exception is thrown, the stack of the called procedure is discarded, and the second procedure is called with the thrown object as an argument.
16	cmp	5 → 1	Compare two objects and branch based on their equality. Pops an argument object, two objects to compare and two procedures. If the two objects are referentially equal, the first procedure is called, otherwise the second procedure is called.
17	concat	2 → 1	Concatenate the raw data in two objects. Pops the two objects, and pushes a new object. The raw data of the two objects are concatenated into the new object. The new object does not have any members, and its parent is set to the current context object.
18	wrap	2 → 1	Wrap raw data in an operation to push it, outputting a raw data object containing PSL code. Pops two raw data objects, the first with the data to be wrapped and the second with the one-byte PSL instruction it should be wrapped in, and pushes a raw data object.
19	resolve	2 → 2	Find an object with a given member. Pops an object and a name or array of names. Given only a single name (as raw data), finds the nearest parent of the object with the given name as a member with a value other than <i>null</i> . Given multiple names (as an array of raw data objects), finds the nearest parent of the object with any of the given names. Pushes the found object and found name. If no object/name combo is found, pushes two <i>nulls</i> .
1A	while	3 → 1	Run a section of code repeatedly. Pops an argument object and two procedures. The first procedure represents the condition, the second procedure represents the code to be repeated. The loop is run like so: <ul style="list-style-type: none"> • The condition is run, with <i>null</i> as an argument. • If the condition returned <i>null</i>, the code is not run and the argument object is returned to the stack. • If the condition returned any other value, the code is run with the argument object as an argument, and its return replaces the argument object. The loop then repeats from running the condition, as above.
1B	calli	1 → 0	Run the immediates in a procedure. Pops an object containing the procedure. The code contained in any <i>immediate</i> instructions will be run (with <i>null</i> as the argument).

1C	replace	2 → 1	Replace markers with raw data in raw data. Pops a raw data object and an array of raw data objects. In the first argument, markers associated with the index of each element of the second argument are replaced with the data in the same index. That is, a marker with the (one-byte) value 0 will be replaced with the first element in the array. This is done with PSL code in mind: That is, if a marker is replaced within a <i>code</i> operation, its length will be updated correctly. Markers are not replaced within <i>raw</i> operations. The parent of the pushed object is the parent of the first raw data operand.
20	array	> 0 → 1	Creates an array from elements on the stack. The top element on the stack should be an integer (see the <i>integer</i> operation), the length of the array. The length is popped, and the appropriate number of elements are popped below it, and added to the array. They will appear in the array in the order that they appear in on the stack, which is of course the order in which they were pushed. The parent of the array is set to the current context object.
21	aconcat	2 → 1	Identical to <i>concat</i> , but for arrays.
22	length	1 → 1	Get the length of an array. Pops the array object and pushes its length.
23	lengthset	2 → 0	Set the length of an array. Pops the array object and an integer for the new length, and sets the length of the array, padding with <i>nulls</i> as necessary.
24	index	2 → 1	Get an element out of an array. Pops the array object and the index as an integer, and pushes the element.
25	indexset	3 → 0	Set an element in an array. Pops the array object, the index as an integer and the object to be assigned, and assigns it as appropriate. If the array is not long enough, it is expanded.
26	members	1 → 1	Pushes an array (see <i>array</i>) containing the names for all of the members of an object. Pops the object, pushes the array. The parent of the array is the current context, as is the parent of each raw data object in the array.
2A	nullarray	1 → 1	Pushes an array (see <i>array</i>) of <i>nulls</i> . Pops the length as an integer, pushes an array object of that size with each element set to <i>null</i> .
60	rawlength	1 → 1	Get the length of raw data. Pops a raw data object, pushes an integer of its length.
61	slice	3 → 1	Slice raw data. Pops a raw data object and two integers, the start and the end. Pushes a raw data object, the data of which is the content of the original raw data, starting at the start byte requested (0-indexed), including the bytes up to but not including the end byte.
62	rawcmp	2 → 1	Compare two raw data objects. Pops two raw data objects and pushes an integer. This operation corresponds to the C function <i>memcmp</i> , so the integer will be 0 if the two data are equal, -1 if the first is less than the second, and 1 if the first is greater than the second, as defined by <i>memcmp</i> .

63	extractraw	1 → 1	Given a raw data object, push a new object with the same raw data. The new object will have the same raw data, but no properties. Its parent is set to the context. If the original object has no raw data, this operation is equivalent to <i>new</i> .
70	integer	1 → 1	Converts raw data into the most efficient integer format of the host. The raw data can be an 8-, 16-, 32- or 64-bit, big-endian integer. Pops the raw general integer and pushes a host-specific integer. The properties and restrictions of integers in PSL are defined in appendix A. The parent of the new integer, if defined, is the same as the parent of the original data. The size of the integer is left intentionally undefined.
71	intwidth	0 → 1	Push an integer containing the width of host integers, in bits.
72	mul	2 → 1	Integer multiply.
73	div	2 → 1	Integer divide. If the second number is 0, pushes null.
74	mod	2 → 1	Integer modulo. If the second number is 0, pushes null.
76	add	2 → 1	Integer add.
77	sub	2 → 1	Integer subtract.
78	lt	5 → 1	Pops an argument object, two integers and two procedures. If the first integer is less than the second integer, the first procedure is called, otherwise the second procedure is called.
79	lte	5 → 1	Less than or equal to.
7A	eq	5 → 1	Equal.
7B	ne	5 → 1	Not equal.
7C	gt	5 → 1	Greater than.
7D	gte	5 → 1	Greater than or equal to.
7E	sl	2 → 1	Shift left.
7F	sr	2 → 1	Shift right (arithmetic)
80	or	2 → 1	Bitwise or.
81	nor	2 → 1	Bitwise nor.
82	xor	2 → 1	Bitwise xor.
83	nxor	2 → 1	Bitwise nxor.
84	and	2 → 1	Bitwise and.
85	nand	2 → 1	Bitwise nand.
8E	byte	1 → 1	Convert an integer modulo 256 to a single raw byte. Pops the integer and pushes the byte as raw data.
90	float	1 → 1	Converts an integer into the most efficient floating point form of the host. Pops an integer, pushes a float. Like integers, floating point numbers are only defined over their own range (91-AF), so they can be implemented by whatever means is most efficient.
91	fint	1 → 1	Floor a floating point number into an integer.
92	fmul	2 → 1	Floating-point multiplication.
93	fdiv	2 → 1	Floating-point division. If the second number is 0, pushes null.
94	fmod	2 → 1	Floating-point modulo. If the second number is 0, pushes null.
96	fadd	2 → 1	Floating-point addition.

97	fsub	2 → 1	Floating-point subtraction.
98	flt	5 → 1	Floating-point less than (see <i>lt</i>).
99	fite	5 → 1	Floating-point less than or equal to.
9A	feq	5 → 1	Floating-point equal.
9B	fne	5 → 1	Floating-point not equal.
9C	fgt	5 → 1	Floating-point greater than.
9D	fgte	5 → 1	Floating-point greater than or equal to.
C0	version	0 → 1	Pushes an array of strings (as raw data), each of which is a specifier for the host environment. Exactly what is contained is open-ended, but in general it will include the architecture, kernel, standard library and native interface. Some examples are provided in appendix C.
			The operations C1-CF are reserved for native interfacing. They are not defined in this specification, since they can potentially be different for different implementations. Instead, they are defined by auxiliary specifications. Appendix D describes one design, CNFI.
			The operations D0-DF are useful for debugging. If debugging is enabled, they will be generated automatically by the runtime parser and used by the PSL interpreter, and so should not generally need to be used in writing PSL code.
D0	dsrfile	1 → 0	Set the source file of the current PSL code to the provided raw data. Pops a raw data object.
D1	dsrcline	1 → 0	Set the source line of the current PSL code to the provided value. Pops an integer.
D2	dsrccol	1 → 0	Set the source column of the current PSL code to the provided value. Pops an integer.
ED	trap	2 → 1	See appendix B.
EE	include	1 → 1	Pops the name of a file, and pushes the content of that file. If the file was not found or for any reason cannot be read, <i>null</i> is pushed. The directories in which the file will be searched for are defined by the implementation.
			The operations EF-FD affect the runtime parser. The details of the runtime parser are discussed in a later section.
EF	parse	3 → 1	Parse Plof code or a PSL file with the runtime parser. Pops the raw data of the Plof code or PSL file content, the name of the top symbol in the runtime parser and the name of the file that the code represents (for generating debugging data), and pushes raw data of the resultant PSL code. If the data is a PSL file matching the PSL specification in this document, the contained PSL code will be extracted and returned as a procedure directly, otherwise it is assumed to be Plof code and parsed with the runtime parser.
F0	gadd	3 → 0	Add a production to the grammar. Pops the production name, the production, and the associated PSL code.
F1	grem	1 → 0	Remove a production from the grammar. Pops the production name and removes it.

FB	gcommit	0 → 0	Commit grammar changes.
FC	marker	0 → 0	Serves no purpose but to be replaced with <i>the replace</i> operation above. Contains a marker symbol provided like raw data.
FD	immediate	0 → 0	The provided code, which is provided like raw data (see below), is run in the compilation phase, not the runtime phase. When the PSL code is provided in PSL form, this code is run before anything else. When the PSL code is generated by the runtime parser (which is discussed below), this code is run as soon as it is generated. The operations which affect the runtime parser (F0-FD) cannot be run outside of an <i>immediate</i> operation.
FE	code	0 → 1	Push raw data which is code. Works identically to <i>raw</i> (below), but is intended for PSL code. This allows the <i>replace</i> operation to work, and also can help optimizations.
FF	raw	0 → 1	Push raw data. The operation itself is immediately followed by a bignum number (the format of which is described below), which represents the number of bytes of raw data. The data itself follows immediately after the length. The raw data is pushed in an otherwise-empty object, the parent of which is the current context object.

2.4.1 Bignums

The *raw* operation requires a bignum, an integer of arbitrary size. The bignum can be any number of bytes, and the most significant bit of every byte signifies whether another byte follows. If the most significant bit of a byte is 1, another byte follows, and the most significant bit is 0 for the last byte. The remaining bits form the number, in big endian format. For example, the number 123,456 (b11110001001000000) is:

```
10000111 11000100 01000000
```

The maximum size supported by an implementation of PSL is intentionally undefined, but must be at least $2^{28} - 1$ (28 bits + 4 marker bits = 32 bits = 4 bytes).

2.4.2 File Format

Although it is not generally necessary to create .psl files, the file format is defined. A .psl file starts with an 8-byte magic number. In hex, the magic number is:

```
9E50534C17F2588C
```

After the magic number, any number of sections follow. Each section contains:

1. A bignum representing the length of the section, not including itself but including the next bignum (2).
2. A bignum of the numerical type of the section. Four section types are defined: 0 is raw program data (there can only be one program data section in a .psl file), 1 is comment data, 2 is stripped program data and 3 is a raw data table. If a section of type 2 is included, a section of type 3 must also be included. Stripped program data consists of the raw PSL data, with the raw data for the *raw* command replaced by bignum indexes into

the raw data table. For example, the PSL [FF 01 95] would be replaced by [FF 01 00], and [95] placed at the first index into the raw data table. All types greater than or equal to 128 (all types requiring more than one byte to describe) may be defined by users, types lower than 128 should only be defined by this specification, and future versions of it.

3. The data contained in the section.

2.4.3 ASCII PSL

For bootstrapping and debugging purposes, PSL has a defined ASCII syntax, compilable into bytecode. ASCII PSL is very simple: every alphanumeric sequence is a command (which must correspond to a name as given above), and the following grouping operations are supported:

Opening symbol	Closing symbol	Operation	Notes
{	}	code	
[]	immediate	
"	"	raw	Must support the escape character \ by simply removing it and including the following character unchanged.

Numbers in ASCII PSL should produce a *raw* command which pushes the specified number as 32-bit big-endian, then the *integer* operation.

Chapter 3

Plof Runtime Parser

Plof code is parsed and compiled by productions created at runtime. A production is a name, an array of targets, and PSL code to run when the production is parsed. The name corresponds to the relevant nonterminal in the grammar. Targets may be nonterminals (productions), negations, or regular expressions. The PSL code should produce a raw data object, itself containing PSL code (the compiled result of the production), and is run with the results of each of the targets in an array as the argument.

Target types (nonterminals, negations or regular expressions) are distinguished by names. Regular expression targets begin and end with a `'/'` character. Negation targets begin with a `'!'` character.

Multiple productions may be created with the same name, which indicates that the nonterminal corresponds to a choice of any of the productions.

The following example productions correspond to a simple arithmetic language:

```
mul = mul /\*/ add => { push0 0 index push1 2 index concat {mul} concat }
mul = add => { push0 0 index }
add = add /\+/ digit => { push0 0 index push1 2 index concat {add} concat }
add = digit => { push0 0 index }
```

The parser must support self-referencing left-recursive productions such as `A = A /b/`, but not more complex left-recursive productions such as `A = B /b/ ; B = A /a/`.

Regular expression targets will parse greedily. If they contain any parenthesized parts, they will consume only up to the end of the first parenthesized part, otherwise the entire matching substring is consumed. The result of a regular expression, returned to the production using it, is PSL code to produce the consumed string as a raw data object. Because the parser's terminals are regular expressions, no tokenization step is necessary in parsing.

The Plof interpreter parses code by repeatedly applying the production named “top” to the remaining input code. As such, the “top” production should parse one top-level statement or declaration of the target language. The interpreter's grammar should contain no productions at startup, so precompiled PSL code is required to initialize the parser.

Chapter 4

Plof User Language

The Plof User Language (called simply Plof, or PUL to differentiate it from the rest of the Plof environment) is the language that programmers most commonly see, and is implemented as a runtime grammar in the Plof Runtime Parser, compiling to PSL code.

The reference implementation of PUL is comprised of 26 files, which are loaded and will be discussed in the following order (top to bottom, left to right):

- base.apsl
- pul_g.plof
- pul.plof
- object_g.plof
- object.plof
- include_g.plof
- exceptions.plof
- boolean_g.plof
- boolean.plof
- comparisons_g.plof
- comparisons.plof
- conditionals.plof
- dynamicTypes_g.plof
- dynamicTypes.plof
- number_g.plof
- number.plof
- string_g.plof
- string.plof
- exception_strings.plof
- modules_g.plof
- modules.plof
- collection_g.plof
- collection.plof
- io.plof
- builtins.plof
- nfi.plof

Neither these exact filenames nor this order of loading is necessary for a correct implementation of PUL, but the resultant grammar and objects visible to the user must be created as specified.

4.1 Language

PUL is an extremely dynamic impure functional language with strong support for imperative programming. Alternatively, it is a dynamic imperative object-oriented programming language with first-class functions and types,

and strong support for functional programming. The language is lazy. The laziness, however, is diminished by the presence of statements and eager assignment.

The object system of PUL is prototype-based, but with additional properties allowing it to be used with annotations. This will be discussed in a later section.

4.1.1 Grammar

The grammar of PUL is built incrementally. Many of the files loaded as part of PUL expand upon the grammar. In this document the grammar will be given in a syntax similar to Backus Normal Form, with a few caveats:

- As described in the previous section, PRP does not include a tokenizer. The terminals are regular expressions.
- Standard regular expression terminals are signified by slashes: `/regular-expression/`
- PUL builds syntax for defining the grammar. This syntax includes terminals which are automatically parsed as if they were tokens; that is, they are followed by whitespace, which is ignored. These tokens are signified by quotes: `"token"`. Do not be fooled, the content of the quotes is still handled as a regular expression. The whitespace is handled by the production `"nnlwhite"`, documented as part of `base.apsl`.
- In most circumstances, newlines are *not* handled as whitespace, but are in fact equivalent to semicolons (they end a statement). Many tokens are marked such that newlines following them *will* be handled as whitespace (by the production `"white"`). These tokens are also signified by quotes, but ending with the character `'n'`: `"token"n`.
- The syntax `A => B` is shorthand for `A_next = B`. This style is used to allow other productions to be added into the grammar by code loaded later. If `A => C` is encountered after `A => B` (for any A, B, C), then C will be inserted between A and B: `A_next = C, C_next = B`. Note that this is a truly imperative statement, with a side-effect to the parsing of PUL: The grammar is in fact changing as files are loaded, and definitions may be removed and replaced.

Whitespace elements in PUL are space, tab, carriage return and C-style and C++-style comments, as well as a backspace followed by a newline, to allow for multi-line statements. The `"white"` production also includes newlines as whitespace.

4.2 base.apsl

This is the only component of PUL which is not itself written in PUL (it is an ASCII PSL file). Its purpose is to create some very basic statements for defining the grammar and using PSL. The grammar-defining statements it defines are used only in `pul_g.psl` to create a higher-level mechanism. `base.apsl` loads the following grammar:

```
top = white base                                base = gcommit /;/ white
base = gadd /;/ white                            gadd = /__grammar_add/ white token white /\(/
base = grem /;/ white                            white /[a-zA-Z0-9_]+/ white /,/ white
```

```

grammarElems /,/ white psl0ps /\)/ white
grem = /__grammar_rem/ white token white /\(/
white /[a-zA-Z0-9_]+/ white /\)/ white

gcommit = /__grammar_commit/ token white /\(/
white /\)/ white

grammarElems = grammarElems grammarElem
grammarElems = grammarElem

grammarElem = /[a-zA-Z0-9_]+/ white
grammarElem = /(\([^\\\/\]*\.\)*[^\\/\]*\)/
white

psl0ps = psl0psX
psl0ps = white

psl0psX = psl0psX white psl0p white
psl0psX = psl0p white

psl0p = /\{/ white psl0ps /\}/ white
psl0p = /\{/ white /\}/ white
psl0p = /\[/ white psl0ps /\]/ white
psl0p = /\[/ white /\]/ white
psl0p = /\x22/ /^[^x22]*/ /\x22/ white
psl0p = number white

psl0p = marker white
psl0p = /aconcat/ token white
psl0p = /add/ token white
...
psl0p = /cwrap/ token white
psl0p = /swrap/ token white
psl0p = /iwrap/ token white

token = /([\^A-Za-z0-9_]|$)/

number = digits token

marker = /\$/ number

digits = digits digit
digits = digit

digit = /0/
digit = /1/
digit = /2/
digit = /3/
digit = /4/
digit = /5/
digit = /6/
digit = /7/
digit = /8/
digit = /9/

white = /((([ \t\r\n]*(#[^\r\n]*[\r\n])?(\\\/[^\r\n]*[\r\n])?(\\\/*(\[^\]*\*[^\\/])*\[^\]*\*\/)?)*)/

```

The "..." above represents the inclusion of every other PSL operator, as documented in the first section of this specification, in the same manner as *aconcat* and *add*.

4.3 Old PUL specification

The following sections are the original (incomplete) PUL specification. They are mostly correct, but not particularly useful for implementors or users. They are slowly being replaced by the above sections. For the moment, the following sections will remain, until their content has been entirely replaced by the new sections.

4.4 PUL ABI

Because PUL has some advanced features not directly supported in PSL, many fundamental PUL operations compile to several PSL operations or procedures, and some constructs are represented in abstract ways.

4.4.1 Indirection Objects

PUL is a lazy language. As such, values in PUL can be represented by special objects providing indirect access to a procedure to evaluate the value. These are called indirection objects. Indirection objects have three significant members: `__pul_e`, `__pul_s` and `__pul_v`.

`__pul_e` is the procedure to evaluate the value. It does not expect anything on the stack, and it pushes the value. It may return the ultimate value, or it may return another indirection object. For that reason, it shouldn't be used directly, but through the `__pul_eval` procedure (which will be discussed later).

`__pul_s`, which is only set when applicable, is the procedure to set the value. It pops the new value, and pushes nothing. The value being assigned must be fully evaluated; that is, it must not be an indirection object.

`__pul_v` is the cached value if it has already been evaluated. If it has not been evaluated, `__pul_v` is unset.

4.4.2 Helper Procedures

There are several procedures in the global object of the Plof environment to aid with the use of indirection objects.

The global `__pul_eval` procedure expects an object on the stack, pops it, and pushes the fully-evaluated form. If the object is not an indirection object, it is left intact. Otherwise, the fully-evaluated form is pushed.

Finally, there are several global functions to automatically convert functions and values to indirect objects. They are documented in `pul.plof`.

4.4.3 Calling Convention

The arguments to a function are pushed onto the stack as an array. The function is expected to push its return value.

4.5 Language

PUL is an extremely dynamic impure functional language with support for the imperative paradigm. The language is lazy. The laziness, however, is diminished by the presence of statements and immediate assignment.

The object system of PUL is prototype-based, but with additional properties allowing it to be used with annotations. This will be discussed in a later section.

4.5.1 Grammar

The following is the grammar of PUL, roughly in BNF. The syntax `"regex"w` is used to represent that the regex will be parsed and expected to be followed by whitespace. Statements are terminated by a newline or semicolon. Whitespace elements in PUL are space, tab, carriage return and C-style and C++-style comments, as well as a backspace followed by a newline, to allow for multi-line statements.

The syntax `A => B` is shorthand for `A_next = B`. This style is used to allow other productions to be added into the grammar by code loaded later.

The following is the base grammar as created by `pul.g.plof` and `object.g.plof`; other extensions to the grammar added by the standard library will be covered in later sections.

```
top => plof_statement

plof_statement = plof_statement_next eos
plof_statement => plof_assign

plof_semicolon = plof_semicolon eos plof_semicolon_next
plof_semicolon = plof_semicolon eos
plof_semicolon = plof_semicolon_next
plof_semicolon => plof_assign

plof_assign = plof_assign_next "="w plof_assign
plof_assign = plof_assign_next
plof_assign => plof_bind

plof_bind = "let"w plof_bind_next "="w plof_bind
plof_bind = plof_bind_next
plof_bind => plof_group

plof_group = "forceEval"w "\"w plof_identifier "\"

plof_group = plof_group "\".w plof_identifier
plof_group = plof_group "\".w "parent"
plof_group = plof_group_next
plof_group => plof_parens

plof_parens = "\"("w plof_semicolon "\"
plof_parens = plof_parens_next
plof_parens => plof_literal

plof_literal = "psl"w "\"{"w psl0ps "\"
plof_literal = /\$/ number nnlwhite
plof_literal = plof_literal_next
plof_literal => plof_var

plof_var = "var"w plof_identifier
plof_var = plof_var_next
plof_var => plof_ident

plof_ident = plof_identification

plof_identification = plof_notkeyword /[A-Za-z_][A-Za-z0-9_]*/ token nnlwhite
```

```

ploff_notkeyword =
  /()(!as[~A-Za-z0-9_])/
  /()(!by[~A-Za-z0-9_])/
  /()(!forceEval[~A-Za-z0-9_])/
  /()(!is[~A-Za-z0-9_])/
  /()(!in[~A-Za-z0-9_])/
  /()(!include[~A-Za-z0-9_])/
  /()(!parent[~A-Za-z0-9_])/
  /()(!return[~A-Za-z0-9_])/
  /()(!rtInclude[~A-Za-z0-9_])/
  /()(!to[~A-Za-z0-9_])/
  /()(!var[~A-Za-z0-9_])/

ploff_group = ploff_group ploff_group_next

ploff_group = ploff_group "\"(w ploff_args \""

ploff_args = white

ploff_args = ploff_args_prime

ploff_args_prime = ploff_args_prime ",w ploff_args_prime_next

ploff_args_prime = ploff_args_prime_next ",w ploff_args_prime_next

ploff_args_prime => ploff_arg

ploff_arg = ploff_semicolon

ploff_group = "\"(w ploff_params \"" "{w ploff_funcbody \""

ploff_params = white

ploff_params = ploff_params ",w ploff_params_next

ploff_params = ploff_params_next

ploff_params => ploff_param

ploff_param = ploff_identifier

ploff_funcbody = white
ploff_funcbody = ploff_funcbody_next
ploff_funcbody => ploff_funcbody_prime

```

```
plof_funcbody_prime = plof_semicolon

plof_group = "return"w plof_group_next

plof_group = "\{"w plof_funcbody "\}"

plof_obj_def = "this"w plof_bind_next

plof_literal = "ref"w plof_var
```

4.5.2 Basic Syntax

PUL's statements and expressions resemble those in most C-style languages. The precedence of all of the operators is described in the above grammar. Statements are implemented in a functional style; that is, semicolon-separated statements really just form an expression, and evaluation of that expression will result in evaluation of the left and right sides in order, resulting in the value of the right side.

Because PUL is lazy, several constructs have special meanings to allow eager evaluation. Statements are always evaluated eagerly, although the resultant value is not used. Note however that this does *not* mean that every value used in the statement is evaluated. Whatever is necessary to attain a final value of the expression is evaluated, and that final value is discarded. The right-hand side of an assignment expression is also evaluated eagerly. All other cases are evaluated lazily, including notably arguments to functions.

4.5.3 Variables

Variables may be declared within any context with the `var` keyword. When first defined, a variable will have the value *null*, so a `var` expression may be used as an l-value or an r-value. Variables may not be used without being declared. Variable assignment uses the `'='` operator as in C-family languages, but `'='` returns *null*, so chains like `a = b = c` do not function as they do in most C-family languages.

4.5.4 Objects

PUL's objects are always derived from other objects, with the exception of the object named `Object`, which is the top of the object hierarchy and is defined in PSL. As well as being the top of the hierarchy, `Object` defines all of the basic operators. All operators on `Object` throw exceptions (described in a later section), with the exception of `'=='` and `'!=='`, which simply compare object references if not overloaded.

4.5.5 Operator Overloading

Most operators in Plof are defined to call functions on their objects (the only exceptions are `'.'`, `'='`, `'>'` and `'<'`), and so can be overloaded with user code. The following table describes operators and the equivalent function call.

Operator	Function call
<code>x y</code>	<code>x.opOr(y)</code>
<code>x && y</code>	<code>x.opAnd(y)</code>
<code>x == y</code>	<code>x.opEqual(y)</code>
<code>x != y</code>	<code>x.opNotEqual(y)</code>
<code>x < y</code>	<code>x.opLess(y)</code>
<code>x <= y</code>	<code>x.opLessEqual(y)</code>
<code>x > y</code>	<code>x.opGreater(y)</code>
<code>x >= y</code>	<code>x.opGreaterEqual(y)</code>
<code>x + y</code>	<code>x.opAdd(y)</code>
<code>x - y</code>	<code>x.opSub(y)</code>
<code>x * y</code>	<code>x.opMul(y)</code>
<code>x / y</code>	<code>x.opDiv(y)</code>
<code>x % y</code>	<code>x.opMod(y)</code>
<code>!x</code>	<code>x.opNot()</code>
<code>x as y</code>	<code>x.opAs(y)</code>
<code>x is y</code>	<code>x.opIs(y)</code>
<code>x in y</code>	<code>y.opContains(x)</code> (note the reversed direction)
<code>x[y]</code>	<code>x.opIndex(y)</code>

Several operators have default implementations which use auxiliary functions, to allow users to implement many operators with only one function. These default implementations are all part of `Object`, and the auxiliary functions throw exceptions as with all default operator definitions. The equivalences are:

Operator	Equivalence
<code>x < y</code>	<code>x.opCmp(y) < 0</code>
<code>x <= y</code>	<code>x.opCmp(y) <= 0</code>
<code>x > y</code>	<code>x.opCmp(y) > 0</code>
<code>x >= y</code>	<code>x.opCmp(y) >= 0</code>
<code>x as y</code>	Multiple steps: <ol style="list-style-type: none"> 1. If <code>x</code> is derived from <code>y</code> (determined by <code>opIs</code>), returns <code>x</code>. 2. <code>x.opCastTo(y)</code> 3. if <code>opCastTo</code> returned <code>null</code>, <code>y.opCastFrom(x)</code> 4. if <code>opCastFrom</code> returned <code>null</code>, <code>null</code>

All operator functions are defined in `Object`, so all objects have defined operators, but most of them are defined in `Object` simply to throw an exception (the exception mechanism will be discussed later).

`opEqual` and `opNotEqual` check object identity in `Object`, but should be overloaded in any code with a well-defined equality operation. `opIs` implements type-checking, which will be described further in the next section.

4.5.6 Object Syntax

Objects are defined by the syntax `Super : [...]`, where 'Super' is the parent type (which must always be specified, even if it is Object), and '...' is the content of the object. Any fields given in the object definition will override those defined by the parent. Fields are specified by semicolon-separated assignment statements, e.g.:

```
var O = Object : [  
  x = 3  
  y = 4  
]
```

Fields may be functions (the syntax of which is defined later), creating methods, but methods are in no other way different from other fields. Field definitions may refer to previous fields but not later ones, as the fields will be evaluated in the order that they appear.

Every object contains a special field, `--pul_type`, which contains an array of objects which this object is derived from, then itself. The ordering of `--pul_type` is well-defined, as an object definition will always simply append one element to the `--pul_type` of the parent. `opIs` implements type-checking by simply comparing its parameter with every element in `--pul_type`. If any match, the type-check succeeds.

4.5.7 Multiple Inheritance

As well as defining new objects, it is also possible to combine existing objects. The syntax is the same, with the exception that `[...]` is replaced by the second object. Important to note is that operators defined by Object will usually not be handled well by multiple inheritance: If for example the left object implements `opAnd` but not `opAdd`, and the right object implements `opAdd` but not `opAnd`, the resultant object will implement `opAdd` but not `opAnd`, as it will inherit the right-hand object's default `opAnd`.

4.5.8 Field Ownership

Every object has a parent object, as defined by PSL. Fields of objects may point to children of the object, or any other object. These cases are treated differently when objects are combined. If a field contains a child object, that object will be duplicated in the process of duplicating the parent object, and the resultant child object's parent will be the resultant parent object (the result of combination). If a field contains a non-child object (an object with a parent of any other object), it is not duplicated, but the appropriate field in the new object is simply set as another reference to the same object.

4.5.9 Prototypes

PUL has no inherent way to distinguish prototype objects (objects intended only to be derived from into other objects) and instance objects (objects meant to be used in program logic). However, to make the distinction convenient for users, a `new` function is provided by the core library. `new` takes an object assumed to be a prototype, duplicates it (`o : []`), and calls an optional 'init' function on the resultant object.

4.5.10 Functions

Functions in PUL are first-class, and are in fact objects (the `Function` object is derived from `Object`, and all functions are derived from `Function`). The fundamental PSL procedure associated with a function is simply the raw data that the object contains, so calling a function is, at the PSL level, just a primitive *call*. The context of functions is the parent of the function object, which results generally in lexical scoping (functions declared in a scope will have that scope as their parent), but is modifiable to allow any form of dynamic scoping.

4.5.11 Function Syntax

In the simplest form, functions in PUL are in the form `{ . . . }`, with “...” being the expression defining the function. All functions are defined in this way, there is no distinction between lambda functions and named functions. That is, a named function is just an anonymous function assigned to a variable:

```
var evalToOne = { 1 }
```

4.5.12 Parameters

Functions may have parameters, specified by a comma-separated list of variable names all within parenthesis (‘(’ and ‘)’) preceding the function:

```
var addOne = (x) { x + 1 }
```

Parameters may be type-checked by using ‘as’:

```
var addOne = (x as Integer) { x + 1 }
```

This is implemented with `opAs`, and an exception is thrown if `opAs` fails.

4.5.13 Provided Objects

The PUL runtime environment provides a large number of builtin objects, serving as prototypes for classes such as `Integer` and `Boolean`. The runtime environment and provided objects will be described in a later version of this specification, as they have not yet been completely defined.

Appendix A

Primitive Types

To allow for a fast implementation, primitive data (integers and floats) may not be represented as an object in the same way as all other PSL objects. As such, most of the properties of primitive types are undefined, with the exceptions being those cases in which the spec explicitly states that integers or floats are used.

Primitive data does not need to have a parent or support for members, nor does it need to be usable as raw data. Primitive data must be able to be a member of a true object. Because primitive data may not have a parent or members, the *parent* and *member* operations may return *null*, the *parentset* and *memberset* operations may do nothing, and the result of the *members*, *combine* and *resolve* operations are undefined when they would need to reference the members of primitive data.

Appendix B

Unsupported Operations and Trap

A compliant PSL implementation may choose not to implement the following operations:

*parse*¹, *gadd*, *grem*, *gcommit*

On such interpreters, these instructions will be trapped.

The purpose of the *trap* instruction is to allow these more complicated parts of PSL to be implemented in PSL or Plof. It pops a raw data argument and a procedure. The first argument is one byte long, the instruction to be trapped. The second argument will be called the emulation procedure in this appendix.

From the point of the *trap* instruction on, the instruction in the first argument *may* be trapped. It will only be trapped if it is unsupported by the PSL interpreter. When the instruction is encountered by the PSL interpreter, if it is unsupported by the interpreter, it will call the emulation procedure instead of failing. The argument to the emulation procedure is an array, containing the arguments to the instruction (as defined in this specification). If the instruction pushes a result, the return from the procedure is that result; if not, the return from the procedure is ignored. When the emulating procedure finishes, execution continues normally.

An implementation that fully supports all of the listed instructions will implement *trap* as a no-op. As such, the user must assume that the emulation procedure may not actually be used. It is an error to call any unsupported operation before an associated *trap* instruction.

¹All conforming implementations must implement *parse* when the input is a PSL file. If the input is not a PSL file, they may call an emulation procedure.

Appendix C

Version Specifiers

Implementations should use whatever version specifiers are appropriate. This is merely a list of suggestions.

Architecture specifiers: ARM, MIPS, PowerPC, x86, x86_64

Kernel specifiers: Darwin, FreeBSD, HURD, Linux, NetBSD, OpenBSD, Solaris, Windows9x, WindowsNT

Standard library specifiers: BSD, glibc, Mac OS X, Windows

Appendix D

C Native Function Interface

Implementations written in C or languages compatible with C may provide the interface described in this appendix. If it is provided, the version specifier CNFI should be active. This interface is based on libffi, as provided with the GNU Compiler Collection, but may be implementable with other systems.

To call a function with the Plof CNFI, you need its address, and you need a call interface specification, or 'cif', which is defined at runtime. The address of a function can be found using an interface similar to POSIX's dlfen.h.

CNFI also includes several new types of data which may be contained in Plof objects: C types and cifs. Exactly how they are implemented is intentionally unspecified, and user code must not rely on their organization. CNFI also includes pointers, but pointers are merely raw data of the appropriate size.

The operations that CNFI provides:

Hex	Name	Stack behavior	Function
C1	dlopen	1 → 1	Open a shared library. Pops a raw data object containing the name of the shared library and pushes a shared library handle, which is a pointer. If an error occurs, <i>null</i> is pushed instead of a handle.
C2	dlclose	1 → 0	Close a shared library. Pops a shared library handle, does not report errors.
C3	dlsym	2 → 1	Resolves a symbol, optionally in a specific shared library. Pops a shared library handle and a raw data object containing the name of the symbol. The handle may be <i>null</i> , in which case the symbol will be searched for in any loaded library.
C4	cget	2 → 1	Get the given amount of data from the given address. Pops a pointer and an integer number of bytes, pushes the data.
C5	cset	2 → 0	Put the given data at the given address. Pops the address and the raw data, pushes nothing. Note that it is not necessary to push a size specifier, as the raw data object has a length implicitly.
C6	cinteger	1 → 1	Given an integer as raw data directly from C, push a Plof integer. Pops raw data, pushes an integer. Equivalent to <i>integer</i> on big-endian systems.

C7	ctype	1 → 1	Creates a representation of a basic C type. Pops an integer, pushes the type. The correlation:
C8	cstruct	1 → 1	Create an aggregate (struct) type. Pops an array of types, pushes the resultant type.
C9	sizeof	1 → 1	Get the size in bytes of the provided type. Pops a type and pushes an integer.
CA	csget	3 → 1	Struct-get. Pops a struct type (created by cstruct), a structure (as raw data) and an integer component index, and pushes the element.
CB	csset	4 → 0	Struct-set. Pops a struct type, a structure, an integer index and a new value, and pushes the newly-modified struct.
CC	prepcif	3 → 1	Pops a return type, an array of argument types (which may be 0-length) and an integer ABI specifier (which may be 0 to use the default ABI, and is otherwise implementation-specific), and pushes a cif.
CD	ccall	3 → 1	Pops a cif, a pointer and an array of arguments, and pushes a return value. The arguments must all be raw data of the correct size, and the return will also be raw data.