Open Pluggable Edge Services
Internet-Draft
Expires: April 26, 2004

A. Beck
Lucent Technologies
A. Rousskov
The Measurement Factory
October 27, 2003

P: Message Processing Language
draft-ietf-opes-rules-p-02

Abstract

P is a simple configuration language designed for specification of message processing instructions at application proxies. P can be used to instruct an intermediary how to manipulate the application message being proxied. Such instructions are needed in an Open Pluggable Edge Services (OPES) context.

Table of Contents

1. Introduction

The Open Pluggable Edge Services (OPES) architecture [I-D.ietf-opes-architecture], enables cooperative application services (OPES services) between a data provider, a data consumer, and zero or more OPES processors. The application services under consideration analyze and possibly transform application-level messages exchanged between the data provider and the data consumer. OPES processors need to be told what services are to be applied to what application messages. P language can be used for this configuration task.

In other words, P language primary objective is to express statements similar to:

```
             if message meets criteria C,
             then apply service S;
```

Figure 1

Thus, P programs mostly deal with formulating message-dependent conditions and executing services.

P design attempts to satisfy several conflicting goals:

flexibility: Application intermediaries deal with a wide range of applications and protocols (SMTP, HTTP, RTSP, IM, etc.). The language must be able to accommodate virtually all known tasks in selecting a desired adaptation service for a message of a known application protocol (and conceivable future applications).

efficiency: Language interpretation must be efficient enough to be comparable with other message processing overheads at a typical application proxy (e.g., interpreting HTTP headers to determine response cachability).

simplicity: Typical configurations must be easy to write and understand for a typical OPES system administrator.

correctness: Many message handling configurations are written without direct access to intermediaries that will use those configurations. The extent of off-line (compile-time) correctness checks should catch all syntax errors and many common semantic errors such as undefined values and type conflicts.

compactness: It is possible that some processing instructions will be piggybacked as headers/metadata to messages they refer to, placing stringent size requirements on language code.

security: It should be difficult if not impossible to write malicious code that would result in security vulnerability of compliant language interpreter.

While P addresses OPES needs, its design is meant to be applicable for a variety of similar intermediary configuration tasks such as access control list (ACL) specification and message routing in proxy meshes or load-balancing environments.

P design is based on a minimal useful subset of features from several programming languages such as R (S), Smalltalk, and C++. Technically speaking, P is a single-assignment, lazy evaluation, strongly typed functional programming language.

2. Syntax

P syntax is defined by the following Augmented Backus-Naur Form (ABNF) [RFC2234]:

```
code = *statement

statement =
     expression-statement /
     assignment-statement /
     compound-statement /
     if-statement /
     comment /
     ";"

if-statement = if-head *if-alt [if-tail]
if-head      = "if" "(" expression ")" "{" code "}"
if-alt       = "elsif" "(" expression ")" "{" code "}"
if-tail      = "else" "{" code "}"

compound-statement = "{" code "}"

assignment = identifier ":=" expression ";"

expression-statement = expression ";"

expression =
     constant-expression /
     name /
     function-call /
     "(" expression ")" /
     "{" code "}" /
     unary-op expression /
     expression binary-op expression

constant-expression = boolean / number / string

name = identifier *( "." identifier)

function-call = name "(" [call-parameters] ")"

call-parameters = expression *( "," expression)

identifier = (ALPHA / "_") *(ALPHA / DIGIT / "_")

unary-op =
     "+" / "-" /
     identifier
```

```
   binary-op =
        "==" / "!=" /
        "<" / ">" / ">=" / "<=" /
        "+" / "-" / "*" / "/" / "%" /
        identifier

   comment = "/*" OCTET "*/"             ; no nesting allowed

   boolean = "true" / "false"

   number = 1*DIGIT ; no leading zeros

   string = DQUOTE *string-char DQUOTE

   string-char =
        %x00-21 / %x23-5B / %x5D-FF / ; any but quote and backslash
        escape-sequence               ; C++ or XML escape sequence? XXX
```

Figure 2

3. Language elements

3.1 Objects

P is centered around the concept of an "object" that is similar to objects from other object-oriented languages. An object is, essentially, a piece of data or information. The value of an object is indistinguishable from the object itself. Object type is defined by the semantics of applicable operations and manipulations. Almost everything in P is an object, even a piece of code. Here are a few P objects, listed one per line:

```
0
"http://www.ietf.org/"
Core
{ a := 1/0; }
```

Many objects contain other objects, often called members. Members are accessible by their name, using the member access operator ("."). Member access operator has a single parameter: the name of the member to access. All P objects support "." operator, but not all objects have members. Here are a few examples:

```
Http.message.headers
Core.interpreter.stop
"string".nosuchmember
```

Many objects support operators other than member access. For example, member objects that support function call "()" operator are often call methods.

```
Http.message.headers.have(header)
Core.interpreter.stop()
1 / 0
"string" + "string"
```

P operators are described in Section 3.3. below.

P does not have built-in facilities for describing object types. When writing a P program, only objects known to interpreter (e.g., Core) and objects generated by known objects (e.g., Http.message.headers) can be used. P supports loadable modules that can be used to add objects to support new application protocols. In fact, P core supports no application protocols directly. Instead, modules like "HTTP" can be used to process messages depending on application protocol being proxied.

3.2 Type conversion

Interpreters MUST NOT silently convert (cast) object types. When explicit conversion (casting) is needed objects should provide polymorphic methods (methods with the same name but different formal parameter types).

3.3 Operators

Operators are used in P to denote common operations on built-in object types and language constructs. No operators are defined for objects provided by modules. P operators do not modify their operands. Note that all operators may result a failure.

Unary Operators

operator	operand type	result type	semantics
+	number	number	returns operand
-	number	number	returns zero minus operand
import	string	module	imports module members into global namespace and returns a module object that can be used to access this module members explicitly; operand is module identifier, a URI; fails on any error
not	boolean	boolean	logical negation
try	code	boolean	interpret operand and return true or fail; try is the only operator defined for code; try never returns false

Binary Operators

operator	operands type	result type	semantics
==	boolean or integer	boolean	simple value equality
!=	boolean or number	boolean	simple value inequality
<	number	boolean	less than; ">", "<=" and ">=" are defined similarly
equal	string	boolean	left string equals right string
contains	string	boolean	left contains right
begins_with	string	boolean	left string begins with the right string
ends_with	string	boolean	left string ends with the right string
and	boolean	boolean	short-circuited logical conjunction: the right expression is evaluated only if the left expression is true
or	boolean	boolean	short-circuited logical disjunction: the right expression is evaluated only if the left expression is false
xor	boolean	boolean	exclusive logical conjunction; cannot be short-circuited: both operands are evaluated
otherwise	statement	any	short-circuited failure detection: the right expression is evaluated only if the left expression

			fails; returns the value of the last expression evaluated
-	number	number	arithmetic difference
+	number	number	arithmetic sum
+	string	string	concatenation
*	number	number	arithmetic product
/	number	number	arithmetic ratio; rounded to the closest integer (XXX?)
%	number	number	arithmetic modulo
.	name	object member	object member access; fails if the object produced by expression on the right does not have the member named by the expression on the left.

A function call is an n-ary operator. Besides the function name, it takes zero or more actual parameters as operands.

All string operators described above are case-sensitive and come with the corresponding case-insensitive operators: EqualS, ContainS, Begins_witH, and Ends_witH (XXX: bad idea to name using case?) (XXX: should we force programmers to pick the right variant instead of providing efficient, but usually wrong default: contains_s and contains_i?)

Operator precedence defines natural evaluation order used in mathematics and many programming languages. In the following list, operators are ordered based on their precedence. Operators with smaller precedence index are evaluated first. Operators with the same precedence index are evaluated in the left-to-right order of occurrence in an expression.

1. .

2. ()

3. not

4. * /

5. + -

6. all binary operators on string: equals, contains, ...

7. import

8. == != < <= > >=

9. and

10. or

11. xor (XXX: misplaced?)

12. try (XXX: misplaced?)

13. otherwise

3.4 Expressions

P expressions are used in if-statements to specify the condition for the if-statement body to be interpreted.

```
if (Http.request.method == "GET" and time.current() > time.noon) {
        ...
}
```

Figure 6

Evaluation of an expression stops when the value of an expression is known and cannot be changed by further evaluation. This short-circuiting optimization technique is common to many programming languages. In the following example, the value of A will never be interpreted when C is interpreted, regardless of the context where C is used:

```
C := false and A;
if (C) { ... };
if (!C) { ... };
...
```

Figure 7

3.5 Statements

Objects are manipulated using if-statements and function-calls.

```
      if (Http.request.method == "GET") {
              Services.applyOne(serviceFoo);
      }
```

Figure 8

3.6 Assignments

Most procedural programming languages use variables to store intermediate processing results. In such languages, a variable is essentially a named piece of memory that can be assigned a value and can be updated with new values as needed. P does not have such variables. Instead, P uses a "single assignment" approach: an expression can be tagged with a name and that name can be reused many times in the program. On the surface, this is equivalent to having all "traditional" variables declared as "constant". The following two if-statements are semantically equivalent in P:

```
      if (Http.request.headers.have(Http.makeHeader("Client-IP"))) {...}

      h := Http.makeHeader("Client-IP");
      hs := Http.request.headers();
      if (hs.have(h)) {...}
```

Figure 9

If the expression changes, a new name must be used to tag the new expression. After an assignment statement, the value of the name is not the value of the expression, but the expression itself. Thus, the following two code fragments are equivalent and make no sense in P (the first fragment would make sense in languages such as C++):

```
      h := Http.makeHeader("Client-IP");
      h := Http.makeHeader("Server-IP");

      h := Http.makeHeader("Client-IP");
      Http.makeHeader("Client-IP") := Http.makeHeader("Server-IP");
```

Figure 10

The interpreter can but does not have to evaluate the expression named in the assignment statement until the name is actually used in an expression that requires evaluation (e.g., as a parameter of a function call statement). This allows for optional performance optimizations where only used expressions are evaluated.

P does not have user-defined functions. However, some code reuse is

possible because P code is a valid expression and, hence, can be named and reused:

```
     code := { ... complicated service action ... };
     if (condition1) { code; };
     ...
     if (condition2) { code; };
```

Figure 11

XXX: document whether expression has to be evaluated in the assignment context or use context.

Names introduced using assignments have global scope. Global scope makes it possible to select among alternative values without user-defined functions or true variables:

```
     if (condition) {
             /* no "service" name exists at this point */
             service := Services.findOne(uri1);
     } else {
             /* no "service" name exists at this point */
             service := Services.findOne(uri2);
             service.authorization(myAuth);
     }
     Services.applyOne(service); /* service name is still visible */
```

Figure 12

4. Modules

Application-specific support is available in P via modules. Module is an object. Interpreters MUST supply two modules named Core and Services. The Core module contains members for manipulating built-in P object types such as integers and strings. The Services module manages OPES services. Application specific modules can be loaded into the namespace of a P program via the import operator (see Section 3.3). For example, the following P code imports an HTTP module, names the result (the module itself) "Http", and checks for the presence of a certain HTTP message header:

```
Http := import "http://ietf.org/opes/rules/p/HTTP";
if (Http.message.headers.have("Accept")) { ... }
```

Figure 13

It is not possible to import a Core or Services module explicitly. Instead, interpreters MUST provide access to Core and Services members as if those modules were imported just before the program text.

Modules are identified by their URIs [RFC2396]. A module specification SHOULD contain a globally unique URI for that module. Module URIs are usually not used to fetch module implementation remotely, but to identify a suitable local copy of a module; they are identifiers, not locators. Interpreters maintain a directory of known-to-them module URIs. When a module needs to be imported, the interpreter checks internal metadata and loads the requested module using module-specific interface. If the module is not known or loading fails, the import operator fails and the failure is propagated using standard failure propagation rules (see Section 6). The following example attempts to import one of the SMTP modules.

```
/* load one of the available SMTP modules */
Smtp := import "http://ietf.org/opes/rules/p/SMTP" otherwise
        import "http://examle.org/opes/optimized/SMTPv3";
```

Figure 14

Import operation has program scope. It is not possible to "unload" an imported module.

```
{
        M := import "http://ietf.org/opes/rules/p/HTTP";
        ...
}
/* M and M members are still visible here */
```

```
        if (M.connection.is_persistent()) { ... }
```

Figure 15

4.1 Interpreter-module interface

Most modules are not written in P since the language lacks native mechanisms for defining module or function interface. Most modules are tightly integrated with OPES processors because application adaptation requires access to processor's internal state. For example, an HTTP intermediary implemented in C++ can use modules written in C++ and may require that implementors inherit their modules from a given C++ class. Such modules may be loaded using, for example, a "dynamically loadable module" mechanism supported by most modern operating systems. Similarly, a Java OPES processor may require that all modules implement a given Java interface and use Java importing mechanism. This specification does not document any specific interface between an interpreter and third-party modules.

Nevertheless, an interpreter MAY support loading of modules written in P (similar to C++ #include directives). The interface for distinguishing URIs of P programs from integrated modules is implementation-dependent and is not described here. For example, an interpreter may assume that all unknown module URIs correspond to raw P programs and attempt to include such a program if the URI scheme is known to the interpreter:

```
        MyLibrary := import "file://usr/local/lib/globalrules.p";
```

Figure 16

4.2 Modules and namespace

Members of imported modules belong to the global namespace and are directly accessible (visible) without the module name prefix. This simple rule may lead to conflicts when two imported modules contain a member with the same name. Interpreters MUST fail if any name resolution is ambiguous. Interpreters MUST NOT use heuristics to guess programmer's intent. Programmers have to use fully qualified names to resolve ambiguities.

For example, all of the import statements below pollute global name space, but the first two provide a way for a programmer to resolve conflicts, if any:

```
/* import HTTP module */
Http := import "http://ietf.org/opes/rules/p/HTTP";

/* import SMTP module */
Smtp := import "http://ietf.org/opes/rules/p/SMTP";

/* import a local file without naming it */
import "file:///usr/local/globalrules.p";
```

Figure 17

In the following example, both the Http and Smtp modules have the same member named "message", and the code leads to an ambiguity, even though Smtp module's message does not have a "method" member:

```
Smtp := import "http://ietf.org/opes/rules/p/SMTP";
Http := import "http://ietf.org/opes/rules/p/HTTP";

method1 := message.method;      /* error: HTTP or SMTP "message"? */
method2 := Http.message.method; /* OK: HTTP "message" */
```

Figure 18

5. OPES Services

Services module contains basic attributes and methods for searching and executing OPES services:

Services.findOne(URI): returns a service object that corresponds to the specified URI. Fails if no corresponding object exists.

Services.applyOne(service, ...): applies the specified service to the current application message and optionally supplies service-specific application parameters. XXX: should parameters include the part of the message to be modified or just services metadata?

Here is a service application example for a German to French translation service:

```
     Http := import("Http");
     if (Http.response.language_is("german")) {
             service := Services.findOne("opes://svs/tran/german/french");
             service.toDialect("southern");
             Services.applyOne(service, Http.request.headers);
     }
```

Figure 19

XXX: explain how failures are propagated and can be handled

XXX: add Core.interpreter.stop and Core.interpreter.restart methods.

6. Failures

Virtually any P statement may fail: expression denominator may be zero, named members may not exist, functions may not support supplied parameters, service execution may fail, interpreter may ran out of resources during an assignment, etc. A failure immediately stops interpretation of the expression that caused it.

Failure is propagated up the expression and statement stack until the stack is empty or an "otherwise" alternative is reached (see Section 3.3). If the stack is empty, the entire P program interpretation terminates with a failure. If an "otherwise" alternative is encountered, the failure is forgotten and interpretation resumes with that alternative.

Failure propagation rules allow to catch failures, similar to an exception mechanisms in languages like C++ or Java, except that P exceptions are not objects (they carry no information). For example, here is a simple way to introduce a backup/failover service:

```
{
        ...
        Services.applyOne(unsafeService);
} otherwise {
        ...
        Services.applyOne(failoverService);
};
```

Figure 20

The "otherwise" operator makes it simple to select among failure-prone alternatives:

```
service := findOne(uri1) otherwise findOne(uri2);
```

Figure 21

The following example illustrates how a failure-prone service can be retried twice if needed:

```
code := {
        /* code executing the service */
};
try code otherwise try code otherwise try code;
```

Figure 22

It is possible to force the interpreter to fail using the

"Core.interpreter.fail(reason)" call. This is handy when there is a logical failure that the interpreter cannot detect on its own:

```
     {
             /* large piece of code executing several services,
                each manipulating the current HTTP message ... */

             /* checkpoint */
             if (!Http.message.headers.have("Content-Length")) {
                     Core.interpreter.fail("services did not set CL");
             }

             /* OK, continue message manipulation ... */
     } otherwise {
             /* recover from failure ... */
     }
```

Figure 23

This specification has no failure reporting requirements. The extent and form of failure reporting depends on the environment: Developer environments would benefit from extensive and detailed reporting of failures. Stand-alone intermediaries processing P instructions may benefit from some reporting, appropriately implemented not to bring down the proxy due to high volume of failures. User environments, especially mobile and similarly resource-constraint applications should probably conserve scarce resources and produce no reports by default.

7. Security Considerations

XXX: document non-obvious vulnerabilities: too many names, too deep nesting, invalid math, too much error logging; execution of unauthorized services, unauthorized exposure of sensitive information to authorized services.

8. Compliance

XXX: define what a compliant interpreter is.

Appendix A. Examples

This appendix contains half-baked examples to illustrate P usage in common OPES environments. Example themes are taken from [I-D.beck-opes-irml] to ease the comparison with IRML.

Here is a data provider example:

```
     interpreter.languageVersion("1.0"); // fails if incompatible

     Http := import("Http");
     lookup(Http);

     // Is the requested web document our home page?
     isHome := request.uri.looksLikeHome();

     // Does the user send us a specific cookie?
     cookie := makeHeader("Cookie", "sew=23");
     haveCookie := request.headers.have(cookie);

     if (isHome and haveCookie) {
             Services := import("Services");
             service := Services.findOne("opes://local.net/add-lcl-
content");
             service.clientIp(request.clientIp);
             Services.applyOne(service);
     }
```

Figure 24

Here is a data consumer example:

```
     Services := import("Services");
     service := Services.findOne("opes://privacy.net/priv-serv");
     service.action("remove-referer");
     Services.applyOne(service);
```

Figure 25

Appendix B. To-do

i18n: What are IETF and real-world internationalization requirements for languages? Can we say that everything is Unicode UTF-8 and be done with it? Does UTF have a notion of space characters like ASCII does? If not, how can we separate grammar tokens without requiring them to be ASCII?

namespaces: Module lookup facility leads to potential conflicts among identical names from different modules. What is the best way to resolve these conflicts? How other languages do it?

security: Write Security Considerations section. A lot can be moved from the IRML security section. Some can be borrowed from OCP Core.

module URI: Is there an IETF document that tells us how to assign/ manage URIs for new "things" like modules? For example, do we use http://ietf.org/opes/http for HTTP module? Or do we use iana.org domain name instead? Is http:// a good choice for the scheme or should we use opes:// or even p://?!. Do we use de-facto file:// for local filenames from where raw P code can be included directly? Note that modules like HTTP are not written in P!

examples: Add more simple but realistic and illustrative examples: HTTP header anonymization, OPES/HTTP trace entry management (e.g., removing trace entries of a given OPES service), removing a virus attachment from an SMTP message. Ask filtering/ICAP people to supply use cases.

interpreter API: Document that we do not document interpreter API -- how, for example, an implemented HTTP module is actually "loaded". Mention that the solution would depend on the interpreter implementation and the same HTTP module is unlikely to be compatible with different interpreters.

define interpreter: Add terminology section. Define interpreter to mean compiler, or run-time interpreter, or bytecode generator, or anything of that kind.

op keywords: Document that operator names (via identifier BNF entry) are not keywords: object members can use identifiers that clash with operator names since there can be no ambiguity.

statement value: Document values of all statements (e.g., compound-statement value is the value of the last statement in a compound)?

RE: Decide whether we should support regular expression matching natively.

if-else-if: Make if-else-if syntax compact.

str ==: Remove "==" for strings in examples. There is no such operator for strings anymore.

Appendix C. Acknowledgments

The authors gratefully acknowledge contributions of: Anwar M. Haneef (Motorola) and Geetha Manjunath (Hewlett Packard).

Appendix D. Change Log

Internal WG revision control IDs: $RCSfile: rules-lang.xml,v $ $Revision: 1.23 $.

2003/10/08

* Added (expression) expression to BNF.

2003/09/22

* Added missing concatenation operator for strings.

2003/09/21

* Explained undocumented relationship between interpreters and third-party modules.

2003/09/19

* Simplified module importing and lookup facilities. Import is now a built-in operator and not a Core method. Explicit lookup control is gone in favor of always-lookup default.

2003/09/18

* Completed syntax BNF except for escape sequences.

* Distinguish interpretation failure from boolean false: use "otherwise" and "or" operators respectively. With just "or" it was impossible to say whether, say, "h.has(foo)" failed or "h" just does not have "foo".

* Use Perl semantics for "otherwise" -- return the value of last evaluated expression, not true/false.

* Nearly completed a set of supported operators, including operators for strings.

* Operators should only be supported for built-in objects because it is difficult to define how "5 + object" is interpreted without running into problems with "object + object" ("object + 5" is easy but we need symmetry). It is unlikely that we are losing much with this limitation anyway -- protocol objects would rarely have good semantics for operators.

* Defined scope rules for new names introduced by assignments.

* Added Acknowledgments section.

Normative References

[RFC2234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[RFC2396] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

[I-D.ietf-opes-architecture]
Barbir, A., "An Architecture for Open Pluggable Edge Services (OPES)", draft-ietf-opes-architecture-04 (work in progress), December 2002.

Informative References

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[I-D.beck-opes-irml]
Beck, A. and M. Hofmann, "IRML: A Rule Specification Language for Intermediary Services", draft-beck-opes-irml-03 (work in progress), June 2003.

Authors' Addresses

Andre Beck
Lucent Technologies
101 Crawfords Corner Rd.
Holmdel, NJ
US

Phone: +1 732 332-5983
EMail: abeck@bell-labs.com

Alex Rousskov
The Measurement Factory

EMail: rousskov@measurement-factory.com
URI: http://www.measurement-factory.com/

Intellectual Property Statement

Full Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.