Procedural Graph-store Processing

Eric Griffis
University of California, Los Angeles
egriffis@cs.ucla.edu

Abstract

Graph-store Processing (GSP) is a novel programming paradigm for managing, sharing, and computing on arbitrarily structured and possibly distributed collections of digital information. As an informal introduction to basic GSP concepts, I present *pGrasp*, an experimental language for specifying computations as procedures within the context of a *graph-store*—a directed property graph over variable contexts. In this report, I outline the need for a paradigm like GSP. I introduce basic GSP concepts and relate them to pGrasp constructs. I produce a series of pGrasp programs which culminates in a simple distributed task queue with interesting properties. I provide a formal specification for pGrasp, along with an overview of a prototype implementation. Finally, I discuss in detail the relationship between pGrasp and GSP, and describe the next steps toward a practical GSP platform.

1. Introduction

We have, generally speaking, many tools at our disposal for solving a given problem in software. Despite the fact that all Turing-complete programming languages are equally expressive, each language tends to excel in some aspects at the expense others. Datalog, a logic programming language that models computations as queries over sets of related terms, solves problems in deductive reasoning elegantly but has some trouble with evolving state. C, an imperative language, handles mutable state well but can be difficult to reason about. Indeed, a language perfect for every situation would obviate the need for all others.

Thus, a useful programming language inevitably fills a niche—a setting in which particular kinds of problems can be solved *naturally*, with minimal complication. This reasoning applies to programming paradigms as well. Object-oriented programming (OOP) decomposes large problems into independent components to maximize productivity. Symbolic programming treats code as data to maximize flexibility. Imperative programming operates close to the underlying hardware platform to maximize performance.

How about for Web information systems [6], or other forms of distributed information systems? What kinds of processes can we execute on loosely structure information repositories like the Web, and how should we structure such efforts? What are the common properties of distributed, ad-hoc information systems? Can we exploit these commonalities to reduce complexity in distributed systems? To summarize these issues, I pose the following question.

How can we most naturally manage, share, and compute on loosely structured digital information?

With respect to existing tools and methods, I see no clear answer to this question. For the Web, we might argue that the current trajectory of Web technologies is acceptable and will continue to improve with time. The evidence is, however, in direct opposition to this argument. The current state of Web software development is quite complex. Indeed, we must maintain an entire software "stack" of distinct products in order to enact a presence on the modern Web, and plans for the Semantic Web promise to inject an entirely new sub-stack into the existing model. Consider the volume of distinct configuration, description, and general programming languages involved in the following scenario.

To share information on the modern Web, we require at minimum a Web server and some HTML pages. To adjust elements of presentation, we use Cascading Stylesheets (CSS). We also use JavaScript for general computations like looping, interactive event handling, and back-end communications via AJAX libraries that transmit messages as JSON or XML documents. If we intend to offer proprietary or complex services, we integrate a server-side application development framework like PHP into our Web server. To manage structured data, we incorporate an SQL database or some form of NoSQL alternative. If we intend our service to become large, we might implement our applications on the SOAP messaging framework. If growth occurs unexpectedly, we might choose CORBA, an IDL-based middleware, instead.

No single technology in this stack is designed to simultaneously manage, share, and compute on a structured information repository. Moreover, each component presents a distinct perspective on the nature of the system as a whole. Disparate languages and priorities lead to duplicated effort and complicated work-arounds. Furthermore, the Web is but one form of distributed information system. We also have, for example, e-mail, video chat, and instant messaging systems, not to mention the plethora of emerging systems unique to mobile platforms. Each form poses different goals and challenges:

push versus pull, stream versus block, reliability versus performance, and so on.

Despite the profusion of concepts and languages throughout the stack, we can observe several common elements. Virtually all back-end processes communicate via tree structures, ranging in complexity from flat key-value headers to complete XML or JSON documents. Each process in the system encodes, decodes, consumes, transforms, and produces structured messages to affect its contribution to the overall behavior of the system. These messages represent structures local to each process, requests to act on these local structures, or responses to such requests.

There are clear opportunities to simplify the stack. Efficient message passing and processing are crucial elements of any distributed software system. Every layer of the stack stands to benefit from a natural setting in which to inspect and operate on messages directly, without spurious translations between representations. Furthermore, each implementation may depend on messaging internally, in the form of OOP method calls. To address this clear opportunity, I propose the graph-store.

1.1 Graph-store Processing

A graph-store is a graph-based data structure, designed specifically to manage, share, and compute on arbitrarily structured collections of information. The nodes of a graph-store are similar to C structs or OOP objects. A graph-store node differs from a struct in that it can be used directly as a context in which to evaluate a program. It differs from an object in that we can can choose dynamically in which node to evaluate each sub-program. In other words, a graph-store node is like an object with methods that affect the internal state of other objects.

This arrangement flagrantly violates the principle of encapsulation. As a principled approach to software construction under such conditions, I propose the Graph-store Processing (GSP) paradigm. GSP appeals to graph rewriting [10, 11] as a basis for structuring and analyzing large programs with formal methods similar to functional programming, but in total absence of closed-world, top-down notions of modularity such as objects or functions—static barriers through which information passes according to code producers. Instead, I adopt an open-world, bottom-up approach, leveraging concepts such as functional pattern matching [7] and multi-staged programming (MSP) [13]—tools for code consumers to inspect and affect sub-program behavior dynamically. GSP stands apart from previous work with graph rewriting [1, 5] in its use of staging as the primary means of abstraction and its emphasis on simple sharing constructs.

GSP strikes a balance between the symbolic and functional paradigms. Specifically, GSP comprises a novel combination of staged evaluation and syntactic pattern matching. The result is a complete inversion of the concept of modularity. In a modular system, we attach names to *code*—each component is a black box that distills arguments into return values. In a GSP system, we attach names to *data*—each component is an arbitrarily complex message, in the object-oriented sense [4], that describes a computation to be carried out though a series of rewrites.

In an imperative language like Java, the methods of an object describe *how* particular tasks are carried out as sequences of instructions that affect its internal state. Object boundaries insulate internal states so methods can not rely on transient external details or lead otherwise to inconsistencies. A GSP language specifies *what* a solution to a particular problem is in a declarative style. A GSP node consumes a sub-program, alters or acts on the parts that it understands, and passes the modified sub-program on as its result, possibly for further modification by other nodes—the sub-program *is* the argument, internal state, and return value.

This is not to say that modularity is impossible in a GSP system, just that modularity is extremely difficult to simulate within an individual graph-store and merely undesirable in general. In a distributed setting where local processes operate within distinct graph-stores, the graph-stores are de-facto modules, if only for the underlying physical boundaries. GSP defeats this limitation elegantly by prescribing a primitive node transmission mechanism.

My decision to abandon modularity might seem absurd at first, but the rewrite model of computation compensates amply by appealing to our extra-computational intuition; it is, for example, naturally concurrent. According to José Meseguer [8],

> The idea of concurrent rewriting is very simple. It is the idea of *equational simplification* that we are all familiar with from our secondary school days, *plus* the obvious remark that we can do many of those simplifications independently, i.e., in *parallel*.

This excerpt highlights the peculiar nature of rewriting, about which reasoning leads typically from a simple premise directly to a desirable property. This quality allows us to express and compose a broad range of behaviors naturally in a bottom-up fashion, as the following comparison illustrates.

To implement matrices in a modular language without native support, we write a function library that constructs and operates on native representations of matrix data. Before we can use this library, we must understand its API. If the language contains meta-programming features, we can simplify the notation with additional work. Given two distinct matrix libraries, we either choose one statically or produce a compatibility layer that encapsulates our dynamic selection logic.

In a modular setting, we build programs by selecting implementations.

To implement matrices in a GSP language, we specify a set of rules that rewrite operations over matrices in an intuitive notation into, say, matrices of operations over scalars. We use the library by applying its rules repeatedly and recursively until no further rewrites are possible. Given an alternative implementation with identical notation, we choose one statically or dynamically and have no need for a compatibility layer. If the alternative follows a different notation, our compatibility layer contains only rules to map between notations.

In a graph-store setting, we build programs by selecting notations.

This report proceeds as follows. Section 2 describes the pGrasp programming language and its core constructs. Section 3 introduces pGrasp programming via three example programming sessions. From core constructs, I implement lists and queues. From these structures, I implement a networked queue management API for community resource pooling. Section 4 specifies the syntax and semantics of the pGrasp language formally, followed by details in section 5 of the prototype implementation that informed the formal design process. Section 6 briefly discusses discrepancies between pGrasp and GSP. Section 7 concludes with a summary of possible next steps for the GSP project.

2. The pGrasp Language

pGrasp is a procedural programming language designed to highlight some core principles of GSP, but the pGrasp language offers neither staged evaluation nor functional pattern matching. It is a proof of concept which favors familiar constructs over innovative design. I work around its deficiencies with convenience constructs and conventions borrowed from low-level languages. The language, nevertheless, is useful and admits compact solutions to real problems.

A pGrasp program constructs and manages a *graph-store*, a labeled, directed property graph that exhibits index-free adjacency of nodes, where each node is a distinct variable name space. pGrasp provides a variety of features, many of which are familiar. A pGrasp program is a semicolon-delimited sequence of statements. We can store and recall variables by name. We have looping and branching. We can capture and apply sub-programs as procedures, and interact with external resources via opaque descriptors. I use common data types like booleans, numbers, and strings in a conventional syntax. For slightly more detail, see the formal syntax specification in section 4.1.

Program evalation progresses from left to right and top to bottom. I use parentheses (resp., braces) to denote explicit alternative groupings or orders of evaluation of expressions (resp., statements). Often, I group terms unnecessarily—especially for procedures, loops, and conditionals—to convey some intuition for constructs with similar behavior in other languages. Reserved words appear in a proportional sans-serif font, like this. Identifiers appear in a monospaced serif font, like `here`.

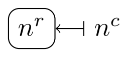
A variable, node reference, cursor (sec. 2.4), or value appears as the italicized letter x, n, c, or v, respectively, possibly with super- or sub-scripts.

2.1 Nodes and References

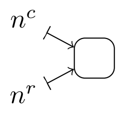
A graph-store instance contains a heap of nodes and a pair of special variables. A *heap* is an indexed set of nodes. A *node* is a dictionary, i.e., a set of mappings from variable names to values. To access a variable, we must obtain a reference to the node it inhabits. A *reference* is an opaque token that identifies a resource. We create a directed link between two nodes by setting a variable in the parent to a reference of the child. Occasionally, I conflate the notions of node, reference, and variable, assuming that the concept at hand is apparent.

A new pGrasp instance contains exactly one non-removable node called the *root*. Initially, both special variables reference the root. The first is immutable and therefore always references the root. The second references the *current node* and is always mutable. We can access the root and current node references with keywords root and this.

The following diagram shows the graph-store of a fresh pGrasp instance. In a graph-store diagram, each node appears as a box with rounded edges that contains its identifying reference at top, followed by any variables that do not contain a node reference. The special references n^r and n^c identify the root and current node, respectively. For each stored node reference, an edge labeled by the variable name points from parent to child.



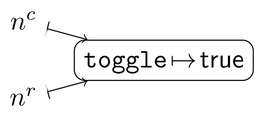
The next diagram shows the heap of a fresh pGrasp instance. In a heap diagram, node references appear on the left. The variables within a node appear to the right of the corresponding reference as a chain of name-value pairs. An empty box represents a node with no variables.



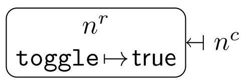
We have several constructs available for creating new variables, but the most direct construct is the set statement. To set the root variable `toggle` to the boolean value true, write:

```
set root toggle true;
```

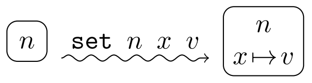
This statement grows the root node in the heap by one mapping.



Equivalently, the graph-store root gains a single non-reference mapping.



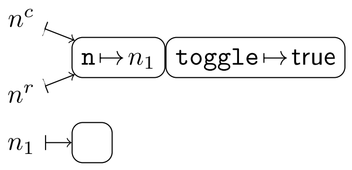
Graph-store diagrams are also useful for describing abstract processes as graph rewrite rules, as in the following illustration of the semantics for the set statement. For convenience, I assume that the left-hand side of a rule matches any node with at least the displayed elements, that the right-hand side of a rule replaces only the elements matched by the left-hand side, and that we apply rules repeatedly and recursively in any convenient order until we exhaust all possible matches.



We also have many ways to create nodes. The most basic, however, is the node statement. Because every non-root node must have a parent, all node construction statements require a parent node and a variable name in which to store a reference to the new node. Here is how we create a fresh node and store its reference in the root variable `n`.

```
node root n;
```

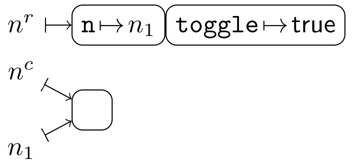
This statement adds a new node to the heap and grows the root node by one mapping.



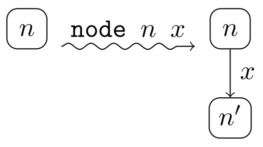
Now that we have a non-root node in the heap, we can set the current node directly with the in statement.

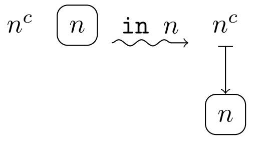
```
in n;
```

This copies the reference stored in root variable `n` into the current node special variable.



Here are the rewrite rules for the node and in statements.





We have two constructs for removing data. To delete a single variable, use the unset statement. If the deleted variable contains the sole copy of a node reference, then the referenced node is removed from the heap. Removing a node deletes all of its variables implicitly, so a cascade effect is possible. We can also use the reset statement to delete all of the variables in a node at once, as in the following sequence which leverages the cascade effect to empty the graph-store completely.

```
in root; reset root;
```

2.2 Paths

We have two forms of variable resolution available, each corresponding to one of the special variables. Most simply, a lone variable name resolves to the value it maps to in the current node, or null if no such mapping exists. We can also access variables outside the current node via path expressions. A *path* is a period-delimited sequence of sub-expressions that each, except possibly the last, evaluate to a node reference.

pGrasp evaluates the first sub-expression of a path in the current node. Because pGrasp evaluates each remaining sub-expression in the node referenced by the previous, all but possibly the last must evaluate to a node reference. If a variable does not exist in the appropriate node, the result is the null value. In conditional test expressions, null and false are equivalent. To test for a null result directly, use an equality test expression; e.g., `x == null`. In its current form, pGrasp offers no mechanism to differentiate between a variable set to null and a variable that does not exist, hence setting a variable to null effectively unsets the variable.

Absolute paths begin with the root keyword.[1] A relative path is any path that is not an absolute path. The following statement uses an absolute path to set the current node to the node referenced by root variable `n`.

```
in root.n;
```

Because path sub-expressions are more-or-less arbitrary, we can embed branch selections directly into the path. The next example sets the current node to either `root.left.target` or `root.right.target`, depending on the value of variable `root.toggle`.

```
in root.(if toggle then left else right)
       .target;
```

Observe that pGrasp does not resolve path sub-expression variables in the current node implicitly. If we want to access a current node variable from within a path sub-expression, we must specify so explicitly. For instance, the following statement conditions the path from the previous example on the current node variable `toggle`.

```
in root.(if this.toggle then left else right)
       .target;
```

[1] To avoid clutter, I assume that the root variables are always in scope for the remainder of this document.

2.3 Procedures

Procedure semantics in pGrasp are extremely simple because the run-time environment does not maintain an implicit call stack. Instead, a called procedure stores its arguments in the current node and then replaces the calling statement with its body. By convention, the caller of a procedure must back up any important variables in the current node before issuing the call.

Procedure construction is simple. To create a procedure, use the proc statement. Take, for example, the following implementation of a list data structure.

```
proc root nil () {set this nil? true};
proc root cons (car,cdr) {set this pair? true};
```

These procedures are essentially data constructors—they embed a particular structure within the current node. The first procedure, `root.nil`, takes no arguments. When called, it sets the current node variable `nil?` to true. The second procedure, `root.cons`, takes two arguments: `car` and `cdr`. When called, it stores its arguments in the current node and then sets the current node variable `pair?` to true. The following rewrite rules illustrate these semantics.

$$\boxed{n^c} \; \overset{\texttt{nil()}}{\rightsquigarrow} \; \boxed{\begin{array}{c} n^c \\ \texttt{nil?} \mapsto \texttt{true} \end{array}}$$

$$\boxed{n^c} \; \overset{\texttt{cons}(v_1, v_2)}{\rightsquigarrow} \; \boxed{\begin{array}{l} \quad\;\; n^c \\ \texttt{car} \mapsto v_1 \\ \texttt{cdr} \mapsto v_2 \\ \texttt{cons?} \mapsto \texttt{true} \end{array}}$$

For convenience, pGrasp provides four procedure calling constructs. The call statement is the fundamental procedure call construct; it simply installs the arguments into the current node and evaluates the body. The call statement is a useful tool for exploring alternatives to traditional stack-based procedure call semantics for, e.g., various forms of distributed task scheduling [2], parsing ambiguous grammars [12], or various other cases involving non-determinism that might be desirable in a distributed setting.

The build statement behaves like the call statement, but changes the current node variable for the duration of the call only. This is useful for directing simple constructor-like procedures, such as `nil` and `cons`, into an existing node. The following example constructs an empty list and stores a reference to it in the current node variable L

```
node this L; build L nil ();
```

Because of the complexity involved in managing the call stack explicitly, the code and build statements are of limited utility. For higher-level programming tasks, the remaining two constructs provide the familiar procedure call semantics. The new statement behaves like the build statement, but first installs a fresh current node. The return statement behaves similarly, but keeps only a single variable from the new node. In this next example, either statement is equivalent to the previous example.

```
new this L nil ();
return this L nil () this;
```

2.4 I/O

A graph-store instance contains, in addition to its heap and special variables, a communications dispatch object. A *dispatch object* is an indexed set of pairs of byte queues. A *cursor* is a reference to a byte queue pair. In order to connect to external resources like human input devices and network sockets, we submit a description of the desired resource to the dispatch object. If the resource is available, the dispatch object returns a cursor. When the external resource is another pGrasp instance, we can exchange sub-graph-stores with simple primitive constructs.

pGrasp mediates all external communications through its dispatch object. To connect to a resource, use the open statement. This construct takes a node describing the requested resource, along with a target node and variable name. The target variable receives a cursor representing a connection to the resource if successful, or null otherwise. For convenience, I assume the existence of two universal cursors, accessible by keywords stdio and stderr. No standard external resource request format exists yet, so I assume the following constructors produce valid requests for network sockets.

```
proc root tcp (host,port) {set this tcp? true};
proc root udp (host,port) {set this udp? true};
```

We can exchange bytes over an established connection with the send and recv statements. To terminate an established connection, use the close statement. The following example creates a TCP connection to the UCLA public web server, sends a simple HTTP request, blocks until it receives the first TCP packet of the HTTP response, then terminates the connection.

```
new this addr tcp ("www.ucla.edu", 80);
open addr this ucla;
send ucla "GET / HTTP/1.0\r\n";
recv ucla this response;
close ucla;
```

When connected to another pGrasp instance, we can exchange entire sub-graph-stores with the print and read statements. This simple mechanism extends the principles of GSP into the realm of distributed computing.

Suppose that pGrasp instance A can request a listening TCP socket on host/port pair `example.com/3636`, which will block until a connection is established. Suppose, further, that pGrasp instance B has a structured metadata collection rooted at `project.data`, which includes any procedures necessary for managing the collection. Then, the following example transmits a complete copy of the collection and management code from A to B.

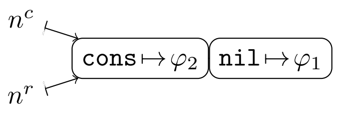
```
/* in instance A */
new this addr tcp ("example.com", 3636);
open addr this B;
print B project.data;
close B;

/* in instance B */
new this addr listen ("0.0.0.0", 3636);
open addr this A;
node this project;
read A project data;
close A;
```

These high-level constructs for structured data interchange reduce the barriers for entry to a broad range of distributed computing tasks. Virtually every communication layer of a modern web software stack fits naturally into this model. Furthermore, because procedures and evaluation contexts are just data, computations can be distributed easily across a network.

3. Example Sessions

3.1 Lists

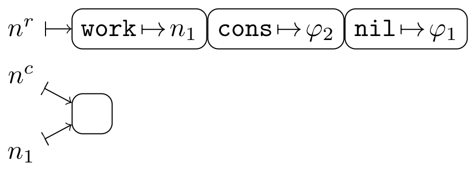
Suppose we start a pGrasp session that contains only the aforementioned list constructors, as illustrated in the following heap diagram.



First, create a work space and set it as the current node.

```
node root work;
in work;
```

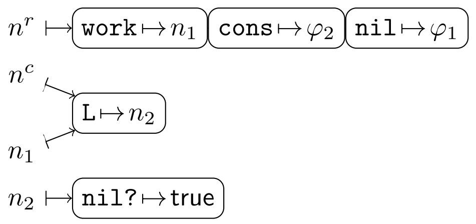
The heap now has the following structure.



Next, construct an empty list.

```
new this L nil ();
```

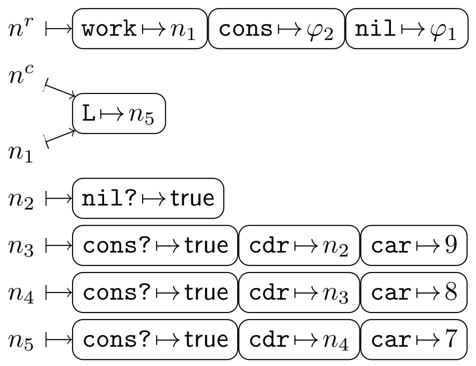
We now have a reference to an empty list in the current node variable L.



Extend the list by three numbers.

```
new this L cons (9,L);
new this L cons (8,L);
new this L cons (7,L);
```

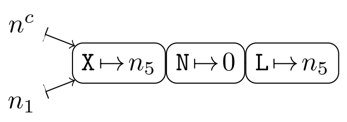
Each new statement overwrites L with a link to a fresh cons cell. The list appears to grow because the cdr of each new link receives the old value of L.



We can calculate the length of L in an imperative style. First, initialize a counter and make a working copy of L.

```
set this N 0;
set this X L;
```

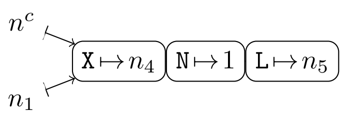
The current node now has the following structure.



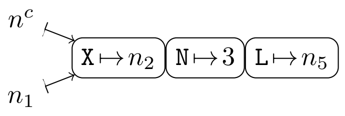
Next, write the counting loop.

```
while (not X.nil?) {
  set this N (N + 1);
  set this X X.cdr }
```

Each iteration increments N by one and discards a single link. After the first iteration, X references the second element of L.



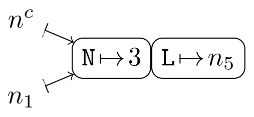
After the third iteration, X references the end of L and the loop terminates.



Delete X, as it is no longer useful.

```
unset this X;
```

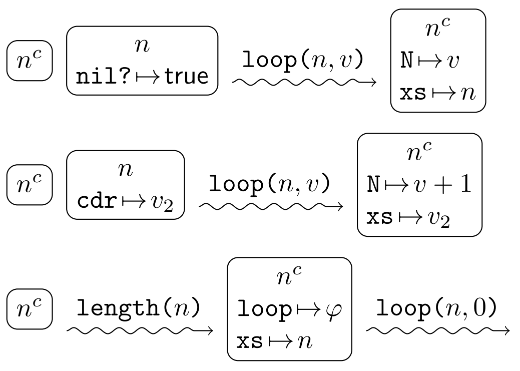
The current node appears as expected.



The following procedure calculates the length of a list, but in a functional style. In this case, the call statements implement tail recursion.

```
proc root length (xs) {
  proc this loop (xs,N) {
    when (not xs.nil?)
      call loop (xs.cdr,N+1);
  };
  call loop (xs,0) }
```

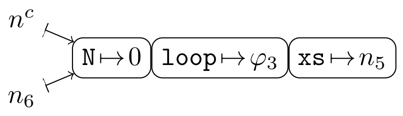
The rewrite rules for this procedure are not so obvious. Since the call statement essentially applies an arbitrary rewrite rule, I represent the loop initiation call as a dangling transition arrow. This notational kludge is an unfortunate consequence of the tension between modular abstraction (procedures) and staging (rewrite rules). In a truly non-modular setting, such artifacts do not arise.



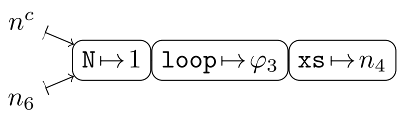
Because pGrasp procedures do not return values in the traditional sense, each procedure must specify a convention for identifying its results. Though more sophisticated conventions are possible, I assume that the result is apparent by inspection. In this case, N contains the result. Since N contains the only interesting value, we use the return statement when calling this procedure.

```
return this M length (L) N;
```

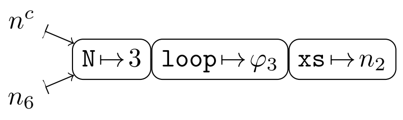
The return statement sets the call up in a fresh working node, and the procedure body installs the inner loop immediately afterward. Then, the first call to loop sets up the counter N.



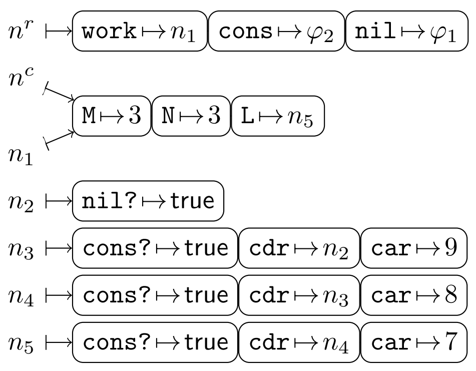
The inner loop call increments the counter by one and discards a single link. After the first recurrence, the current node appears as follows.



After the third recurrence, the procedure ends.



Finally, the caller's current node variable is restored and the current node variable M receives a copy of N. We lose the call's working node because no other references to it exists. Figure 1 shows the resulting graph-store. Here is the final heap.



3.2 Queues

Queues are a handy data structure for, among other things, distributed task scheduling. We can construct queues readily from pairs of lists [9] according to the banker's method. In this method, we call one list the *front* and the other the *rear*. New elements are *snoc*ed onto the *rear*. Whenever the rear becomes longer than the front, we append the front to the reversed rear. Thus, we need list reverse and append operations.

The following implementation of the reverse operation simply copies the elements of list xs onto a new list ys in a tail recursive style. Because I use call to initiate the inner loop procedure, I do not need to give ys as an argument to loop. Despite the apparent lexical scoping of ys, pGrasp enforces no single variable scoping policy beyond the nature of nodes and paths. We might say that pGrasp encourages a policy of selective dynamic scope.

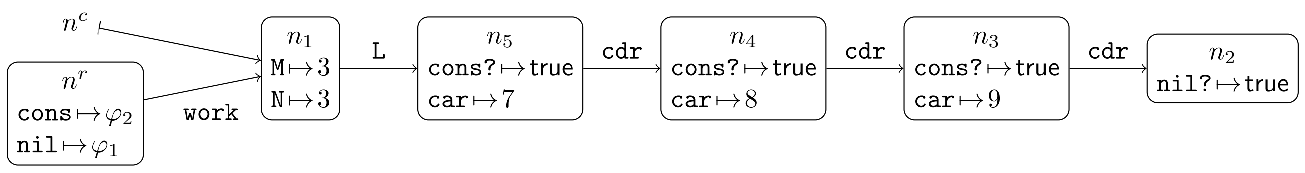


Figure 1: A list structure.

```
proc root reverse (xs) {
  new this ys nil ();
  proc this loop (xs) {
    when (not xs.nil?) {
      new this ys cons (xs.car,ys);
      call loop (xs.cdr) } };
  call loop (xs) }
```

We also need a list append operation. The next procedure first copies the reversed elements of ys, followed by the reversed elements of xs, onto a new list zs. An imperative style produces the most compact code for this example.

```
proc root append (xs,ys) {
  new this zs nil ();
  return this ys reverse (ys) ys;
  return this xs reverse (xs) ys;
  while (not ys.nil?) {
    new this zs cons (ys.car,zs);
    set this ys ys.cdr };
  while (not xs.nil?) {
    new this zs cons (xs.car,zs);
    set this xs xs.cdr } };
```

We are ready to implement the queue. The following constructor takes no arguments and constructs two lists with corresponding length counters. It also installs two helper procedures like OOP methods. The first, snoc, conses an element onto the rear of the queue and then calls the second, cleanup, to enforce the constraint on member list lengths.

```
proc root queue () {
  new this F nil ();
  new this R nil ();
  set this lenF 0;
  set this lenR 0;
  proc this snoc (x) {
    new this R cons (x,R);
    set this R (R + 1);
    call cleanup () } };
  proc this cleanup () {
    when (lenR > lenF) {
      return this R reverse (R) ys;
      return this F append (F,R) zs;
      new this R nil ();
      set this lenF (lenF + lenR);
      set this lenR 0 } }
```

Because the effects of snoc and cleanup are confined entirely to the queue instance they inhabit, the object-oriented style admits the simplest code. The same is not true for the remaining mutators because they return some result to the caller. The head procedure copies the element at the front of queue q into the variable x so the caller can fetch it with a return statement. If q is empty, x is null.

```
proc root head (q) {
  when (not q.F.nil?)
    set this x q.F.car }
```

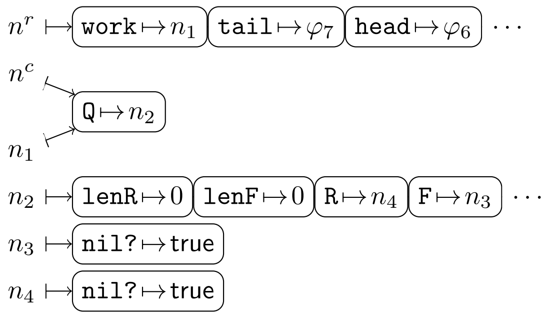
The tail procedure duplicates the node referenced by q into variable r, then eliminates the front-most element of r. I use the build statement to direct r.cleanup into r. Equivalently, I could have built q.cleanup inside r.

```
proc root tail (q) {
  if (q.F.nil?)
    set this r q;
  else {
    dup r this q;
    set r F F.cdr;
    set r lenF (r.lenF - 1);
    build r r.cleanup () } };
```

Now, suppose we start a new pGrasp session with lists and queues, and a fresh work space in the current node. First, construct an empty queue.

```
new this Q queue ();
```

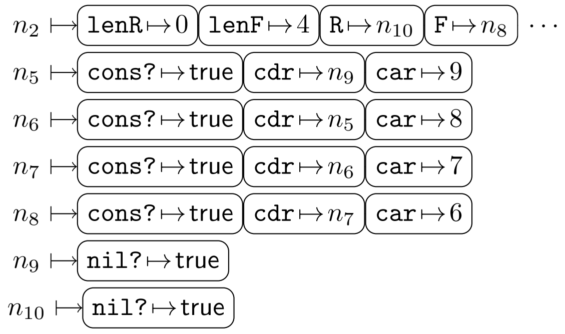
The heap now has the following structure.



Enqueue four numbers.

```
build Q Q.snoc (6);
build Q Q.snoc (7);
build Q Q.snoc (8);
build Q Q.snoc (9);
```

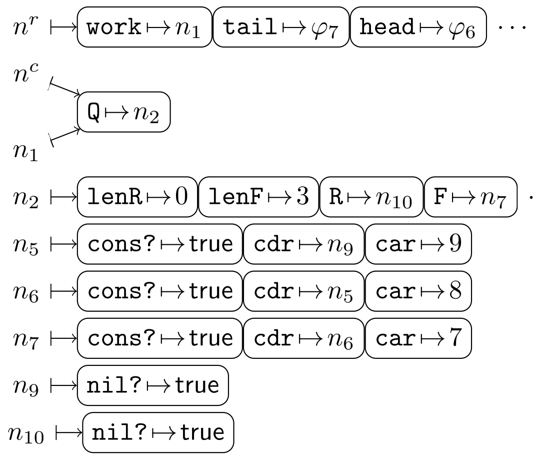
The first `snoc` conses 9 onto the rear of `Q`. Since `Q.F` is empty, `cleanup` moves the rear to the front. The second `snoc` conses 8 onto `Q.R` without calling `cleanup`. The third `snoc` behaves similar to the first.


n_2 ↦ lenR↦0 lenF↦4 R↦n_{10} F↦n_8 ⋯
n_5 ↦ cons?↦true cdr↦n_9 car↦9
n_6 ↦ cons?↦true cdr↦n_5 car↦8
n_7 ↦ cons?↦true cdr↦n_6 car↦7
n_8 ↦ cons?↦true cdr↦n_7 car↦6
n_9 ↦ nil?↦true
n_{10} ↦ nil?↦true

Pop an element off the queue. Since `Q.R` is empty, no `cleanup` is necessary.

```
return this Q tail (Q) r;
```

The queue appears to shrink because the variable `Q` receives a copy if its "tail." Figure 2 show the resulting graph-store. Here is the final heap.


n^r ↦ work↦n_1 tail↦φ_7 head↦φ_6 ⋯
n^c
Q↦n_2
n_1
n_2 ↦ lenR↦0 lenF↦3 R↦n_{10} F↦n_7 ⋯
n_5 ↦ cons?↦true cdr↦n_9 car↦9
n_6 ↦ cons?↦true cdr↦n_5 car↦8
n_7 ↦ cons?↦true cdr↦n_6 car↦7
n_9 ↦ nil?↦true
n_{10} ↦ nil?↦true

The loose connectivity of this distributed task queue exemplifies a growing class of socially-motivated distributed software systems, where the operator of a system can contribute computing resources to others. With reasonable effort, we could extend this example with identity management and task submission quotas. Perhaps we could tie quotas meaningfully to the amount of resources contributed over time, to be bought and sold like a commodity, and enabling a digital equivalent to taxes.

3.3 Distributed Tasks

Since pGrasp does not support concurrency directly, I implement a distributed task scheduler in the following round-about manner. A dedicated pGrasp instance hosts a distributed task queue, along with a server program and a client API. The server program listens for incoming requests. The client API contains commands to request and submit labeled tasks and results.

The server's task queue remains empty until another pGrasp instance submits some tasks. If an instance requests a task and the queue is not empty, the head task is removed from the queue and given to the requester along with a serial number. The requester must then run the task and submit the result back to the server. The server caches results until another instance requests them.

Suppose we have two descriptively named procedures, `listen` and `connect`, that take no arguments and produce a cursor named `c`. Then figure 3 contains an implementation of the distributed task queue.

The task submission command takes a list of nodes, where each node in the list contain a `label` variable and a `run` procedure that takes no arguments and stores its result in a variable named `result`. The results request command takes a label and stores any returned results in a variable named `results`. The task request command takes no arguments. It requests and runs a single task from the server, then returns the result back to the server along with the serial and label embedded in the task. The server program takes no arguments. It listens for a client action and reacts accordingly.

4. Formal Specification

4.1 Syntax

Figures 4-8 present the formal grammar. I partition the grammar into three major syntactic classes: statements, expressions, and values. A valid statement reduces to ε (skip) while possibly affecting the run-time state. A valid expression evaluates, without effect, to a *value*, i.e., a normal form. Denote statements, expressions, and values by meta-variables s, e, and v, respectively.

4.2 Semantics

Figures 9-12 present the operational semantics of pGrasp as inference rules in a small-step style. Denote by G a graph store, and by D a dispatch object. A judgment of the form $\langle G, e\rangle \rightsquigarrow e'$ specifies a step from expression e to expression e' in the context of graph-store G. A judgment of the form $\langle G, D, s\rangle \rightsquigarrow \langle G', D', s'\rangle$ specifies a step from statement s to statement s' in the context of graph-store G and dispatch object D. The resulting state consists of graph-store G' and dispatch object D'.

Denote by H, Σ, and n a heap and a node, and a node reference, respectively. Denote by $\varnothing$ and Q a fresh empty node and a fresh empty queue, respectively. Denote by n^r and n^c a root reference and a current node reference, respectively. I occasionally decompose a graph-store into a tuple of its components: $G = (H, n^r, n^c)$.

Write Δ a dictionary such as a node or dispatch object. Denote by $\Delta[\alpha \mapsto \beta]$ the dictionary Δ extended with a mapping from α to β, where α is some reference object or variable name, depending on the nature of Δ, and β is some

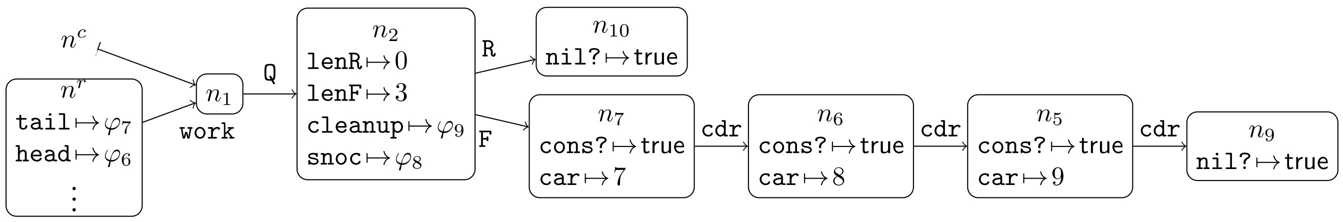


Figure 2: A queue structure.

```
proc root sendtasks (label,tasks) {
  return this c connect () c;
  print c "send tasks";
  print c label;
  while (not tasks.nil?) {
    print c tasks.car;
    set this tasks tasks.cdr };
  print c "done";
  close c };

proc root getresults (label) {
  return this c connect () c;
  print c "get results";
  print c label;
  read c this results;
  close c };

proc root gettask () {
  return this c connect () c;
  print c "get task";
  read c this task;
  close c;
  when (not (task == null))
    build task task.run ();
  return this c connect () c;
  print c "send result";
  print c task;
  close c };
```

```
proc root server () {
  node this R;
  new this Q queue ();
  set this S = 1;
  while (true) {
    return this c listen () c;
    read c cmd;

    if (cmd == "send tasks") {
      read c task;
      while (not (task == "done")) {
        set task serial S;
        set this S (S + 1);
        build Q Q.snoc (task);
        read c task } }

    else if (cmd == "get task") {
      return this task head (Q) x;
      return this Q tail (Q) r;
      print c task }

    else if (cmd == "send result") {
      read c label;
      read c task;
      when (not R.label)
        new R label nil ();
      new R label cons (task, R.label) }

    else if (cmd == "get results") {
      read c label;
      print c R.label;
      unset R label } } }
```

Figure 3: A distributed task queue.

object suitable for storage in Δ. The equation $\Delta = \Delta'[\alpha \mapsto \beta]$ denotes a dictionary Δ' of all the mappings of Δ except the one from α, if it exists. Denote by $\Delta(\alpha)$ a dictionary look up:

$$\Delta(\alpha) = \beta \quad \Longleftrightarrow \quad \Delta = \Delta'[\alpha \mapsto \beta]$$

If Δ contains no mapping from α, then $\Delta(\alpha) = \mathsf{null}$. Denote by $\Delta(\alpha_1)(\alpha_2)$ a two step look up:

$$\Delta(\alpha_1)(\alpha_2) = \beta \quad \Longleftrightarrow \quad \Delta(\alpha_1) = \Delta' \wedge \Delta'(\alpha_2) = \beta$$

For convenience, define the following single-variable update relation:

$$\Delta[\alpha_1{:}\alpha_2 \mapsto \beta] \quad \equiv \quad \Delta[\alpha_1 \mapsto \Delta(\alpha_1)[\alpha_2 \mapsto \beta]]$$

Denote by $\Delta_1 \cup \Delta_2$ the dictionary that contains all mappings of Δ_1 and Δ_2, where the mappings of Δ_2 take precedence. Assuming Δ contains a mapping from α to some queue, denote by $\Delta[\alpha \Lleftarrow \beta]$ the enqueue operation of β onto the queue referenced by α, and by $\Delta[\alpha \mapsto \beta]$ the dequeue operation of β from the queue referenced by α.

4.2.1 Helpers

Denote by $\mathrm{args}(e, v)$ a fresh node in which the parameters of e are assigned to the corresponding arguments of v. Denote by $\mathrm{copy}(n)$ a fresh node reference to a shallow copy of the node referenced by n. Denote by $\mathrm{request}(n)$ a fresh cursor to the resource described by the node referenced by n, or null if the resource is unavailable. Denote by $\mathrm{flush}(c)$ the queue referenced by cursor c, emptied by unspecified means. Denote by $\mathrm{encode}(v)$ and $\mathrm{decode}(v)$ the serialized and unserialized representation, respectively, of value v.

5. Implementation

I implemented the prototype in Racket [3, Version 5.3.6], a Scheme variant. The implementation is, however, behind the current specification, so some of the programs in this document have not been tested live.

In the implementation, I represent the skip statement by the empty string. For instance, in the statement,

```
while true;
```

the body of the loop is a single skip statement. For heap garbage collection, I track node linkage by simple reference counting. For term encoding and decoding, I use the built-in Racket term printer and reader. I transmit most primitive values as-is, omitting cursors, but effective node reference encoding is still an open question. For now, I encode referenced nodes recursively.

6. Discussion

The syntax of pGrasp is closest to C, but pGrasp is closer in spirit to assembly language—neither enforces a particularly complex call stack discipline and both require careful though to avoid unintentional clobbering of data. Additionally, pGrasp has some elements of Java, e.g., references and garbage collection, along with the print/read constructs of Scheme. Because of the apparently low level at which the most basic pGrasp constructs operate, I believe a practical pGrasp platform with excellent performance, or at least reasonably good performance, is within reach. I can not yet predict the performance of a rewrite-based GSP language, although I think we could reasonably expect performance comparable to a purely functional language due to the maturity of the underlying theories and the performance of existing rewrite-based languages.

At a higher level of detail, pGrasp requires many constructs of convenience in order to produce reasonably compact code—four primitive procedure call constructs seems like too many. I also rely heavily on unscalable naming and calling conventions. I could alleviate these symptoms with a proper abstraction mechanism, but a basic macro facility could have similar effect and be more useful. Neither solution is ideal because neither is simple and none of these issues exist in a proper rewrite-based approach. Though staged evaluation might provide a scalable solution for returning values from procedure calls, the outcome betrays the overall vision for GSP and so I can not justify a possibly messy exploration into imperative staging.

The potential for explicit, high level call stack management in pGrasp is interesting in its own right, but a GSP setting offers benefits that are difficult or impossible to achieve in an imperative language. For example, fully pluggable type systems are all but trivial in a GSP language.

As a final thought, in either pGrasp or a pure GSP language, we could hook native programs directly into the encoding and decoding mechanisms with a simple callback mechanism, which would simplify integration with virtually all other technologies that use messaging, in the loose sense of messaging defined in the introduction.

7. Conclusion

I started the pGrasp project as an exercise in comparing GSP concepts to existing programming language technologies, to discover concrete examples from which to start a discussion about GSP and the issues it addresses. Designing a language like pGrasp, however, is like driving a screw with a sledge hammer—I get my point across, but I destroy the foundation in the process.

There is a clear discrepancy between the short-term goals of this project and the broader vision of the graph-store programming paradigm. With that in mind, I have identified at least three possible paths forward.

The GSP paradigm requires a fundamentally non-modular approach to software construction. Considering the pervasive importance of modularity in modern software engineering, I do not expect rapid adoption of the paradigm until I put significant work into a production platform and supporting tools. While implementing the entire vision is clearly too large a task for a single Masters thesis, a coherent language based on graph rewriting—a staged, pattern-based design—is certainly within reach.

On the other hand, GSP principles may be applied, to some extent, in any language with dictionaries, including C, Python, and Perl. For each language, I could implement the pGrasp constructs in a cross-platform message processing and passing library, though performance of such a library in any language remains unclear. From a usability perspective, this approach is most attractive for languages with strong meta-programming or other DSL construction features, such as Scheme, Haskell, and Ruby.

The current pGrasp design alludes to a similar but more intuitive and performant revision. With some work, I could polish the design and produce a useful and reasonably efficient platform. The prospect of a practical pGrasp implementation has already drawn interest from members of academia and industry.

$s ::=$	ε	skip
	$\mid\ s; s$	sequence
	$\mid$ node $e\ x$	create fresh node
	$\mid$ dup $e\ e\ x$	duplicate node
	$\mid$ set $e\ x\ e$	set variable
	$\mid$ unset $e\ x$	drop variable
	$\mid$ reset e	empty existing node
	$\mid$ in e	change work node
$e ::=$	x	variable
	$\mid\ e.e$	path expression
	$\mid$ root	root node
	$\mid$ this	current node
$v ::=$	n	node reference
	$\mid$ null	null reference

Figure 4: Core syntax.

$s ::=$	$\ldots$	
	$\mid$ proc $e\ x\ e\ s$	define procedure
	$\mid$ call $e\ e$	call procedure
	$\mid$ build $e\ e\ e$	call-in-node
	$\mid$ new $e\ x\ e\ e$	call-in-new-node
	$\mid$ return $e\ x\ e\ e\ x$	call-and-save
$e ::=$	$\ldots$	
	$\mid$ ()	unit expression
	$\mid\ e, e$	tuple expression
$v ::=$	$\ldots$	
	$\mid$ ()	unit value
	$\mid\ v, v$	tuple value
	$\mid\ \varphi(e, s)$	procedure object

Figure 5: Procedure syntax.

References

[1] T. Brus, M. C. van Eekelen, M. Van Leer, and M. J. Plasmeijer. Cleana language for functional graph rewriting. In *Functional Programming Languages and Computer Architecture*, page 364384, 1987.

[2] T. L. CASAVANT and J. G. KUHL. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 14(2):141, 1988.

[3] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang.org/tr1/.

[4] A. Goldberg and A. Kay. *Smalltalk-72: Instruction Manual*. Xerox Corporation, 1976.

[5] A. Graf. *The Pure programming language*. See, 2009.

[6] T. Isakowitz, M. Bieber, and F. Vitali. Web information systems. *Communications of the ACM*, 41(7):7880, 1998.

[7] C. B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911937, 2004.

[8] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1):73155, 1992.

[9] C. Okasaki. *Purely functional data structures*. PhD thesis, Citeseer, 1996.

[10] D. Plump. Term graph rewriting. *Handbook of graph grammars and computing by graph transformation*, 2:361, 1999.

[11] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation, vol 1: Foundations*. World Scientific, 1997.

[12] E. Scott and A. Johnstone. GLL parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, Sept. 2010. ISSN 15710661. .

[13] W. Taha. *Multi-stage programming: Its theory and applications*. PhD thesis, Citeseer, 1999.

E-PATH1
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, e_1.e_2\rangle \rightsquigarrow e_1'.e_2}$$

E-PATH21
$$\frac{\langle (H, n^r, n), e\rangle \rightsquigarrow e'}{\langle G, n.e\rangle \rightsquigarrow n.e'}$$

E-PATH22
$$\frac{}{\langle G, n_1.n_2.e\rangle \rightsquigarrow n_2.e}$$

E-PATH
$$\frac{}{\langle G, n.v\rangle \rightsquigarrow v}$$

E-VARIABLE
$$\frac{}{\langle G, x\rangle \rightsquigarrow G(n^c)(x)}$$

E-ROOT
$$\frac{}{\langle G, \mathsf{root}\rangle \rightsquigarrow n^r}$$

E-THIS
$$\frac{}{\langle G, \mathsf{this}\rangle \rightsquigarrow n^c}$$

S-SEQUENCE1
$$\frac{\langle G, D, s_1\rangle \rightsquigarrow \langle G, D, s_1'\rangle}{\langle G, D, s_1; s_2\rangle \rightsquigarrow \langle G, D, s_1'; s_2\rangle}$$

S-SEQUENCE2
$$\frac{}{\langle G, D, \varepsilon; s\rangle \rightsquigarrow \langle G, D, s\rangle}$$

S-NODE1
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{node}\ e\ x\rangle \rightsquigarrow \langle G, D, \mathsf{node}\ e'\ x\rangle}$$

S-NODE
$$\frac{G' = G[n{:}x \mapsto \varnothing]}{\langle G, D, \mathsf{node}\ n\ x\rangle \rightsquigarrow \langle G', D, \varepsilon\rangle}$$

S-DUP1
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{dup}\ e_1\ e_2\ x\rangle \rightsquigarrow \langle G, D, \mathsf{dup}\ e_1'\ e_2\ x\rangle}$$

S-DUP2
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{dup}\ n\ e\ x\rangle \rightsquigarrow \langle G, D, \mathsf{dup}\ n\ e'\ x\rangle}$$

S-DUP
$$\frac{G' = G[n_2{:}x \mapsto \mathrm{copy}(n_1)]}{\langle G, D, \mathsf{dup}\ n_1\ n_2\ x\rangle \rightsquigarrow \langle G', D, \varepsilon\rangle}$$

S-SET1
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{set}\ e_1\ x\ e_2\rangle \rightsquigarrow \langle G, D, \mathsf{set}\ e_1'\ x\ e_2\rangle}$$

S-SET3
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{set}\ n\ x\ e\rangle \rightsquigarrow \langle G, D, \mathsf{set}\ n\ x\ e'\rangle}$$

S-SET
$$\frac{G' = G[n{:}x \mapsto v]}{\langle G, D, \mathsf{set}\ n\ x\ v\rangle \rightsquigarrow \langle G', D, \varepsilon\rangle}$$

S-UNSET1
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{unset}\ e\ x\rangle \rightsquigarrow \langle G, D, \mathsf{unset}\ e'\ x\rangle}$$

S-UNSET
$$\frac{G(n) = \Sigma[x \mapsto v]}{\langle G, D, \mathsf{unset}\ n\ x\rangle \rightsquigarrow \langle G[n \mapsto \Sigma], \varepsilon\rangle}$$

S-RESET1
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{reset}\ e\rangle \rightsquigarrow \langle G, D, \mathsf{reset}\ e'\rangle}$$

S-RESET
$$\frac{G' = G[n \mapsto \varnothing]}{\langle G, D, \mathsf{reset}\ n\rangle \rightsquigarrow \langle G', D, \varepsilon\rangle}$$

S-IN1
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{in}\ e\rangle \rightsquigarrow \langle G, D, \mathsf{in}\ e'\rangle}$$

S-IN
$$\frac{}{\langle G, D, \mathsf{in}\ n\rangle \rightsquigarrow \langle (H, n^r, n), D, \varepsilon\rangle}$$

Figure 9: Core semantics.

$s ::= \ldots$		
$\mid$	open $e\ e\ x$	create cursor
$\mid$	close e	destroy cursor
$\mid$	send $e\ e$	raw output
$\mid$	recv $e\ e\ x$	raw input
$\mid$	print $e\ e$	structured output
$\mid$	read $e\ e\ x$	structured input
$e ::= \ldots$		
$\mid$	stdio	standard input/output
$\mid$	stderr	standard error
$v ::= \ldots$		
$\mid$	c	cursor value

Figure 6: I/O syntax.

$s ::= \ldots$		
$\mid$	when $e\ s$	optional statement
$\mid$	if $e\ s$ else s	conditional statement
$\mid$	while $e\ s$	iteration
$e ::= \ldots$		
$\mid$	if $e\ e$ else e	conditional expression
$\mid$	e and e	conjunction
$\mid$	e or e	disjunction
$\mid$	not e	negation
$\mid$	e == e	equality test
$\mid$	true	truth
$\mid$	false	untruth
$v ::= \ldots$		
$\mid$	true	truth value
$\mid$	false	untruth value

Figure 7: Logic syntax.

E-TUPLE1
$$\frac{\langle G, e_1\rangle \leadsto e_1'}{\langle G, e_1, e_2\rangle \leadsto e_1', e_2}$$

E-TUPLE2
$$\frac{\langle G, e\rangle \leadsto e'}{\langle G, v, e\rangle \leadsto v, e'}$$

S-PROC1
$$\frac{\langle G, e_1\rangle \leadsto e_1'}{\langle G, D, \mathsf{proc}\ e_1\ x\ e_2\ s\rangle \leadsto \langle G, D, \mathsf{proc}\ e_1'\ x\ e_2\ s\rangle}$$

S-PROC
$$\frac{G' = G[n{:}x \mapsto \varphi(e, s)]}{\langle G, D, \mathsf{proc}\ n\ x\ e\ s\rangle \leadsto \langle G', D, \varepsilon\rangle}$$

S-CALL1
$$\frac{\langle G, e_1\rangle \leadsto e_1'}{\langle G, D, \mathsf{call}\ e_1\ e_2\rangle \leadsto \langle G, D, \mathsf{call}\ e_1'\ e_2\rangle}$$

S-CALL2
$$\frac{\langle G, e\rangle \leadsto e'}{\langle G, D, \mathsf{call}\ \varphi\ e\rangle \leadsto \langle G, D, \mathsf{call}\ \varphi\ e'\rangle}$$

S-CALL
$$\frac{G' = G[n^c \mapsto G(n^c) \cup args(e, v)]}{\langle G, D, \mathsf{call}\ \varphi(e, s)\ v\rangle \leadsto \langle G', D, s\rangle}$$

S-BUILD1
$$\frac{\langle G, e_1\rangle \leadsto e_1'}{\langle G, D, \mathsf{build}\ e_1\ e_2\ e_3\rangle \leadsto \langle G, D, \mathsf{build}\ e_1'\ e_2\ e_3\rangle}$$

S-BUILD2
$$\frac{\langle G, e_2\rangle \leadsto e_2'}{\langle G, D, \mathsf{build}\ n\ e_2\ e_3\rangle \leadsto \langle G, D, \mathsf{build}\ n\ e_2'\ e_3\rangle}$$

S-BUILD3
$$\frac{\langle G, e_3\rangle \leadsto e_3'}{\langle G, D, \mathsf{build}\ n\ \varphi\ e_3\rangle \leadsto \langle G, D, \mathsf{build}\ n\ \varphi\ e_3'\rangle}$$

S-BUILD
$$\frac{s = (\mathsf{in}\ n; \mathsf{call}\ \varphi\ v; \mathsf{in}\ n^c)}{\langle G, D, \mathsf{build}\ n\ \varphi\ v\rangle \leadsto \langle G, D, s\rangle}$$

S-NEW
$$\frac{s = (\mathsf{node}\ e_1\ x_1; \mathsf{build}\ e_1.x_1\ e_2\ e_3)}{\langle G, D, \mathsf{new}\ e_1\ x_1\ e_2\ e_3\rangle \leadsto \langle G, D, s\rangle}$$

E-RETURN
$$\frac{s = (\mathsf{build}\ n\ e_2\ e_3; \mathsf{set}\ e_1\ x_1\ n.x_2) \qquad n\ \mathsf{fresh}}{\langle G, D, \mathsf{return}\ e_1\ x_1\ e_2\ e_3\ x_2\rangle \leadsto \langle G, D, s\rangle}$$

Figure 10: Procedure semantics.

$$\begin{array}{rcll}
e & ::= & e \mathbin{\mathsf{+}} e & \text{addition} \\
& | & e \mathbin{\mathsf{-}} e & \text{subtraction} \\
& | & e \mathbin{\mathsf{*}} e & \text{multiplication} \\
& | & e \mathbin{\mathsf{/}} e & \text{division} \\
& | & \mathbb{R} & \text{number} \\
v & ::= & \dots & \\
& | & \mathbb{R} & \text{numeric value}
\end{array}$$

Figure 8: Arithmetic syntax.

E-Stdio

$$\frac{}{\langle G, \mathsf{stdio}\rangle \rightsquigarrow c^{io}}$$

E-Stderr

$$\frac{}{\langle G, \mathsf{stderr}\rangle \rightsquigarrow c^{err}}$$

S-Open1

$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{open}\ e_1\ e_2\ x\rangle \rightsquigarrow \langle G, D, \mathsf{open}\ e_1'\ e_2\ x\rangle}$$

S-Open2

$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{open}\ n\ e\ x\rangle \rightsquigarrow \langle G, D, \mathsf{open}\ n\ e'\ x\rangle}$$

S-Open

$$\frac{G' = G[n_2{:}x \mapsto c] \qquad D'[c \mapsto Q] \qquad c = \mathrm{request}(n_1)}{\langle G, D, \mathsf{open}\ n_1\ n_2\ x\rangle \rightsquigarrow \langle G', D', \varepsilon\rangle}$$

S-Close1

$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{close}\ e\rangle \rightsquigarrow \langle G, D, \mathsf{close}\ e'\rangle}$$

S-Close

$$\frac{D = D'[c \mapsto \mathrm{flush}(c)]}{\langle G, D, \mathsf{close}\ c\rangle \rightsquigarrow \langle G, D', \varepsilon\rangle}$$

S-Send1

$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{send}\ e_1\ e_2\rangle \rightsquigarrow \langle G, D, \mathsf{send}\ e_1'\ e_2\rangle}$$

S-Send2

$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{send}\ c\ e\rangle \rightsquigarrow \langle G, D, \mathsf{send}\ c\ e'\rangle}$$

S-Send

$$\frac{D' = D[c \Mapsfrom v]}{\langle G, D, \mathsf{send}\ c\ v\rangle \rightsquigarrow \langle G, D', \varepsilon\rangle}$$

S-Recv1

$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{recv}\ e_1\ e_2\ x\rangle \rightsquigarrow \langle G, D, \mathsf{recv}\ e_1'\ e_2\ x\rangle}$$

S-Recv2

$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{recv}\ c\ e\ x\rangle \rightsquigarrow \langle G, D, \mathsf{recv}\ c\ e'\ x\rangle}$$

S-Recv

$$\frac{D' = D[c \Mapsto v] \qquad G' = G[n{:}x \mapsto v]}{\langle G, D, \mathsf{recv}\ c\ n\ x\rangle \rightsquigarrow \langle G', D', \varepsilon\rangle}$$

S-Print1

$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{print}\ e_1\ e_2\rangle \rightsquigarrow \langle G, D, \mathsf{print}\ e_1'\ e_2\rangle}$$

S-Print2

$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{print}\ c\ e\rangle \rightsquigarrow \langle G, D, \mathsf{print}\ c\ e'\rangle}$$

S-Print

$$\frac{D' = D[c \Mapsfrom \mathrm{encode}(v)]}{\langle G, D, \mathsf{print}\ c\ v\rangle \rightsquigarrow \langle G, D', \varepsilon\rangle}$$

S-Read1

$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{read}\ e_1\ e_2\ x\rangle \rightsquigarrow \langle G, D, \mathsf{read}\ e_1'\ e_2\ x\rangle}$$

S-Read2

$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{read}\ c\ e\ x\rangle \rightsquigarrow \langle G, D, \mathsf{read}\ c\ e'\ x\rangle}$$

S-Read

$$\frac{D' = D[c \Mapsto v] \qquad G' = G[n{:}x \mapsto \mathrm{decode}(v)]}{\langle G, D, \mathsf{read}\ c\ n\ x\rangle \rightsquigarrow \langle G', D', \varepsilon\rangle}$$

Figure 11: I/O semantics.

E-IF1
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, \mathsf{if}\ e_1\ e_2\ \ \mathsf{else}\ e_3\rangle \rightsquigarrow \mathsf{if}\ e_1'\ e_2\ \ \mathsf{else}\ e_3}$$

E-IF2
$$\frac{v \notin \{\mathsf{false}, \mathsf{null}\}}{\langle G, \mathsf{if}\ v\ e_2\ \ \mathsf{else}\ e_3\rangle \rightsquigarrow e_2}$$

E-IF3
$$\frac{v \in \{\mathsf{false}, \mathsf{null}\}}{\langle G, \mathsf{if}\ v\ e_2\ \ \mathsf{else}\ e_3\rangle \rightsquigarrow e_3}$$

E-AND1
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, e_1\ \ \mathsf{and}\ e_2\rangle \rightsquigarrow e_1'\ \ \mathsf{and}\ e_2'}$$

E-AND2
$$\frac{v \notin \{\mathsf{false}, \mathsf{null}\}}{\langle G, v\ \ \mathsf{and}\ e_2\rangle \rightsquigarrow e_2}$$

E-AND3
$$\frac{v \in \{\mathsf{false}, \mathsf{null}\}}{\langle G, v\ \ \mathsf{and}\ e_2\rangle \rightsquigarrow \mathsf{false}}$$

E-OR1
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, e_1\ \ \mathsf{or}\ e_2\rangle \rightsquigarrow e_1'\ \ \mathsf{or}\ e_2}$$

E-OR2
$$\frac{v \in \{\mathsf{false}, \mathsf{null}\}}{\langle G, v\ \ \mathsf{or}\ e_2\rangle \rightsquigarrow e_2}$$

E-OR3
$$\frac{v \notin \{\mathsf{false}, \mathsf{null}\}}{\langle G, v\ \ \mathsf{or}\ e_2\rangle \rightsquigarrow v}$$

E-NOT
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, \mathsf{not}\ e\rangle \rightsquigarrow \mathsf{not}\ e'}$$

E-NOT1
$$\frac{v \in \{\mathsf{false}, \mathsf{null}\}}{\langle G, \mathsf{not}\ v\rangle \rightsquigarrow \mathsf{true}}$$

E-NOT2
$$\frac{v \notin \{\mathsf{false}, \mathsf{null}\}}{\langle G, \mathsf{not}\ v\rangle \rightsquigarrow \mathsf{false}}$$

E-EQUALS1
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, e_1 \mathrel{==} e_2\rangle \rightsquigarrow e_1' \mathrel{==} e_2}$$

E-EQUALS2
$$\frac{\langle G, e_2\rangle \rightsquigarrow e_2'}{\langle G, v_1 \mathrel{==} e_2\rangle \rightsquigarrow v_1 \mathrel{==} e_2'}$$

E-EQUALS3
$$\frac{v_1 = v_2}{\langle G, v_1 \mathrel{==} v_2\rangle \rightsquigarrow \mathsf{true}}$$

E-EQUALS4
$$\frac{v_1 \neq v_2}{\langle G, v_1 \mathrel{==} v_2\rangle \rightsquigarrow \mathsf{false}}$$

S-WHEN
$$\frac{\langle G, e\rangle \rightsquigarrow e'}{\langle G, D, \mathsf{when}\ e\ s\rangle \rightsquigarrow \langle G, D, \mathsf{when}\ e'\ s\rangle}$$

S-WHEN1
$$\frac{v \in \{\mathsf{false}, \mathsf{null}\}}{\langle G, D, \mathsf{when}\ v\ s\rangle \rightsquigarrow \langle G, D, \varepsilon\rangle}$$

S-WHEN2
$$\frac{v \notin \{\mathsf{false}, \mathsf{null}\}}{\langle G, D, \mathsf{when}\ v\ s\rangle \rightsquigarrow \langle G, D, s\rangle}$$

S-IF
$$\frac{\langle G, e_1\rangle \rightsquigarrow e_1'}{\langle G, D, \mathsf{if}\ e\ s_1\ \ \mathsf{else}\ s_2\rangle \rightsquigarrow \langle G, D, \mathsf{if}\ e'\ s_1\ \ \mathsf{else}\ s_2\rangle}$$

S-IF1
$$\frac{v \notin \{\mathsf{false}, \mathsf{null}\}}{\langle G, D, \mathsf{if}\ v\ s_1\ \ \mathsf{else}\ s_2\rangle \rightsquigarrow \langle G, D, s_1\rangle}$$

S-IF2
$$\frac{v \in \{\mathsf{false}, \mathsf{null}\}}{\langle G, D, \mathsf{if}\ v\ s_1\ \ \mathsf{else}\ s_2\rangle \rightsquigarrow \langle G, D, s_2\rangle}$$

S-WHILE
$$\frac{}{\langle G, D, \mathsf{while}\ e\ s\rangle \rightsquigarrow \langle G, D, \mathsf{when}\ e\ (s; \mathsf{while}\ e\ s)\rangle}$$

Figure 12: Logic semantics.