1 The pCG Language

1.1 Introduction

This chapter describes in detail the pCG language and its interpreter. It details the language in terms of its design goals, informal semantics, and major features, in order to make the reader familiar enough with it to be able to understand pCG programs in the 'Experiments' chapter and appendices. The documentation for actors and processes occurs near the end of this chapter after the essentials of pCG have been dealt with, much of which is supporting infrastructure.

After reading this chapter, or even before completing it, the reader may wish to skip to the 'Experiments' chapter for more substantial examples of working with pCG, before delving into more detail in 'The Implementation of pCG'. The content of the various tables in this chapter may be skimmed on a first reading and consulted later for reference.

The distribution (see 'Availability and Requirements') contains additional documentation such as JavaDocs[1] for pCG's run-time class library, and Extended Backus-Naur Form (EBNF) for pCG's grammar.

The primary motivation for the development of pCG is as a means to express Mineau's process formalism [Mineau 1998] in a concrete form. Accordingly, the abbreviation, *pCG*, is meant to indicate a *CG* being operated upon by some process *p*. The language is also this author's interpretation of Mineau's CPE.

1.2 Availability and Requirements

The pCG interpreter and its source code are available as a ZIP archive from http://www.adelaide.net.au/~dbenn/Masters/index.html with the executables and source code being redistributable under the GNU Public Licence. Simply unzip the downloaded archive and view the README file for usage information.

Java ™ 2 (in particular JDK or JRE 1.2.2 or higher) is required to run pCG. Earlier versions of Java will not suffice as pCG uses the Collections component of the Java Foundation Classes. Although pCG was developed under Linux, it can in principle be executed on any machine with a Java 2 run-time environment. It has been used by the author under Red Hat Linux 6.0 and Windows 98.

The pCG interpreter as currently implemented is invoked from the command-line, interprets a single source file, and has no interactive mode. pCG is not yet production quality software, but is quite usable. See also 'Future Work'.

1.3 Design Goals and Influences

One ought to approach the design of a programming language with some trepidation as it is fraught with dangers [2]. One strategy to mitigate the risk of failure is to decouple syntax from semantics to the extent possible, so that the nature of the source language can change easily if desired. The focus then shifts to getting the semantics right. The means by which this decoupling has been achieved shall be discussed in 'The Implementation of pCG'. The choice of a new language is a deliberate one, to clearly state that something different is being represented, and to avoid confusing similarities with existing languages. Given that major new constructs are required (e.g. actor type definition, process statement) which will take the form of new syntax, and that significant new types must be introduced (e.g. concept, graph), an existing language would require significant extension. One drawback of creating a new language is that no pre-existing utility source code in that language exists, but since pCG has an extensible type system (see 'User-defined Types'), utility code written in Java can be used.

The author agrees with the following intuitions regarding language design, particularly the second:

---

[1] HTML documentation automatically extracted from Java source code.

[2] Other adventures in programming language design and implementation by the author can be found at http://www.adelaide.net.au/~dbenn/docs/projects.html

> [s]mall languages tend to be better designed than large ones, showing fewer signs of ad hoc compromise between conflicting aesthetic principles and design goals.
>
> (Stone 1993), cited in [Gabriel 1996]

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today. [3]
>
> (Clinger and Rees 1993), cited in [Gabriel 1996]

There is a distinct tension between purity and utility insofar as language design is concerned. Very little has been documented regarding the Conceptual Programming Environment in [Mineau 1998] or elsewhere, except what has already been alluded to in the 'Literature Review' and 'Objectives' chapters. The following design goals have guided the development of pCG:

- Making those entities of primary interest to the developer, first class.
- Easy extensibility.
- Rapid development.
- Portability.

The first goal is satisfied by making concepts, graphs, actors, and processes first class in the language. Numbers, booleans, strings, lists, files, and functions are also first class types. All types may be passed as parameters to or returned from functions.

The second is made possible by exposing the run-time system as a set of Java classes to which attributes and operations (see the next section) may be added by way of methods following simple conventions. After recompilation of the relevant class, the new attributes or operations become available in the pCG language. Further, new types may be added to the language by creating subclasses of a particular Java class and creating instances using pCG's `new` operator. See 'Objects', 'Built-in Types', and 'User-defined Types' for more details.

The third goal is realised by:

- The interpretive nature of pCG. Since the core functionality of the language lies in the run-time system, there is little to be gained from compiling pCG to bytecodes or machine code. Like Perl, pCG's interpreter executes abstract syntax trees, an efficient intermediate representation. In the same way as Perl's key features such as regular expressions come in the form of pre-compiled library code, pCG's graph, actor, and process features are handled by pre-compiled Java classes. What this amounts to is that pCG programs can be executed with acceptable speed without the need for compilation, removing "compile" from the edit-compile-run cycle.

- The fact that all memory allocation in pCG is garbage collected[4] means that the programmer does not need to worry about chasing memory leaks.

- The built-in types provide the essential features required for working with CGs, Mineau's processes, and for general programming. The developer can spend more time focussing upon the problem itself, rather than how to represent it. This is a consequence of the first design goal.

'The Implementation of pCG' describes how pCG achieves the fourth goal of portability by virtue of its basis in Java.

pCG has been influenced by such languages as:

---

[3] Given its beauty and utility, Scheme could have been used as the basis for pCG, but not everyone is comfortable with Lisp dialects.

[4] This is courtesy of the Java implementation.

- Perl: first-class strings, dynamic arrays, the `foreach`, `system`, and `die` commands, but not by its purity since it has none;
- OO languages such as Java: objects and class libraries, the `instanceof` and other operators;
- Lisp: lists, dynamic typing, first class functions (lambda), `apply`;
- Prolog: assert, retract, variable binding as a result of unification;
- NewtonScript: Algol-like syntax combined with dynamic typing, *self*.

pCG does not make the statement that CGs are the best way to represent all aspects of a problem, instead permitting a hybrid approach based upon more familiar programming paradigms, not enforcing any particular one. Sowa has this to say regarding logic and procedural programming [Sowa 2000]:

> …logic can represent the same kinds of procedures as a programming language. The primary difference is that logic requires explicit relations or predicates to express the sequence, while procedural languages depend on the implicit sequence of the program listing. Ideally, programmers should use whatever notation they find easiest to read and write.

CG tools such as Synergy [Kabbaj 1999a] have been developed (see 'Literature Review') which support purely visual programming, but as Lukose remarks of his own formalism:

> …one must not be mislead into believing that the conceptual graphs and the executable conceptual structures are all good, and all encompassing, representational scheme that solves all modelling problems. [Lukose 1997b] [5]

In [Kremer 1997] a simple, general-purpose, visual flow-charting notation, Annotated Flow Chart (AFC), is mapped to MODEL-ECS in order to provide a knowledge modelling capability to non knowledge engineers in order to avoid the requirement to understand MODEL-ECS, a fairly complex language. Kremer et al show how selection and iteration constructs in AFC map to MODEL-ECS. In turn, MODEL-ECS itself implements selection and iteration constructs using CGs and special intrinsic constructs (see 'MODEL-ECS' section in the 'Literature Review' chapter).

The pCG language takes a different approach, by acknowledging that traditional programming languages and their constructs have a role to play. As a consequence, pCG has ordinary selection and iteration constructs and simple but powerful containers (lists), all of which are found in many modern languages, leaving to CGs the role of representing knowledge. If one wishes to apply a procedural operation to a KB of CGs, a `foreach` loop can be used to iterate over the set of graphs in the KB, invoking operations on each one. In short, CGs are used where knowledge representation is required, while traditional constructs are used where imperative or functional programming is most appropriate.

1.4 Informal Semantics

The pCG language is characterised by a few key semantic features. It is:

Dynamically typed: Values not variables determine type. Variables are not declared. The *is* operator can be used to determine the type of a value at run-time.

Lexically scoped: Functions and processes introduce a new lexical scope when invoked.

Object-based: Wegner (1987) says that an object is an entity that has a set of operations and a state which remembers the effect of those operations. He defines an object-based language as one which "…supports objects as a language feature" but cautions that the support of objects "…is a necessary but not sufficient requirement for being object-oriented. Object-oriented languages must additionally support object classes and class inheritance." [Wegner 1987].

On these grounds, pCG can be characterised as an object-based language since its fundamental types have state and operations on that state. However, pCG does not support the creation of new arbitrary object types within the language, but as noted above, it is possible to add new types to the language using Java, and to modify intrinsic types.

[5] Grammatical errors have been copied verbatim.

Objects in pCG have documented public *operations* (functions in objects) and *attributes*. In addition, each object "knows" its capabilities with respect to the standard in-built *operators* (`+`, `-`, `*`, `/` etc), an implementation detail which simplifies the design of the interpreter. See also 'Objects', 'Built-in Types', 'User-defined Types', and 'The Implementation of pCG'.

Minimal: The basic philosophy of pCG is that there should be few special statements and functions and that most computation should proceed through interaction with the fundamental objects of the language. Where a statement or function does appear in isolation from an object, it is because the internal state of the core interpreter itself must be modified, there is no object with which to associate it, or it eases use of the language.

Multi-paradigm: Apart from its object-based characteristic, pCG supports *imperative*, *functional*, and *declarative* styles of programming.

- *Imperative* programming is supported since there exists variables, assignment, operators, selection, iteration, and sequential execution.

- *Functional* programming is possible since functions are first class values and may be anonymous, closures may be created, and an `apply` operator is provided — in short, higher order functions are part of the language. Dataflow graphs (actors) may also be anonymous, and like functions anonymous actors may be recursive.

- *Declarative* programming is supported in the guise of process definitions and invocations, since one specifies rules containing pre and post conditions representing knowledge. The details of testing preconditions against the KB, and the assertion/retraction of post-conditions in the KB are left to the process execution engine. Processes could also be seen as supporting *constraint-based* programming (as found in languages like CLIPS [Sowa 2000]) since precondition graphs essentially specify constraints on the set of graphs in the KB.

The remainder of this chapter describes specific pCG features[6].

1.5 Lexical Conventions

Identifiers and reserved words in pCG programs are case sensitive (but see the `option` directive in 'Ad hoc Statements'). This includes concept and relation type names[7]. Whitespace (tabs, newlines, spaces) is ignored by the interpreter. Single and multiple line comments are supported, for example:

```
# This is a single-line comment.

// So is this.

/*
 * This is not.
 * The second line of this comment.
 */
```

The first form (`#`) was included so that environments supporting command shells with interpreter invocation lines[8] could specify pCG as the interpreter in the first line of a program, e.g.

```
#!/home/david/bin/pCG
```

The ASCII character set is supported in comments, strings, and graphs. String literals are double-quoted. Graph and concept literals are delimited by grave accents, e.g. `` `[Baby: *a Nicholas] [Small *b] (?a?b)` ``. For the syntax of CGIF and LF graphs, see [Sowa 1999] (and more recently `http://www.cs.nmsu.edu/~hdp/CGTools/cgstand/cgstandnmsu.html`).

---

[6] pCG code snippets appear in a fixed-width `Courier` font.
[7] Whether concept and relation types should be case-insensitive is debatable. In pCG, this has been done for consistency.
[8] For example, Unix shells, third party shell implementations for Win32.

Identifiers may contain alphanumeric characters and underscores, but must begin with a letter or underscore. Numeric values take the form: `n[.m]`, e.g. `2`, `45.789`, but exponential notation is not currently supported [9]. Numbers are stored internally as IEEE double-precision floating-point values.

1.6 Program Structure

A pCG program consists of statements, each of which is followed by a semi-colon [10]. Functions, and selection and iteration constructs consist of statement blocks, containing zero or more statements. Such blocks also appear within the definition of processes. See the relevant sections below for specific details.

Certain statements may only appear at the top-level:

- Concept and relation type lattice declarations;
- Named function definitions;
- Lambda (of the CG variety), actor and process definitions;
- Certain option directives (`LF`, `TRACE`).

There is no main function in a pCG program, and code outside of functions and processes is executed in the order it appears in the text. A pCG program may currently only span a single source file[11].

1.7 Run-time Environment

Before a pCG program runs, a syntax check is performed leaving only semantic errors to be detected at run-time. Graph and concept literals are not parsed until run-time however[12]. The interpreter currently stops after encountering the first error.

Each function and process invocation introduces a new lexical scope. A process invocation also introduces a new KB scope. So, there are two run-time stacks in pCG: one for variables and another for KBs. There is one of each at the top-level for global code execution.

Variables are not declared to have a particular type, and a given variable may take on different types of values in various parts of the same program. The first lexical appearance determines the scope of a variable. So, if a variable is first used at the top-level (i.e. outside of a function or process definition) it will have global scope, while a variable which appears only within a function or process definition has a scope which is local to that function or process. Local variables shadow those with the same name further up the scope stack, as would be expected from a lexically scoped language. A variable that is used before being assigned a value has a default value of `undefined`.

Variables and other named entities (e.g. functions; see 'Built-in Types') are stored in a symbol table on a per scope basis. All such entities are first class, i.e. they are values that can be passed to functions, returned from functions, assigned to variables, have attributes and associated operations and/or operators. All types implement the `is` and `+` operators, the latter for string concatenation.

KBs in pCG store concept and relation type hierarchies, and a set of graphs. Look-up is confined to the KB which is top-most on the stack. The KB contents of a process caller are copied to the invoked process's KB. This prevents anything except a process's parameterised output graphs from mutating the caller's KB, as per [Mineau 1998] [13]. The alternative is a proliferation of graphs in the caller's KB, meaningless outside of a given process, which would need to be retracted by some other means upon exit from that process. See 'Experiments' and 'The Implementation of pCG' for more details.

---

[9] This would be trivial to add to the grammar, but a number of features have not yet been added due to time constraints. These are documented in the appropriate sections. However, see the `string.toNumber()` operation in Table 1-2.

[10] An isolated semi-colon is the *empty statement*.

[11] However, a Perl-like `require` directive is planned.

[12] Concept and graph parsing relies upon Notio [Southey 1999] functionality.

[13] The `option export` directive provides a way to circumvent this restriction. See 'Ad hoc Statements'.

1.8 Objects

All values in pCG are *objects* with associated *attributes*, *operations*, and *operators*. The term *value* and object will be used interchangeably. Some attributes may appear on the left hand side of an assignment statement. All may appear in expressions. All objects have an associated type attribute with a string value (itself a type) indicating the name of a value's type. For example, the following code prints the word `number`:

```
x = 42;
println x.type;
```

An operation is the equivalent of a method in Java or a member function in C++. An operator may be monadic or dyadic (unary or binary). Given a string assignment such as:

```
s = "Hello, world!";
```
[14]

the next two lines of code show an operation with two parameters and a dyadic operator, respectively:

```
s = s.substring(1,4);
s = s + "Take me to your leader.";  // string concatenation
```

Operations may be overloaded, which is to say that two or more operations may have the same name, but different formal parameter lists [15].

1.9 Built-in Types

1.9.1 Attributes and Operations

The pCG language has the following intrinsic types: number, boolean, string, list, file, concept, graph, and a special value: `undefined`. Function, lambda, actor, and process definitions also yield values with particular types. There is also a special type for Knowledge Bases, an instance of which is made available by the interpreter in the current scope. This is considered to be an internal type only. See 'Ad hoc Variables' for details.

Table 1-1 shows the attributes associated with each built-in type. An attribute name is followed by a colon, a type, and an optional "+" if the attribute is mutable, i.e. can appear on the left hand side of an assignment statement.

Type	Attributes	Comments

actor	`type: string`
	`name: string`
	`defgraph: graph`
	`sourceconcepts: list`
	`sinkconcepts: list`
		• May be "anonymous".
		• Defining graph.

boolean	`type: string`	

concept	`type: string`
	`label: string`
	`designator: number, string, boolean +`
	`descriptor: graph +`
		• pCG value type.
		• Concept type.

file	`type: string` `kind: string`	• "reader" or "writer"

function	`type: string`
	`name: string`
	`argcount: number`
		• May be "anonymous".

graph	`type: string` `concepts: list`	• List of concept type values.

[14] In pCG, there is only one kind of double or single quote character for strings and referents, respectively, contrary to what is displayed in this document.

[15] Currently pCG differentiates only on parameter list length, not types, but this can easily be rectified.

	`relations: list`
	
	`actors: list`
		• List of lists of relation type label (string), input, and output argument lists. This includes actors which are a special kind of relation.
		• Same as above, but only for actor relations.

lambda	`type: string`
	`name: string`
	`defgraph: graph`
		• Defining graph.

list	`type: string` `length: number`	
number	`type: string`	
process	`type: string` `name: string`	
string	`type: string` `length: number`	

Table 1-1 Attributes for intrinsic types.

Table 1-2 shows the operations associated with each built-in type. An operation name is followed by parenthesised ordered parameter types (if any), an arrow (⇒), and a return type. The symbol "|" between return types indicates options. If "+" is appended, the object on which the operation is invoked will be mutated, i.e. its state will change. A type of `t` or `t'` indicates that any type may be passed to or returned from the operation in question.

Type	**Operation**	**Comments**
actor	None	
boolean	None	

concept	`restrict(concept)⇒boolean +`
	`copy()⇒concept`
	`nocomments()⇒concept`
	
	`isGeneric()⇒boolean`
	`isContext()⇒boolean`
		• By type and/or referent.
		• Omits comments from a concept, e.g. when created with a tool such as CharGer.
		• Has no referent?
		• Has a descriptor?

file	`readline()⇒string | undefined`
	`readall()⇒list | undefined`
	
	`readGraph()⇒graph | undefined`
	
	`write(string)`
	`writeln(string)`
	`close()`
		• Reads all lines from a file. Text files are assumed in pCG.
		• Reads a CGIF or LF graph from a file.
		• Writes a string to standard output.
		• Adds a linefeed after writing.

function	None	

graph	`copy()⇒graph`
	`nocomments()⇒graph`
	
	`project(graph)⇒graph | undefined +`
	
	`join(graph)⇒graph | undefined +`
	
	`joinAtHead(graph)⇒graph |`
	`undefined +`
		• Omits comments from a CGIF graph, e.g. after creation with a tool such as CharGer.
		• Projection operator. See 'The Implementation of pCG' for details.
		• Joins this graph to specified graph on first compatible concepts found in each graph.
		• Joins this graph to specified graph at the head concept (first argument of first relation) of each.

lambda	None	

list	`hasMember(t)⇒boolean`	• Looks for a value in the current list and returns true or false, but does not into recurse list members.
	`member(t)⇒t'`	• Looks for a value in this list, or in each sub-list of this list, recursively. If found, the sub-list within which it is embedded is returned. Note that this may be the outermost list.

	`prepend(t)⇒list +`
	`append(t)⇒list +`
	`merge(t)⇒list +`	

number	`pow(number)⇒number`
	`sqrt()⇒number`
	`sin()⇒number`
	`cos()⇒number`
	`tan()⇒number`
	`floor()⇒number`
	`ceil()⇒number`
	`round()⇒number`
		• Standard math functions.

	`inc()⇒number +`	• Increment.
	`dec()⇒number +`	• Decrement.
	`chr()⇒number`	• Return a string which this number represents in ASCII.
process	`None`	
string	`substring(number, number)⇒string | undefined`	• Start and end parameters are ≥ 1 and ≤ length of string.
	`substring(number)⇒string | undefined`	• Start parameter is ≥ 1 and ≤ length of string. Sub-string from start value to end of string is returned.
	`index(string) ⇒number`	• If string not found, -1 is returned.
	`toBoolean()⇒boolean`	• Anything but string "true" (case insensitive) is considered false.
	`toNumber()⇒number | undefined`	• If the string contains a valid number, a number object will result, otherwise `undefined` will be returned [16].
	`toGraph()⇒graph | undefined`	• If the string contains a valid graph, a graph object will result, otherwise undefined will be returned.
	`replace(string, string)⇒string +`	• Replaces all occurrences of the first single character string with the second and returns the result.

Table 1-2 Operations for intrinsic types.

Operators are considered below. Note that undefined attributes evaluate to `undefined`, just as uninitialised variables do. Operation or operator failure may also give rise to the `undefined` value.

The means by which attributes, operators, and operations may be added to intrinsic objects is discussed in 'The Implementation of pCG'.

[16] Indeed, this is a way to sneak an exponential format number in the back door since underlying Java class library code handles this, not pCG's lexical analyser.

Only those attributes, operations, and operators required for the current thesis work have been implemented at the time of writing.

1.9.2 Concept and Graph Values

In pCG, concepts are existentially quantified by default, and no other explicit form of quantification is currently supported. Concept referents (see the draft CG ANSI standard [Sowa 1999]) may otherwise be:

- *Literal designators* of type number, string (single or double quoted) , or boolean[17];
- *Locator designators*, e.g. `#123658` or `#Nicholas`;
- *Variable designators*. These must currently be quoted and are distinct from CGIF defined and bound variables, for example in the CGIF graph:

  ```
  [Number: *a '*n'][Number: *b '*result']<sqr?a|?b>;
  ```

  `*a` and `*b` are defining variables, `?a` and `?b` are bound variables for identifying concepts, while `*n` and `*result` are variables representing designator values. Such variables derive from the original actor notation of [Sowa 1984] and have no basis in the proposed ANSI standard of [Sowa 1999]. The terms *variable designator,* and *referent variable* are used interchangeably in this document.

- *Descriptor graphs*, especially useful in contexts passed as process parameters.

Graphs and concepts may be specified as literals or read from a file in the CGIF or LF[18] formats. Complex graphs may be made available to pCG by using a graph editor such as CharGer [Delugach 1999] and saving them as CGIF to a file.

1.10 User-defined Types

By creating a subclass of the abstract Java class `cgp.runtime.Type`, invoking a setType() method in the new class's constructor, and optionally overriding any or all of a number of the base class's methods, a new type is made available to pCG. The class `cgp.runtime.Type` provides default behaviour for all pCG operators. The means by which this is accomplished is described in 'The Implementation of pCG'. If a new type overrides `java.lang.Object`'s `toString()` method, string concatenation is automatically available in the form: `t + s or s + t`, where `t` is the new type and `s` is a string value. One can for example say:

```
x = "This is a graph: " + g;
```

where `g` is a graph value, the result being a string value assigned to `x` with the shown string literal preceding a CGIF graph.

The distribution (see 'Availability and Requirements') contains examples of new types, e.g. `Window` and `Util` [19]. The first adds to pCG the ability to open windows, draw on a window's canvas via Turtle Graphics [Abelson 1992], and display text or arbitrary GIF or JPG images. The second class is a starting point for additional miscellaneous useful functions such as `sleep` and `random`. Both were used in a solution of the Sisyphus-I room allocation problem [Linster 1999], described in the 'Experiments' chapter. Table 1-3 details the operations in these classes. Neither contains any attributes other than the default `type` attribute. It is important to realise that once implemented, these types are no different from pCG's intrinsics except that they must be created using pCG's `new` operator [20].

---

[17] Boolean values (`true`, `false`) must be double-quoted strings;
[18] LF handling relies upon Notio's functionality and is currently unreliable.
[19] See the `cgp/cgp/runtime/newtypes` directory.
[20] Currently, no parameter may be passed at object creation time, so the appropriate attributes must be set later if necessary.

Type	Operation	Comments

Util	`sleep(number)`
	
	`random(number)⇒number`
		• Sleep for the specified number of seconds or fraction thereof.
		• Return a random integer (as a number type) in the range `-n-1 ≤ 0 ≤ n-1`. This uses a random number generator which is seeded by default from the system's time.

Window	`open(string, number, number, number, number)`
	
	`close()`
	
	`setColor(list)`
	
	`drawLine(number, number, number, number)`
	
	`drawText(string, number, number)`
	
	`drawImage(string, number, number)`
	
	`moveTo(number, number)`
	
	`lineTo(number, number)`
	
	`turn(number)`
	
	`walk(number)`
		• Opens a window with the specified title, left and top coordinates, width, and height (in that order).
		• Closes the window associated with this `Window` object.
		• Sets the current window's colour using the specified red, green, and blue components.
		• Draws a line in the current window from `(x1,y1)-(x2,y2)`, using the parameters in that order.
		• Draws a string in the current window at the specified coordinates.
		• Draws an image in the current window at the specified coordinates. The first parameter is the URL of the image.
		• Turtle graphics. Move to the specified location in the current window.
		• Turtle graphics. Draw a line to the specified location in the current window.
		• Turtle graphics. Turn the turtle by the specified number of degrees (left is negative).
		• Turtle graphics. Walk the turtle in the current direction [21]. If a negative value is specified the turtle will move in reverse.

Table 1-3 Sample user-defined types supplied with the distribution.

1.11 Operators

Unlike attributes and operations, the set, precedence, and associativity of operators is fixed in pCG. Table 1-4 lists these operators for built-in types from lowest to highest precedence, specifying permissible operand types, with `t` indicating any type. All pCG operators are left associative, e.g. `8-4-2` is 2 not 6. These operators are a subset of those found in Java.

Operator	Operand types	Comments
`or`	`boolean`	• Logical or. Not short-circuit.
`and`	`boolean`	• Logical and. Not short-circuit.

[21] The turtle's current direction is specified by the `turn` operation, but defaults to 270 degrees, or north.

`> < >= <=`	`number, string`	• Relational operators.
`== !=`	`number, string, boolean, concept, list`	• More relational operators.
`is`	`t is string` [22]	• Type equivalence operator.
`+`	`number + number, string + t, t + string`	• Numeric addition and string concatenation.
`-`	`number`	• Numeric subtraction.
`* div mod`	`number`	• Numeric multiplication, division, modulus.
`-`	`number`	• Unary negation.
`not`	`boolean`	• Logical complement.
`[ ]`	`list`	• Array indexing.
`.`	`t`	• Attribute and operation access. [23]

Table 1-4 pCG operators

Although not strictly an operator, one comment needs to be made regarding assignment: it does not copy what is being assigned to the variable on its left hand side. The assigned variable merely references the object on the right hand side of the assignment operator. While some types have a `copy` operation, e.g. `concept`, `graph`, this is not intrinsic to pCG.

1.12 Selection and Iteration

The pCG language has the following basic control constructs: `if`, `while`, and `foreach`, where ellipsis represents one or more statements, and square-bracketing indicates optional parts.

```
if…then…end [else…end]

while…do…end

foreach…do…end
```

for example:

```
foreach n in {1,2,3,4,5} do
  println sqr(n);
end

foreach con in myConceptList do
  println con.designator;
end
```

Examples of these familiar constructs abound in the supplied examples.

1.13 Ad hoc Statements

As mentioned earlier, pCG has several statements and functions that either:

- Do not fit neatly into one of the intrinsic objects, although arguably some could be shoe-horned into one or more;
- Or, must have special access to or modify the internal state of the interpreter.

[22] For intrinsic types, the type name string does not have to quoted.
[23] Due to a grammar bug in pCG, a statement such as: `c = g.concepts[0]` must currently be written: `c = (g.concepts)[0]`

Table 1-5 describes these statements. The notation `(...)*` is a Kleene Closure, indicating zero or more occurrences of some pattern. The text "`id`" means that an identifier is expected, `expr` means that an arbitrary expression is expected. In some cases, specific types are specified for values.

Syntax	Description
`activate`	Actually an intrinsic function which activates a dataflow graph or process invocation graph. See also the 'Lambda and Actors' and 'Processes' sections in this chapter.
`apply function list`	Applies the function object to the list of arguments. This can also be invoked in an expression, permitting a return result to be captured.
`assert graph`	Asserts a graph in the currently active KB.
`exit [number|string]`	Terminates a program with or without a numeric or string value. The former is a result code that becomes available in the invoking shell (assuming one exists). If instead a string is supplied, it is sent to standard error before the program exits with a result code of 1. If no value is supplied, the program is terminated with a result code of zero.
`last`	Breaks out of the enclosing `while` or `foreach` loop.
`new`	This is discussed in the section 'User-defined Types', and is in fact an intrinsic function rather than a statement since it returns a new object instance.

`option id (, id)*`
 	
`An id may also be followed by "=" and a double quoted string.`
	An arbitrary interpreter directive. In this statement, the case of identifiers is *ignored*. The scope of an option depends upon where it occurs, e.g. at the top-level, an option's scope is global, but if associated with a process rule, the scope is limited to that rule. An option's scope may even be limited to a single post-condition graph. Five options are currently implemented, the first two at the top-level, the next two in process rules, and the final one in process rules or associated with single post-condition graphs:
	
	• `option LFout`: string representation of graphs in LF not CGIF.
	• option CGIFparser = "<string>": declares the fully-qualified Java class to be used for CGIF parsing instead of the default CGIF parser. For example, "cgp.translators.CGIFParser" is compliant with the June 2001 CG Standard. While the latter handles all of the new syntax, the semantics of CG lambda expressions, disjuncts, conjuncts, and structure descriptors are not yet implemented owing to a lack of time. On the other hand, this parser does correctly handle relations of type GT, using them to populate the current KB's type hierarchy.
	• option CGIFgen = "<string>": declares the fully-qualified Java class to be used for CGIF output instead of the default CGIF generator. For example, "cgp.translators.CGIFGenerator" is compliant with the June 2001 CG Standard.
	• `option trace`: provides verbose output from the interpreter, starting with a Lisp-style syntax tree of the parsed program; useful for debugging or reporting errors to the author.
	• `option exportretract`: all post-condition graphs which correspond to retractions will affect the caller's KB, not the local KB.
	• `option exportassert`: all post-condition graphs which correspond to assertions will affect the caller's KB, not the local KB.
	• `option export`: all post-condition graphs will affect the caller's KB, not the local KB.

`print expr`	Prints the expression on the standard output.
`println expr`	Same as `print` but adds a newline [24].
`retract graph`	Retracts a graph from the currently active KB.
`return [expr]`	Exits functions and processes with or without a value.
`system string`	Executes an arbitrary external command and is therefore operating system dependent. This may either be invoked as a statement or as a function (part of an expression) if the exit code for the invoked process is required.

Table 1-5 Ad hoc Statements in pCG.

1.14 Ad hoc Variables

In addition to user-defined entities being entered into the symbol table for the current scope, a number of special variables are also entered by the interpreter. By convention, these special variables begin with an underscore and consist of upper-case letters. The "me" variable is an exception to this rule, because it is considered to be a special, silent parameter to each invoked function, somewhat like `this` in OO methods, but representing the function object itself. Table 1-6 details these special variables.

Variable and Type	Description
`_ARGS: list`	The command-line arguments passed to the pCG program.
`_ENV: list`	Environment variables, or more particularly, Java system properties, such as home directory, platform-specific path delimiter, and so on.

`_KB` [25]	This is inserted into the local scope of the currently executing process, and represents the local KB for the process. It evaluates to a special type with the following attributes:
	• `graphs: list`
	• `concepttypes: string`
	• `relationtypes: string`
	• `corefvars: list`
	One can either obtain these values individually, or print the value of _KB as a whole. The purpose of this variable is to provide information about updates to the currently active KB, and to provide access to its contents. One could for example, iterate over the graphs in the current KB thus, where target is some graph:
	`foreach g in _KB.graphs do`
	`  p = target.project(g);`
	`  if p != undefined then println p; end`
	`end`
	which prints successful projections of the graphs in the current KB onto a target graph.
	Every pCG KB has some default concept types such as `Number`, `String`, and `Boolean`, although they are not really necessary since such types can be added as needed in a program, and there is currently no conformity relation in pCG.
	The `corefvars` attribute is a list of bound variable designators as described in 'Concept and Graph Values' in the 'Built-in Types' section. These are bound as a result of actor activity in processes,

[24] Since pCG does not yet support escape sequences such as "\n" for newline, this is necessitated. A future revision may replace `println "foo bar"` with print `"foo bar\n"`. Once again, time constraints have not permitted this at time of writing.
[25] See section 'Built-in Types' regarding type.

	and concept restriction (e.g. during a projection operation in a process). They may be used by subsequent actors, assertions, or retractions. See 'The Implementation of pCG' for details of the algorithms [26]. Since the conformity relation is not implemented in pCG, the KB's catalog of individuals is unused.
`_MATCHES: list`	For each rule in an executing process, the list of successfully matched precondition graphs is made available via this locally scoped variable.
`me: function`	This silent parameter to every invoked function, is the function object itself. It is useful for obtaining a function's name or arity in generic error messages via the function's `name` or `argcount` attributes. It may also be used for referring to an anonymous function. See 'Functions' section.

Table 1-6 Ad hoc Variables in pCG.

1.15 Functions

Functions in pCG may be named or anonymous. The syntax of a named function definition is:

```
'function' name parameter-list block
```

An example of a named pCG function definition is:

```
function sqr(n)
  return n*n;
end
```

The definitions of functions, lambda, actors, and processes must appear before their first invocation.

Parameters may take on any value during function execution, just as variables can. A function may be invoked as a statement or as part of an expression, as in C. In the former case, any return value is ignored, e.g.

```
sqr(4);
n = sqr(4);
```

Functions may of course, be recursive. Here is the stereotypical factorial function in pCG.

```
function fact(n)
  if n < 1 then
    return 1;
  end else
    return n*fact(n-1);
  end
end
```

A pCG function may also be anonymous, e.g.

```
println apply function (n)
                if n < 1 then
                  return 1;
```

[26] The need for such a capability is a result of the assertion in [Mineau 1998] that coreference should be global in the process mechanism. That paper also suggests that the activity of a process must be cleaned up when the process exits. Accordingly, this mechanism in pCG is *local* to a process invocation, but the variables can be captured by the concepts of output parameter graphs, thus satisfying Mineau's requirement. It is worth noting that this notion of coreference is not the same as that of [Sowa 1999], but derives from the dataflow graph domain.

```
  end else
    return n*me(n-1);
  end
end       {7};
```

The difference here is that the function definition has no name following the `function` keyword, and appears as part of an expression, modifying the above syntax to:

```
'function' parameter-list block
```

This example illustrates the use of the built-in `apply` function which takes a function and list as arguments. It also shows that anonymous functions may still be recursive, courtesy of the silent `me` parameter which is passed to all functions. This idea followed the author's refinement of the actor invocation mechanism which led to recursive anonymous actors (see 'Lambda and Actors'). Given that the `me` parameter had previously been added to pCG, anonymous recursive functions were a natural outcome. Other languages that support this capability were subsequently found. For example, the Joy and R programming languages [von Thun 2000] [Bates 1997]. [Tierney 1997] shows techniques for anonymous recursive functions in a Lisp dialect, including the lambda calculus's Y combinator [Friedman 1989] [Louden 1993] [Tierney 1997]. None of these mechanisms appear to be simpler than pCG's.

As an aside, since named function definitions are entered into the top-level symbol table, the following is also possible:

```
println apply fact {7};
```

By itself this is not useful, but a function such as `map` may be written to apply a function to a sequence of values, yielding another sequence (i.e. a list). This `map` function would take a function and a list of lists as an argument, so for example, one could map the factorial function to a list of values thus:

```
myList = map(fact, {{0,1,2,3,4,5,6,7,8,9}});
```

to yield a list of factorials of the first n positive integers. Listing 1 in Appendix B shows an implementation of `map` in pCG.

Functions in pCG are also *closures* [Louden 1993] since the environment in which anonymous functions are defined is carried with the definition, e.g.

```
function mkCounter(n)
  count = n;
  return function() count=count+1; return count; end;
end

c1 = mkCounter(3);
c2 = mkCounter(10);
println c1 + " is a closure.";
println c2 + " is a closure.";
foreach i in {1,2,3,4,5} do
  println c1() + " " + c2();
end
```

This code creates two different "counter" closures, one of which has an initial value of `n` of 3, and the other of 10. Each counter function is invoked several times in order to show that they have separate copies of `n`. The following output results:

```
function anonymous; arity 0 is a closure.
function anonymous; arity 0 is a closure.
4.0 11.0
5.0 12.0
6.0 13.0
```

```
7.0 14.0
8.0 15.0
```

pCG's anonymous functions are the equivalent of `lambda` [Louden 1993] in such languages as Scheme, making functional programming possible in pCG.

1.16 Lambda and Actors

The 'Background' chapter's 'Lambda Expressions' section described Sowa's use of Alonzo Church's lambda calculus [Louden 1993] for types in CST. Recall the following example given in that section:

```
MaleBaby = [Baby: λ] -> (Chrc) -> [Gender: Male]
```

This can also be written as:

```
type MaleBaby(*x) is [Baby: ?x] -> (Chrc) -> [Gender: Male]
```

In pCG one can express this as:

```
lambda MaleBaby(x) is
             `[Baby: *a '*x'][Gender: *b "Male"](Chrc?a?b)`;
```

which has elements of the notation used in [Sowa 1999] and [Sowa 2000]. In this example, CGIF is being used to specify the defining graph. One can then say:

```
g = MaleBaby("Nicholas");
println g;
```

to yield the graph:

```
[Baby: *a "Nicholas"][Gender: *b "Male"](Chrc?a?b)
```

The essential point of similarity between the kind of lambda expression facility described in the previous section 'Functions', and the mechanism described above is that in both cases, there is β-reduction taking place to bind parameters to free variables in a copy of the defining graph [Louden 1993]. In pCG, `MaleBaby` is not added to the concept type lattice, making this implementation of CG lambda expressions of limited utility except as a kind of macro capability. However, it forms the basis for actors in pCG.

Consider the following function and actor definitions in pCG:

```
function sqr(n,m)
  nVal = n.designator;
  if not (nVal is number) then
    exit "Input operand to " + me.name + " is not a number!";
  end
  m.designator = nVal*nVal;
end

actor SQR(x) is `[Num:*a'*x'][Num:*b'*y']<sqr?a|?b>`;
```

The actor defines a graph (a dataflow graph) with an active element — actor node or executor — as discussed in the 'Actors' section of the 'Literature Review' chapter. This executor is ultimately defined in terms of a pCG function, `sqr`, which takes two concepts as parameters, a source and a sink concept. The sink concept's referent (designator) is mutated according to the square of the source concept's designator, which is first tested to ensure it is a number. One way to invoke this actor is to write:

```
g = SQR(4)
```

which results in a mutated copy of the defining graph being assigned to g:

```
[Num:*a 4][Num:*b 16]<sqr?a|?b>
```

A point of departure in pCG's implementation of actors compared to [Sowa 1984] and [Lukose & Mineau 1998] is that only the input parameter is specified in the actor definition's parameter list, and then only for the purpose of binding. The actor mechanism can otherwise determine the correct order in which to pass concepts to a particular executor, so long as the arcs are correctly ordered [27]. Assuming the executor (a function or other dataflow graph) performs correctly, the returned mutated graph copy will have appropriately mutated sink concepts [28]. The defining graph in the SQR example still has a sink concept referent variable to indicate that the designator is unbound prior to actor invocation.

The above invocation code is not useful for when an actor appears in a process rule's precondition. An invocation graph may be constructed and the graph activated directly, e.g.

```
n = 4;
mySqrGrStr = "[Num:*a " + n + "][Num:*b'*y']<sqr?a|?b>";
g = activate mySqrGrStr.toGraph();
```

Notice that no actor definition is required here. The graph is constructed as a string, the source concept's referent is bound in that string, and the string is converted to a graph which is then activated, returning the mutated graph copy. This is akin to actor invocation that occurs during process execution, except that the process engine implicitly activates precondition graphs containing actors. The actor node — `<sqr?a|?b>` — refers to an executor called `sqr` which takes as input a concept referred to by the bound variable `?a`, and mutates another concept, `?b`.

What if the actor is recursive? To what entity should the executor responsible for the recursive step refer? The nature of a recursive dataflow graph is that it must invoke another actor, itself in the case of direct recursion. A reasonable solution is to supply an actor definition which names the actor to be invoked. Another solution is to employ a special self-referential actor node such as `self` [29], avoiding the need for a definition. The 'Experiments' chapter gives two versions of a recursive factorial dataflow graph. The first uses an explicit actor definition, the second uses a `self` node.

Although an explicit definition works, why should recursive actors require a special case? Further consideration of this problem led the author to revisit the Y combinator. This function permits recursive functions, without explicit definition of the recursive function, by repeatedly generating the body of the next recursive invocation of the function, then invoking it for each step. Since each invocation of a dataflow graph involves copying the defining graph, before binding its source concepts and executing it, there is a distinct similarity to the Y combinator. This is more so than for recursive anonymous function invocation in pCG (or any language of which the author is aware) since there is no sense in which the body of a function is copied at invocation time.

For more substantial actor examples (recursive, multiple executors per graph), see the 'Experiments' chapter. The role of actors in process rule preconditions is also revealed in that chapter. For details of the dataflow algorithm, see 'The Implementation of pCG' chapter.

1.17 Processes

1.17.1 Definition

A process definition in pCG is similar to that proposed in [Mineau 1998]. Here is Mineau's formal definition:

```
process p(in g1, out g2,...) is {r_i = <pre_i, post_i>, ∀_i∈[1, n]}
```

[27] CharGer can help with this by permitting explicit arc numbering, ensuring the correct ordering in the generated CGIF. [Mineau 1998] acknowledges the importance of arc ordering in process invocation graphs also.

[28] In order to make the mechanism more robust, it may be prudent to add an attribute to the concept type which indicates whether a concept is a sink or a source, so that the underlying function could check that it was not mutating a source concept, for example.

[29] `me` and `self` were defined independently and for different purposes, but arguably their names should be reconciled for consistency.

This translates to the following syntax in pCG:

```
'process' name '(' in | out parameter [, …]')'
   ['initial' block]
   ('rule' ident
     [option-list]
     'pre'
       ['action' block] (['~'] pre-condition)*
     'end'
     'post'
       ['action' block] (post-condition [option export])*
     'end'
   'end')*
'end'
```

What this essentially says is that a process has a name and a list of `in` and `out` parameters, followed optionally by a block of code for miscellaneous initialisation purposes, followed by a set of zero or more rules. Each rule consists of an arbitrary (and hopefully descriptive) identifier, optionally followed by a list of one or more options (see 'Ad hoc Statements') for the rule, and a pre and post condition section. A precondition section consists of an optional action block and zero or more possibly negated graph expressions. A post-condition section consists of an optional action block and zero or more possibly exported contexts.

1.17.2 Code Blocks

An initial or action block consists of arbitrary pCG code. The intent of such blocks is to aid in the debugging of processes, to provide useful output during process execution, or to combine procedural and declarative programming styles. Such code may also be used to construct graphs which are subsequently used in precondition graph matching, or post-condition graph assertion or retraction.

1.17.3 Pre and Post Conditions

A precondition is an arbitrary graph expression. If preceded by a '~' character, the sense of the match for that graph is reversed. A post-condition is a context with one of the following concept types: PROPOSITION or ERASURE, corresponding to assertion or retraction.

1.17.4 Parameters

An input parameter must be a concept with a descriptor graph, i.e. a context, and have one of following concept types: PROPOSITION or CONDITION. The descriptor of a PROPOSITION context is asserted before process execution starts and the intention is that such a graph will act as a trigger which (along with possibly other asserted graphs) causes a suitable rule to fire. The alternative is to assert a trigger graph prior to invoking a process, but this may pollute the caller's KB unnecessarily. A CONDITION context's descriptor graph is added to the precondition list of the first rule of a process, as per [Mineau 1998]. Also in accordance with [Mineau 1998], output parameters are appended to the post-condition list of the last rule. When this rule is reached, the process is terminated after asserting or retracting graphs in the caller's KB (rather than the local KB) depending upon whether the context's concept type is PROPOSITION or ERASURE, respectively. Arguably, this should be refined such that all final rule post-conditions modify the caller's KB, not just the output parameters, since once that rule is finished, the process will terminate, resulting in the loss of any updates to the local KB. Currently however, only the final rule's output parameters update the caller's KB, unless a directive such as `option export` (see 'Ad hoc Statements') is used.

Contexts with concept types PROPOSITION, ERASURE, and CONDITION are essentially being used as a packaging mechanism (see 'Appendix A — Contexts'), as a means by which to transport graphs to a different KB. Sowa's notion of import and export of concepts in the 'Literature Review' OO CGs section is not dissimilar.

PROPOSITION and ERASURE context types are also used in the body of post-conditions to distinguish between assertions and retractions. An alternative would have been to have each pre-condition block subdivided into assertion and retraction blocks.

1.17.5 Invocation

A process may be invoked in a similar manner to an actor. What follows is a trivial but complete process definition:

```
process p(in trigger, out finalAssertion)
  rule r1
    pre
      `[Line:*a'#1'](to_do?a)`;
    end

    post
      `[ERASURE:[Line:*a'#1'](to_do?a)]`;
      `[PROPOSITION:[Line:*a'#2'](to_do?a)]`;
    end
  end // rule r1

  rule r2
    pre
      `[Line:*a'#2'](to_do?a)`;
    end

    post
      `[ERASURE:[Line:*a'#2'](to_do?a)]`;
      `[PROPOSITION:[Line:*a'#3'](to_do?a)]`;
    end
  end // rule r3

  rule r3
    pre
      `[Line:*a'#3'](to_do?a)`;
    end

    post
    end
  end // rule r3
end
```

The following invocations of this process are equivalent.

```
// Explicit invocation. Note that while concepts are
// being passed as parameters, they are only acting as a
// vehicle (contexts) for the graphs of interest.
x = p(concept `[PROPOSITION:[Line:*a'#1'](to_do?a)]`,
      concept `[PROPOSITION:[Foo:'on you']]`);
```

and

```
// Invocation by graph activation.
g = `[PROPOSITION:*a[Line:*b'#1'](to_do?b)]
     [PROPOSITION:*c[Foo:'on you']]
     <p?a|?c>`;
x = activate g;
```

Notice that the result of process invocation is assigned to a variable in both cases. This is because it is possible for a code block in a process to contain a return statement. If this code block is reached, the process will terminate and possibly return a value to the caller. The actor node in this graph — `<p?a|?c>` — refers to a process `p` with an input parameter referred to by the bound variable `?a`, and an output parameter, referred to by `?c` [30]. Note that the names of parameters in the formal parameter

[30] While concept argument ordering and actual parameter context type are critical in order for formal and actual parameters to match, pCG currently ignores arc directionality. Such a check should be added for completeness. CGIF enforces that at least one output argument exist in an actor node.

list of a process are not used anywhere within the body of the process, serving only as useful mnemonics.

1.17.6 Further Comments Regarding Preconditions, Post-conditions, and Contexts

A rule's preconditions consist of a conjunction of graphs, all of which must be matched against the local KB for the rule to fire, i.e. for the post-conditions to be acted upon.

When a graph is preceded by a tilde ("~"), it must not be possible for that graph to be matched against the local KB, if further precondition processing is to occur for the rule in question. So, the purpose of this special case of negation is not to assert a negated graph in a KB — to assert the falsity of some fact — but to reverse the sense of a graph match [31]. Such "negation as failure" is discussed in [Bos 1997].

pCG's model of KB update is a simple one of assertion and retraction, and utilises ERASURE and PROPOSITION concept types in post-condition contexts to determine which operation to apply to the KB. pCG's erasure is akin to the *erasure* Rule of Inference of [Sowa 1999] which generalises a graph to the blank graph in a positive context. Conversely, a proposition post-condition is akin to the *insertion* Rule of Inference of [Sowa 1999] which specialises the blank graph in a negative context.

The use of ERASURE seems theoretically safe, given that there are no explicit negative contexts in pCG [Sowa 1999]. pCG's use of PROPOSITION has the desired effect of asserting new facts on the sheet of assertion, i.e. in the current KB's graph set, but since the creation of a negated context is not recognised in pCG, such assertion may be theoretically questionable.

In [Esch 1994] we find the suggestion that the "…basic thing to remember about contexts and coreference is that it closely models scope of variables in block structured languages." One interpretation of a local process KB in pCG is that its graph set represents a positive outermost context. But this is an imperfect comparison, since a process KB, like a process or function scope is ephemeral and additionally, contains a copy of the caller's KB as a basis from which to begin process execution. Alternatively, one might argue that pCG's stack of KBs itself represents nested contexts.

These issues relating to assertion, retraction, and contexts in pCG and the process mechanism (as specified in [Mineau 1998]) require closer examination to determine the extent to which they are consistent with CST. See also 'Future Work'.

There are precedents for pCG's notion of assertion and retraction, in the Prolog and CLIPS languages for example [Sterling 1986] [Sowa 2000] [Giarratano 1989]. Sterling and Shapiro provide pause for caution in the use of assertion and retraction in Prolog however:

> Asserting a clause is justified, for example, if the clause already logically follows from the program. In such a case adding it will not affect the meaning of the program…Similarly, retracting a clause is justified if the clause is logically redundant. In this case retracting constitutes a kind of logical garbage collection, whose purpose is to reduce the size of the program.

While these comments probably apply to pCG, the author contends that this is a non-trivial issue, further consideration of which is beyond the scope of this thesis. See also the 'Future Work' chapter.

1.17.7 The Process Engine

When a process is invoked, the first rule is retrieved, and each graph of the rule's precondition is tested in turn. If one does not match, matching for that rule is discontinued and the next rule is retrieved. This is a kind of short-circuit feature, analogous to C's && and || operators. If no matching rule is found, the process terminates. If one is found, the post-condition contexts of the matched rule are retrieved. For each post-condition, its descriptor graph is either asserted or retracted — depending upon the concept type of the context — from the local KB, or the caller's KB if one of the export options is active. The process engine then starts again at the top of the ordered rule collection, and attempts to find a matching rule during the next cycle.

---

[31] Note that this is currently only available in the graphs defined in rule preconditions, not in CONDITION input parameters.

Assuming the pattern of rule firings does not lead the process into an infinite loop, and it finally reaches the last rule, the process will terminate, updating the caller's KB based upon any output parameters. See 'The Implementation of pCG' for further details of the process algorithm.

At process invocation time, the optional initial code block is executed. Before a rule's precondition matching begins, the optional block of arbitrary pCG code is executed. Another optional block is executed before every rule's post-conditions are applied. These procedural aspects of pCG are pragmatic additions to Mineau's processes.

The core process mechanism of pCG is consistent with the existential conjunctive subset of logic — with some embellishments such as precondition negation — in which only existential quantification and conjunction are required [Sowa 2000].

1.17.7.1 Implication

In the 'Background' chapter, double-negation as implication was introduced, but pCG does not currently support this. However, the precondition of a rule can be considered to be the *antecedent* of a *production rule*, while that rule's post-condition is the corresponding *consequent*, providing a form of *modus ponens*. [Sowa 2000]

1.17.7.2 Disjunction

A process's rule set is like a sophisticated case statement in the sense that when one rule fires instead of another, a choice of one from many has been made, essentially a logical *or* operation, or *disjunction*. This is contrasted with the fact that each rule's precondition constitutes a *conjunction* of graphs.

1.17.7.3 Forward Chaining

The pCG process engine is essentially a forward chaining production rule system, similar to the one found in the CLIPS[32] language [Giarratano 1989] [Sowa 2000]. The difference is essentially in the richness of knowledge expression in the form of CGs. After defining a production rule as `pattern => action`, Sowa describes how a forward chaining system processes such a rule:

> When it is executed, the inference engine searches working memory for some combination of data that matches the pattern. If the match succeeds, action on the right is executed to assert, retract, or modify facts or to call external programs that perform some computation. [Sowa 2000]

[Shinghal 1992] provides a similar description of a forward chaining production rule system. At the start of a cycle, assertions stored in a "working memory" may cause a number of production rules to become "heated" (Shinghal's word) by virtue of a match between asserted facts and rule antecedents. Only one of these is selected, on the basis of priority or specificity or some other criterion, and the consequent of the selected rule is acted upon, i.e. the rule fires. All heated rules are then "chilled". The change to working memory caused by the fired rule may cause another rule to fire on the next round.

> The practice of repeatedly heating prodrules and firing one of them is known in the literature as a recognise–act cycle/loop or as a select–execute cycle/loop. [Shinghal 1992]

The parallels with pCG are clear. pCG adds to this: CGs as patterns and the subjects of KB update, actors in preconditions, and graph export.

See the 'Experiments' chapter for non-trivial process examples.

[32] Although this also supports backward chaining, and pCG does not. See 'Future Work'.