Pipeline Creation Language (PCL)

User Manual

Version: 1.1.0-beta

Ian Johnson

November 19, 2013

Abstract

Pipeline Creation Language (PCL) is a general purpose language for creating non-recurrent software pipelines. This manual describes the syntax of PCL, how to compile it, and run it. Also, how to adapt your existing scripts or programs for use with PCL.

PCL was developed as part of the MosesCore project sponsored by the European Commission's Seventh Framework Programme (Grant Number 288487) `http://www.statmt.org/mosescore/`. For more information on the Seventh Framework Programme please see `http://cordis.europa.eu/fp7/home_en.html`.

Contents

List of Figures

Chapter 1

Introduction

Pipeline Creation Language (PCL) is a general purpose language for creating non-recurrent software pipelines. PCL is a small grammar for combining computation using re-usable components which can be shared or distributed. A number of combinator operators are defined which execute components sequentially or in parallel. Also, a number of pre-defined components can be used to "glue" components together, merge parallel outputs, or conditionally execute components.

PCL was developed as part of the MosesCore project sponsored by the European Commission's Seventh Framework Programme (Grant Number 288487) http://www.statmt.org/mosescore/. For more information on the Seventh Framework Programme please see http://cordis.europa.eu/fp7/home_en.html.

1.1 License and Availability

PCL compiler and runtime has been released under a LGPL v3.0[1] license. It is available from GitHub using the following command:

```
git clone https://github.com/ianj-als/pcl.git
```

1.2 Dependencies

Today the PCL compiler and runtime scripts require two dependencies that should be installed manually. The first dependency is PLY: a parser library. PCL re-

[1] See the GNU Lesser General Public License at http://www.gnu.org/copyleft/lesser.html

quires version 3.4 of PLY which can be found at `http://www.dabeaz.com/ply/ply-3.4.tar.gz`. Follow the instructions on how to build and install PLY v3.4. Addtionally, if you have Pyhton's `easy_install` you can issue the command:

```
sudo easy_install ply
```

The second dependency is Pypeline, the Python pipelining library that underpins PCL. Pypeline is a submodule of your PCL git clone. If you haven't already initialised the submodule, use the following command now to do so:

```
git submodule update --init
```

Pypeline can be installed using the instructions in the `README.md` in the directory `libs/pypeline`, or by setting your `PYTHONPATH` environment variable to:

```
<git clone root>/pcl/libs/pypeline/src
```

Running the Python REPL you should now be able to import the PLY and Pypeline packages using the following Python commands:

```
$ python
Python 2.7.3 (default, Apr 10 2013, 06:20:15)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> import ply
>>> import pypeline
>>>
```

Chapter 2

PCL Compiler

PLCc is the PCL compiler. It is located in the `src/pclc` directory of the Git clone and your path should be set according. This chapter introduces the PCL syntax using *railroad diagrams*. Railroad diagrams illustrate valid PCL and are read from left to right, following the lines as a train would. The symbols in the yellow ovals are to be typed *as is*. The symbols in the tan rectangles references another railroad diagram. The referenced diagram should be used to expand the rectangle in more valid PCL. Hexagons contain *character classes* and specify a range of characters that will be accepted.

2.1 PCL Syntax

PCL is a free-form language which allows the programmer to use arbitrary white-space to format your component definitions. Comments are a single line and should start with the # and can appear at any point in a PCL file. The top level syntax of a PCL file is shown in Figure 2.1, and consists of the following sections:

- **Imports**: Imports can be optionally specified. Importing, as in other language, makes available other components to the PCL component being written.
- **Component**: This starts the component definition and provides the name. The component's name must be the same as the filename. E.g., a component in `fred.pcl` must be called `fred`.
- **Inputs**: Defines the inputs of the component. This information is used to verify that the outputs of a previous component is compatible with another.

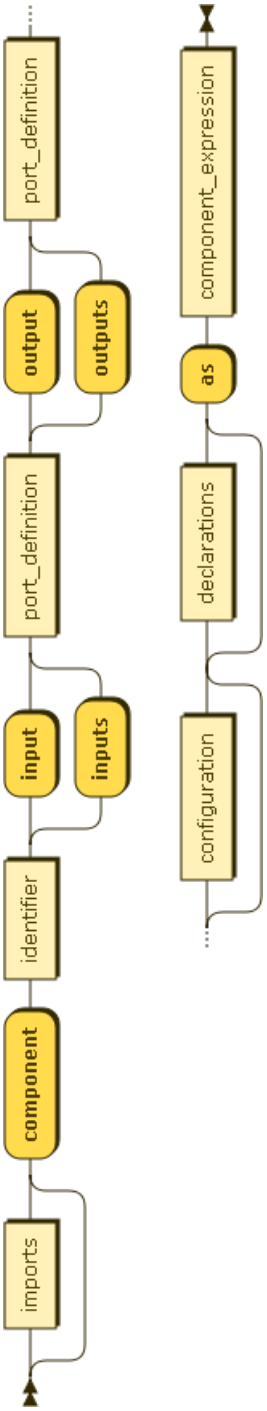


Figure 2.1: PCL file syntax.

- **Outputs**: Defines the outputs of the component. This information is used to verify that the inputs of a subsequent component is compatible with another.
- **Configuration**: Optional configuration for the component. This is static data that shall be used to construct imported components used in this component.
- **Declarations**: Optional declarations of components used in this component. This is where the import components are constructed.
- **Definition**: This portion is the component definition. It is an expression which defines how the constructed components are to be combined to create the computation required.

2.1.1 Imports

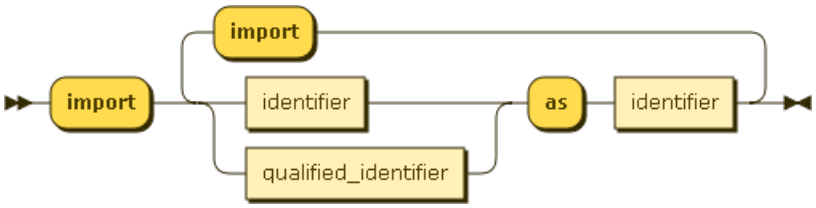


Figure 2.2: `imports` : Importing PCL files.

Extant PCL components can be used in other PCL components. The mechanism for using components is through importing. There can be zero or more imports in a PCL file. Figure 2.2 shows the syntax for importing.

The imported component is referenced using an identifier which can be fully qualified using a dot separated name. If the component is referenced using one or more dots the sequence of identifiers, apart from the last one, is used to address a package of components. The last identifier specifies the component name. The environment variable `PCL_IMPORT_PATH` is a colon separated list of directories from which a search shall take place for the PCL components. If this environment variable is not set then the current working directory is used as a starting point for the component search.

Each imported component must specify an alias. This is the name by which this component shall be referred to in this PCL file. E.g., `import components.utility.sleep as sleep_comp` shall import a PCL component called `sleep` from the package `components.utility` and shall be refereed to as, i.e. has the alias, `sleep_comp`.

2.1.2 Port Definition

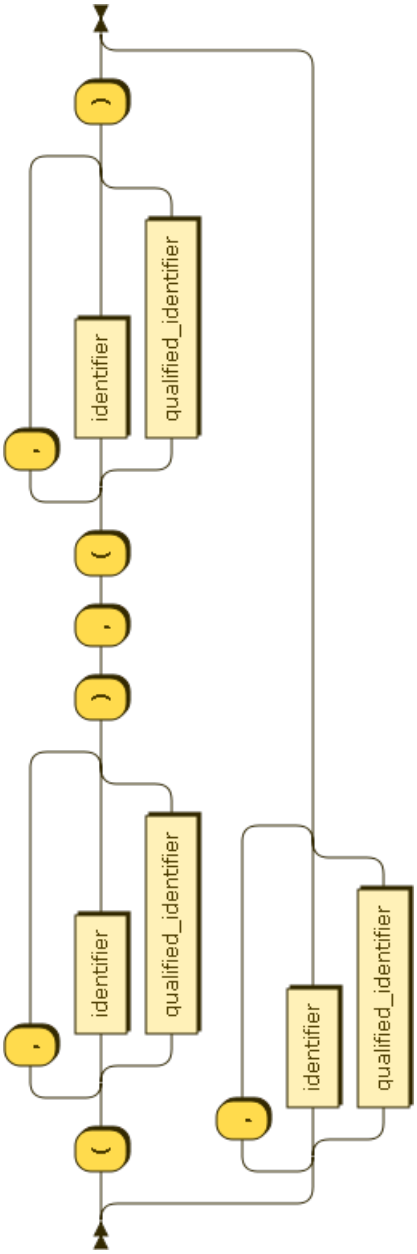


Figure 2.3: `port_definition` : Component port definition.

A port definition informs the PCL compiler about the nature of a component's input or an output. Components can have 2, 3 or 4 ports: one input and output port, one input port and two output ports, two input ports and one output port, or two

input and output ports. Figure 2.3 shows the syntax for this grammatical construct.

A port can carry one or more *signals*. A signal is a piece of data that flows through ports and has a unique name to that port, and can be fully qualified[1]. The signal names, for a port, are declared in a port definition. The "shape" of a port definition declares whether an input or an output has one or two ports. For example, consider the following input and output port definitions:

- A component with one input and one output port. Input signals are `tom`, `dick`, `harry` and output signals are `disley` and `standedge`.

```
component tunnel
    inputs tom, dick, harry
    outputs disley, standedge
    ...
```

- A component with one input and two output ports. Input signal is `fruit.banana`. The *top* output port signal is `veg.carrot` and the *bottom* output port signals are `fruit.super`, and `fruit.salad`.

```
component split
    inputs fruit.banana
    outputs ( veg.carrot ), ( fruit.super, fruit.salad )
    ...
```

- A component with two inputs and one output port. Input signals for the *top* port are `tea` and `coffee`, and the signals for the *bottom* port are `milk`, and `water`. The output port's single signal is `drink`.

```
component pot
    inputs ( tea, coffee ), ( milk, water )
    outputs drink
    ...
```

- A component with two input and output ports. Input signals for the *top* port are `zippy` and `george`, and the signals for the *bottom* port are `rod`, `jane`, and `freddy`. The output signals are for the *top* port are `geoffrey`, and the *bottom* port signals are `bungle` and `zippo`.

[1] It may be easier to group signals through the use of a hierarchical naming convention, e.g., `a.b.c` and `a.b.d`.

```
component rainbow
    inputs ( zippy, george ), ( rod, jane, freddy )
    outputs ( geoffrey ), ( bungle, zippo )
    ...
```

2.1.3 Configuration

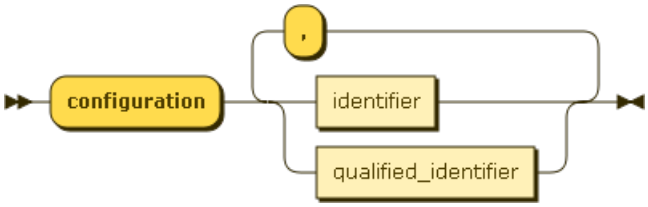


Figure 2.4: `configuration` : Component configuration.

A component's configuration is static data that is primarily used for constructing import components. Configuration data is named using identifiers, which can be fully qualified. Figure 2.4 shows the configuration syntax. Configuration identifiers may also be used in *if* components see Section 2.1.5 for details. A PCL component can declare zero or more configuration identifiers.

Figure 2.5 shows an example of configuration being declared in a PCL file. Here the `parallel_sleep` component shall be constructed using two configuration values, namely `sleep_command` and `sleep_time`.

```
import sleep as sleep

component parallel_sleep
    ...
    configuration sleep_command, sleep_time
    ...
```

Figure 2.5: Example declaration of PCL configuration.

2.1.4 Declarations

The declaration section of a PCL file is where the imported components can be constructed using the configuration available in the importing component's PCL. Figure 2.6 shows the syntax for component construction. All declarations are assigned to an identifier which shall be unique. The import alias (see Section 2.1.1)

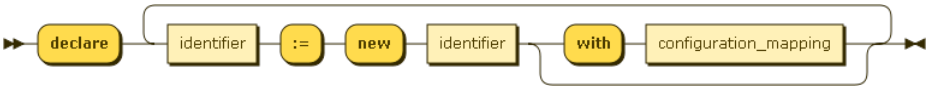


Figure 2.6: `declarations` : Imported component construction.

is used to reference the imported component from which an instance is created. There is an optional *with* clause which allows configuration to be mapped into a component's constructor. Figure 2.7 shows the configuration mapping syntax

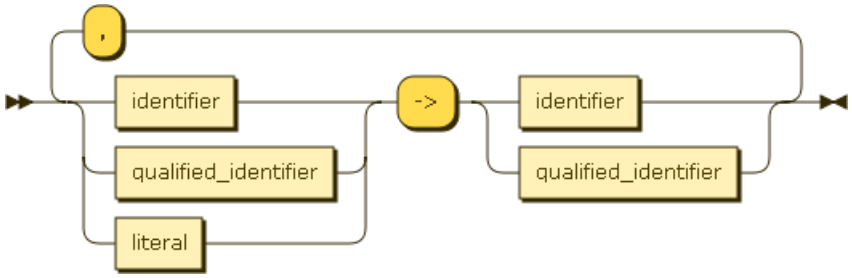


Figure 2.7: `configuration_mapping` : Declaration configuration mapping

The PCL snippet in Figure 2.8 shows an example of constructing imported components. Here two instances of the same component are constructed, namely

```
import sleep as sleep_component
import message as message_component

component parallel_sleep
    input sleep_time
    outputs (complete), (complete)
    configuration sleep_command
    declare
        top_sleep := new sleep_component
            with sleep_command -> command.sleep
        bottom_sleep := new sleep_component
            with sleep_command -> command.sleep
        message_world := new message_component
    ...
```

Figure 2.8: Example of component construction.

the `sleep` component, which has the alias `sleep_component`. The `sleep` component specifies the configuration `command.sleep` so the *with* clause maps the `parallel_sleep` component's configuration to the `sleep` component's configu-

ration. The `message` component, on the other hand, requires no configuration in order to construct it.

2.1.5 Definition

This section of the PCL file is where the component's computation is to be defined. The components constructed in the declarations are to be combined to produce a composite component. Figure 2.9 shows the recursive syntax, which builds a single expression, for this section.

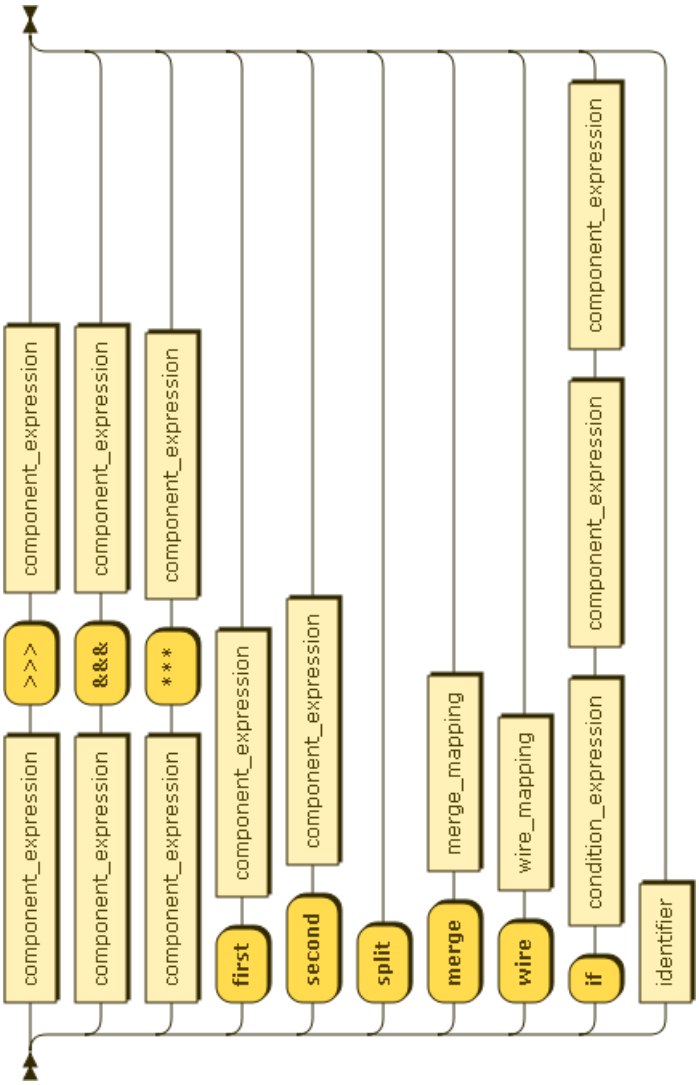


Figure 2.9: `component_expression` : Component definition.

Composition

The >>> combinator sequences two components, one after the other. For example `comp_one >>> comp_two` will produce a component which will apply the output of `comp_one` to the input of `comp_two`.

Consider two components both with one input and output port, the first component, c_1, has an input signal called `b` and output signal called `c`, and the second component, c_2, has an input signal called `c` and an output signal called `d`. Then composing these components, c_1 >>> c_2, yields a component whose input signal is called `b` and an output signal called `d`.

In order for the components to compose the output of the left-hand side must be compatible with the input of the right-hand side. That is, the number of ports must be identical and the signals must correspond. PCLc will produce an error during compilation if the components are not compatible.

First

The `first` combinator takes a single component and produces a two input and output port component. The *top*, or *first*, element of the input is applied to the component, whilst the *bottom*, or *second*, input element passes through the resultant component untouched. This second input signal's value is remembered across the computation of the provided component.

Consider a component, c, with one input and one output port whose signals are `b` and `c` respectively. Applying the `first` combinator to c, with $first\ c$, yields a component that has two input ports with the port specification $(b), (d)$, and two output ports with specification, $(c), (d)$.

Second

The `second` combinator takes a single component and produces a two input and output port component. The *bottom*, or *second*, element of the input is applied to the component, whilst the *top*, or *first*, input element passes through the resultant component untouched. This first input signal's value is remembered across the computation of the provided component.

Consider a component, c, with one input and one output port whose signals are `b` and `c` respectively. Applying the `second` combinator to c, with *second* c, yields a component that has two input ports with the port specification $(d), (b)$, and two output ports with specification, $(d), (c)$.

Parallel

The `***` combinator composes two components such that they run in *parallel*. This combinator is best explained in terms of the first, second and composition combinators. Thus,

$$a \ *** \ b = first\ a >>> second\ b$$

Hence, for two components:

- c_1: One input port, with port specification b, and one output port, with specification c, and
- c_2: One input port, with port specification d, and one output port, with specification e.

Then $c_1 \ *** \ c_2$ shall yield a two input and output port component with:

- Input port specification $(b), (d)$, and
- Output port specification $(c), (e)$.

Split

Split is a pre-defined component which has one input port and two output ports. Split simply takes the signals on it's input and copies them to both output ports. Hence, *splitting* the input to the output.

Fanout

The `&&&` combinator yields a component with one input port and two output ports. It is defined as:

$$a \ \&\&\& \ b = split >>> (first\ a \ *** \ second\ b)$$

Since `split` is begin used both components, a and b, require their input port signals to be compatible with the input to split.

Consider two components:

- c_1: One input port, with port specification b, and one output port, with specification c, and
- c_2: One input port, with port specification b, and one output port, with specification d.

Then c_1 &&& c_2 shall yield a one input and two output port component with:

- Input port specification b, and
- Output port specification $(c), (d)$.

PCLc verifies that components, used with the fanout combinator, have compatible ports and signals. The compiler will report errors if there are incompatible components used.

Merge

The merge component is a pre-defined component that expects a mapping and *merges* the output from a two output port component to a single port component. Hence, a merge component has two input ports and one output port.

The merge mapping syntax is shown in Figure 2.10. The merge mapping specifies which signals from the *top* and *bottom* input ports shall be mapped to the output port to uniquely named signals. Constant values can also be introduced to the output. Further to this, signals can be dropped, that is, a signal that is not passed onto the subsequent component.

Consider a component with two output ports, with the following port specification

$$(src.file.name,\ src.file.size),\ (obj.file.name,\ obj.file.size)$$

that needs merging into a single port, with specification:

$$source\_filename,\ object\_filename,\ source\_language$$

The following merge expression could be used:

```
merge top[src.file.name] -> source_filename,
      top[src.file.size] -> _,
      bottom[obj.file.name] -> object_filename,
      bottom[obj.file.size] -> _,
      "PCL" -> source_language
```

This merge mapping does the following:

- The signal `src.file.name`, in the *top* input port, is mapped to the output port signal `source_filename`,

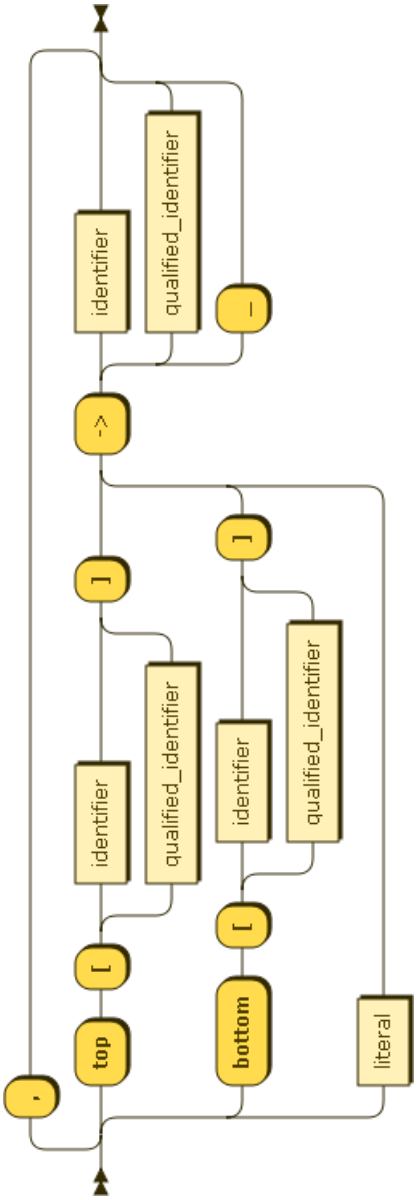


Figure 2.10: `merge_mapping` : Merge component mapping.

- The signal `src.file.size`, in the *top* input port, is dropped and is not present in the output port signals,
- The signal `obj.file.name`, in the *bottom* input port, is mapped to the output port signal `object_filename`,
- The signal `obj.file.size`, in the *bottom* input port, is dropped and is not

present in the output port signals, and

- The string literal `PCL` is mapped into the output port's `source_language` signal.

Other constant literals can be used in a merge mapping and are described in Section 2.1.8.

Wire

Wire components are used to adapt one component's output signals to match the expected input signals of a subsequent component. Wires can only be used to adapt adjacent components that have an equal number of ports, i.e., the resultant wire component always has the same number of input ports as output ports. The *wire mapping* determines whether a one or two port component is being adapted, this is shown in Figure 2.11. The mapping syntax is shown in Figure 2.12. In common

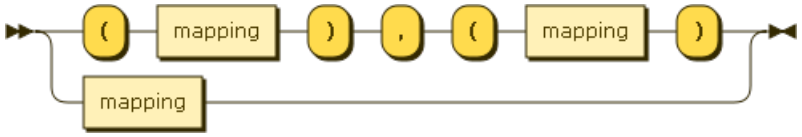


Figure 2.11: `wire_mapping` : Wire mapping.

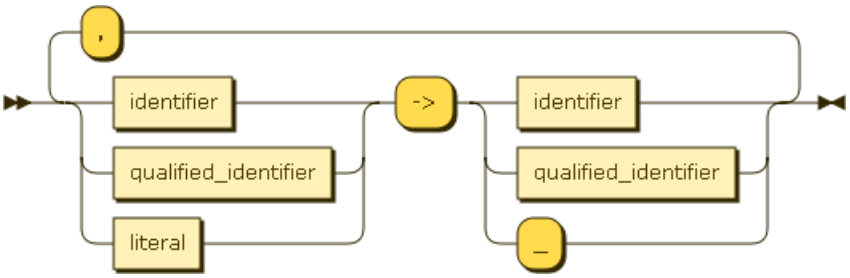


Figure 2.12: `mapping` : Mapping.

with the merge mapping, signals can be mapped into other signals, dropped, or assigned literal values.

Consider the following wire component:

```
wire src.file.name -> source_filename,
     src.file.size -> _,
     "PCL" -> source_language
```

This mapping adapts two components with one output and one input port with the following port specifications,

$$src.file.name,\ src.file.size$$

and

$$source\_filename,\ source\_language$$

respectively.

A two input and output port wire is defined thus,

```
wire (src.file.name -> source_filename,
      src.file.size -> _,
      "PCL" -> source_language),
     (obj.file.name -> object_filename,
      obj.file.size -> _,
      "Python" -> object_language)
```

This mapping adapts two components with two output and two input ports with the following port specifications,

$$(src.file.name,\ src.file.size),\ (obj.file.name,\ obj.file.size)$$

and

$$(source\_filename,\ source\_language),\ (object\_filename,\ object\_language)$$

respectively.

If

The *if* component provides a mechanism to conditionally execute components in a pipeline. The first argument to the *if* component is a condition expression, the second is the *then* component, and the third is the *else* component. The *then* and *else* components *must*:

- only specify one input port,
- specify identical signals on the input ports,
- have identical numbers of output ports (one or two), and

- specify identical signals in their output ports.

If the condition expression is evaluated to a truthy value the *then* component shall be executed, otherwise the *else* component is executed. In Figure 2.13 the

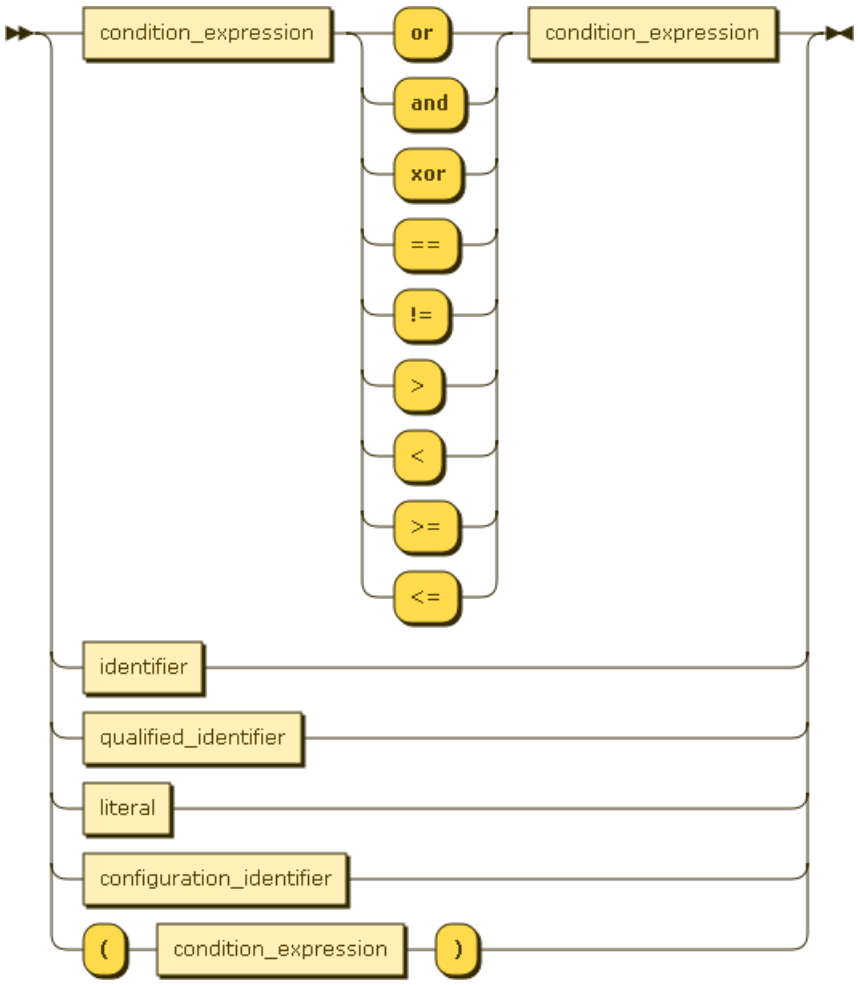


Figure 2.13: `condition_expression` : If component's condition expression.

recursive syntax for the condition expression is shown. A condition expression is built up using the logical operators (`or`, `and`, and `xor`)[2], and the relational operators (==, !=, >, <, >=, and <=)[3]. The `identifier` and `qualified_identifier` refer to the signals in the input ports of the *then* and *else* components. Also, configuration identifiers may be used in a condition expression, the grammar of which is shown in Figure 2.14. This allows both the values of input port signals and

[2]Hopefully these operators require no explanation.
[3]And these! ;)

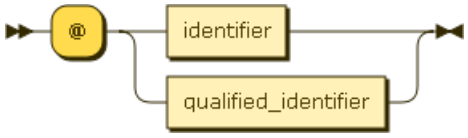


Figure 2.14: `configuration_identifier` : If component's condition expression configuration identifier.

configuration to be used to decide on which component to execute.

Consider the following PCL snippet:

```
...
component conditional
  inputs a, b
  outputs z
  configuration f
  ...
  as
    if (a == True or a != b) and @f == False
       then_component
       else_component
    ...
```

The `then_component` shall only be executed if the input `a` is true or `a` and `b` are equal, and the configuration `f` is false.

2.1.6 Identifier

In common with other programming languages PCL offers identifiers which can start with a letter or an underscore and then any number of letters, numbers or underscores, or diagrammatically see Figure 2.15.

2.1.7 Qualified Identifier

Qualified identifiers allows PCL developers to namespace their identifiers using dot separated identifiers, e.g., `tokeniser.source.filename`. Their syntax is shown in Figure 2.16. Qualified identifiers are available for use in:

- Imports: qualified identifiers are used to address compiled PCL components (see Section 2.1.1),

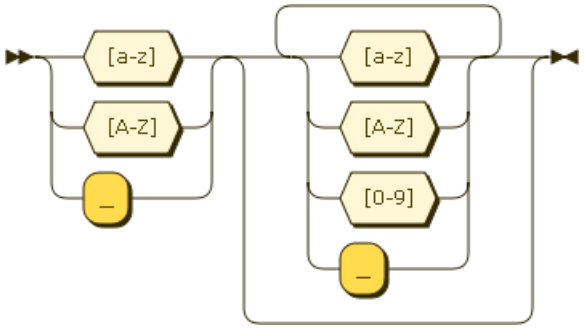


Figure 2.15: `identifier` : Identifier.

- Port definitions: input and output signals can be namespaced for clarity (see Section 2.1.2),
- Configuration: configuration identifiers can be namespaced for clarity (see Section 2.1.3),
- Declarations: configuration mapping may required namespaced configuration to be used (see Section 2.1.4),
- Merge and wire mappings: mapped port can contain namespaced signal names (see Section 2.1.5 and Section 2.1.5 respectively), and
- *If* conditions: configuration used in *if* condition expressions may be namespaced (see Section 2.1.5).

2.1.8 Literal

Literal constants can be used to inject values into component constructors, merge and wire mappings, and *if* condition expressions. They take the form of:

- Numbers: integer and floating point, e.g., `-7`, `2.71828`, `6.674e-11`,
- Strings: string must be quoted using double quotes (") and special characters can be escaped using a backslash (\), e.g. `"This is a line of text\n"`, and `"Quoting is \"allowed\" by escaping the quotes"`, and
- Booleans: `true` and `false`.

Figure 2.17 shows their syntax.

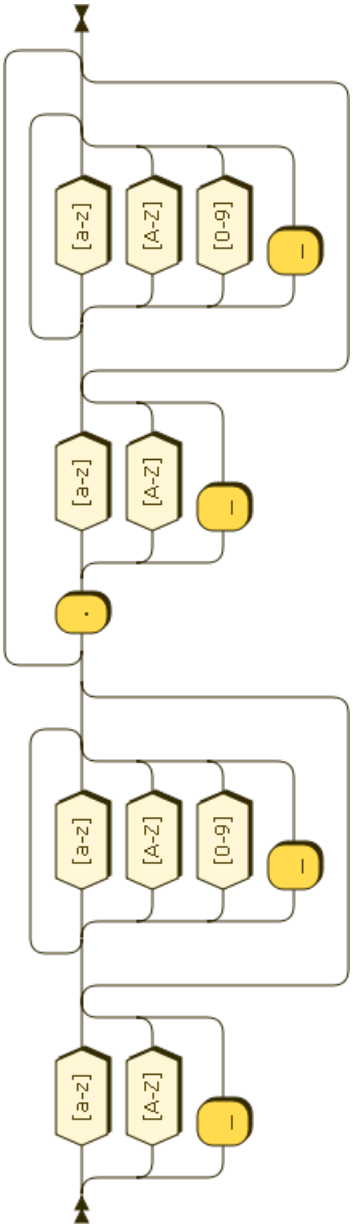


Figure 2.16: `qualified_identifier` : Qualified identifier.

2.1.9 Example PCL file

This example PCL file can be found in the `parallel_sleep` example PCL. This component constructs two sleep components using a static sleep command to execute. The two sleep components are composed using the fanout operation, such that they run in parallel. Other example PCL files can be found in the `examples`

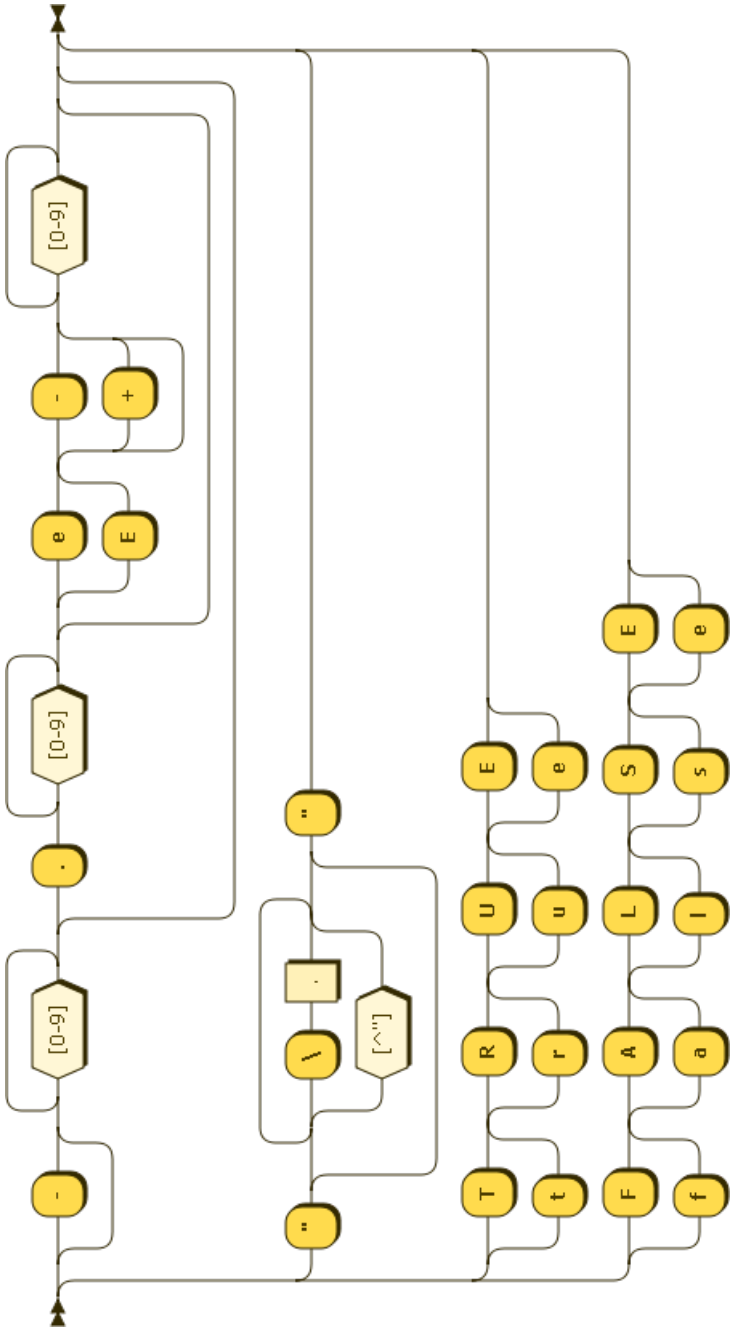


Figure 2.17: `literal` : Literal.

directory of your Git clone.

```
import sleep as sleep

component parallel_sleep
  input sleep_time
  outputs (complete), (complete)
  configuration sleep_command
  declare
    top_sleep := new sleep with sleep_command -> sleep_command
    bottom_sleep := new sleep with sleep_command -> sleep_command
  as
    top_sleep &&& bottom_sleep
```

Figure 2.18: Example PCL file.

2.2 Usage

Ensure you have `src/pclc/pclc.py` in your platform path. Running `pclc.py -h` yields:

```
Usage: pclc.py [options] [PCL file]


Options:
  -h, --help            show this help message and exit
  -l LOGLEVEL, --loglevel=LOGLEVEL
                        parser log file reporting level
                        [default: WARN]
  -i, --instrument      Generated code shall instrument
                        components
  -v, --version         show version and exit
```

The command-line options are:

- `-h`, `--help`: Display the help message,
- `-l`, `--loglevel`: The logging level for the `pclc.log` file that is created during compilation. This file, depending on log level, shall show information about the parsing internals of PCLc,
- `-i`, `--instrument`: Specifying this flag shall add code to the generated component which shall log to standard error when the component's con-

structed and used components start and finish. The log messages are time stamped so can be used as a rudimentary profiling tool, and

- `-v`, `--version`: Show the version of PCLc.

For example, change directory to `src/examples/parallel_sleep` and issue the command:

```
pclc.py -i parallel_sleep.pcl
```

The `pcl` extension is not required so running the compiler with:

```
pclc.py -i parallel_sleep
```

has the same effect. The compilation process will generate three new files:

- `parallel_sleep.py`: The object code from the compilation. PCL compiles to Python and this file shall be used by the runtime to build the final pipeline,
- `__init__.py`: Since the compiled PCL is a Python module, in order to import it at runtime PCLc must write this file to show that this is a Python package, and
- `pclc.log`: The log file for the compilation. This is mainly used for support but may be of interest to others.

In the next chapter you will see how to run pipelines using the Python based runtime.

Chapter 3

PCL Runtime

The PCL runtime is an optional method of running a pipeline. It can be found in the `src/pcl-run` directory of the Git clone. Ensure this directory is in your platform path and issue:

```
pcl-run.py -h
```

This yields:

```
Usage: pcl-run.py [options] [PCL configuration]

Options:
  -h, --help            show this help message and exit
  -v, --version         show version and exit
  -n NO_WORKERS, --noworkers=NO_WORKERS
                        number of pipeline evaluation
                        workers [default: 5]
```

The command-line options are:

- `-h`, `--help`: Display the help message,
- `-v`, `--version`: Show the version of PCLc.
- `-n`, `--noworkers`: The components are executed in a thread pool. This option determines the maximum size of this thread pool. If you find that components that are expected to execute in parallel are running sequentially, then increasing the number of threads in the pool may help.

3.1 Pipeline Configuration

The pipeline configuration file contains the static configuration used by components to construct other components, and the pipeline's inputs. The filename must be the same as the component you wish to run with a `.cfg` extension, e.g., the `parallel_sleep` configuration file is called `parallel_sleep.cfg`. The configuration file contains two sections `[Configuration]`, for configuration values, and `[Inputs]`, for pipeline inputs. Each section contains key value pairs, e.g., the `parallel_sleep` configuration file looks like this:

```
[Configuration]
sleep_command = /bin/sleep
[Inputs]
sleep_time = 5
```

Environment variables can be used in configuration files with `$(VAR_NAME)`. The environment variable, if it exists, shall be substituted and used in the pipeline.

3.2 Running a Pipeline

At the end of the last chapter you compiled the `parallel_sleep` component from the `examples` directory. To run this pipeline return to the `examples/parallel_sleep` direcotry and run:

```
pcl-run.py parallel_sleep.cfg
```

or

```
pcl-run.py parallel_sleep
```

After 5 seconds the runtime should display on `stdout`:

```
({'complete': True}, {'complete': True})
```

If, on the other hand, you have compiled this pipeline with instrumentation enabled (see Section 2.2) you should see something like this:

```
07/02/13 15:45:40.851373: MainThread: Component parallel_sleep
is constructing bottom_sleep (id = 38338448) with
configuration {'sleep_command': '/bin/sleep'}
(sleep instance declared at line 27)
```

```
07/02/13 15:45:40.851504: MainThread: Component parallel_sleep
is constructing top_sleep (id = 38392400) with
configuration {'sleep_command': '/bin/sleep'}
(sleep instance declared at line 26)
07/02/13 15:45:40.852697: Thread-2: Component parallel_sleep
is starting top_sleep (id = 38392400) with input
{'sleep_time': 5} and state {'sleep_command': '/bin/sleep'}
07/02/13 15:45:40.856738: Thread-2: Component parallel_sleep
is starting bottom_sleep (id = 38338448) with input
{'sleep_time': 5} and state {'sleep_command': '/bin/sleep'}
07/02/13 15:45:45.857495: Thread-1: Component parallel_sleep
is finishing top_sleep (id = 38392400) with input
{'complete': True} and state {'sleep_command': '/bin/sleep'}
07/02/13 15:45:45.859939: Thread-5: Component parallel_sleep
is finishing bottom_sleep (id = 38338448) with input
{'complete': True} and state {'sleep_command': '/bin/sleep'}
({'complete': True}, {'complete': True})
```

The timestamped lines appear on `stderr`.

3.3 Gotchas

PCL allows for components to be defined in hierarchical namespace. All directories, in your PCL component heirarchical namespace, that do not contain PCL files must contain `__init__.py` in order for the Python runtime to "see" these directories as Python packages. Failure to do so will yield an error in the form:

```
ERROR: Failed to import PCL module parallel_sleep: No module
named parallel_sleep
```

3.4 Using PCL in your own Python programs

If you wish to running PCL pipelines in your own programs a function exists in `src/pcl-run/runner/runner.py` called `execute_module(executor, pcl_import_path, pcl_module, get_configuration_fn, get_inputs_fn)`. This function returns a 2-tuple whose first element is the expected outputs of the pipeline, and the second element is the output of the executed pipeline.

For example, the `parallel_sleep` pipeline would output:

```
(([’complete’], [’complete’]),
 ({’complete’: True}, {’complete’: True}))
```

The inputs are:

- `executor`: A `concurrent.futures.ThreadPoolExecutor` object,
- `pcl_import_path`: A colon separated string of directories from which to search for PCL components,
  e.g., `com.mammon.wizz.components.pre_processing:`
  `com.mammon.wizz.components.workers`.
- `pcl_module`: A dot separated string representing the path to a compiled PCL module, e.g., `trail.pipelines.gonzo`,
- `get_configuration_fn`: A function which shall receive an iterable which contains the expected configuration for the component. This fucntion shall return a dictionary whose keys are the expected configuration along with their values, e.g.,

  ```
  def get_configuration(expected_configuration):
    configuration = dict()
    for config_key in expected_configurations:
      configuration[config_key] =
        # You need to implement this function
        get_configuration_from_provider(config_key)

     return configuration
  ```

- `get_inputs_fn`: A function that shall receive the input port specification for the component. A tuple indicates a two port input and shall contain two iterable collections containing the signals for both input ports, otherwise it is an iterable collection of signal names for the single output port. The function shall return a 2-tuple of dictionaries whose keys are the expected input signal names and values when the component has two input ports. Or, a dictionary whose keys represent the signals of a single input port, e.g.,

```
def get_inputs(expected_inputs):
  def build_inputs_fn(inputs):
    input_dict = dict()
    for an_input in inputs:
      input_dict[an_input] =
        # You need to implement this function
        get_input_from_provider(an_input)
    return input_dict

  if isinstance(expected_inputs, tuple):
    inputs = list()
    for set_inputs in expected_inputs:
      inputs.append(build_inputs_fn(set_inputs))
    inputs = tuple(pipeline_inputs)
  else:
    inputs = build_inputs_fn(expected_inputs)

  return inputs
```

Chapter 4

Adapting to PCL

Adapting pre-existing executables to PCL can be achieved in two ways, they are:

- **Imperative PCL**: The PCL language supports an imperative style grammar which allows component authors to use runtime libraries to run external executables. This portion of the PCL grammar is not *Turing complete* since there are no looping constructs. This feature of PCL exists to quickly initialise pre-requisites of an external executable, run the executable, post-process a result, and return the result. If it turns out your component needs a more complex pre- and post-processing, or a component cannot be, or need not be an external executable, the second approach can be used.
- **Python Wrapper**: A Python file, containing, six functions can be written that informs PCLc about the nature of the component. Properties defined are: component's name, input and output port specifications, configuration and pre-processing configuration, and the component's computation. Since the computation of the component is described using Python, an arbitrarily complex component can be written.

Care must be taken when adapting your existing work to PCL pipelines. Threading issues and batch or on-line processing must be considered as the dynamics of your final pipeline may depend on it. Also, any state that may need to accumulate over the lifetime of a PCL component must be handled by the adaptor for your programs.

4.1 Imperative PCL

The imperative PCL language is a free-form language which allows the programmer to use arbitrary white-space to format your component definitions. Comments are a single line and should start with the # and can appear at any point in a PCL file. The top level syntax of a PCL file is shown in Figure 4.1, and consists of the following sections:

- **Imports**: Imports can be optionally specified. The imports here are PCL runtime libraries. These are written in Python and are modules which contain functions. Imports must specify an alias and this is use to call the functions, e.g., `list.insert(...)`.
- **Component**: This starts the component definition and provides the name. The component's name must be the same as the filename. E.g., a component in `fred.pcl` must be called `fred`.
- **Inputs**: Defines the inputs of the component. This information is used to verify that the outputs of a previous component is compatible with another. Only *one* input port can be defined.
- **Outputs**: Defines the outputs of the component. This information is used to verify that the inputs of a subsequent component is compatible with another. Only *one* output port can be defined.
- **Configuration**: Optional configuration for the component. This is static data that shall be used to construct imported components used in this component.
- **Commands**: This portion is the component definition. It is a list of commands which are executed in order from top to bottom.

An example imperative PCL file can be seen in Figure 4.2. This example can be found in the `parallel_sleep` example in the PCL Git repository.

4.1.1 Imports

Imperative PCL files can use runtime library functions to import functionality. These runtime libraries are Python files which contain functions. The names of functions are the name of the function in PCL. Figure 4.3 shows the syntax for importing. The environment variable `PCL_IMPORT_PATH` is a colon separated list

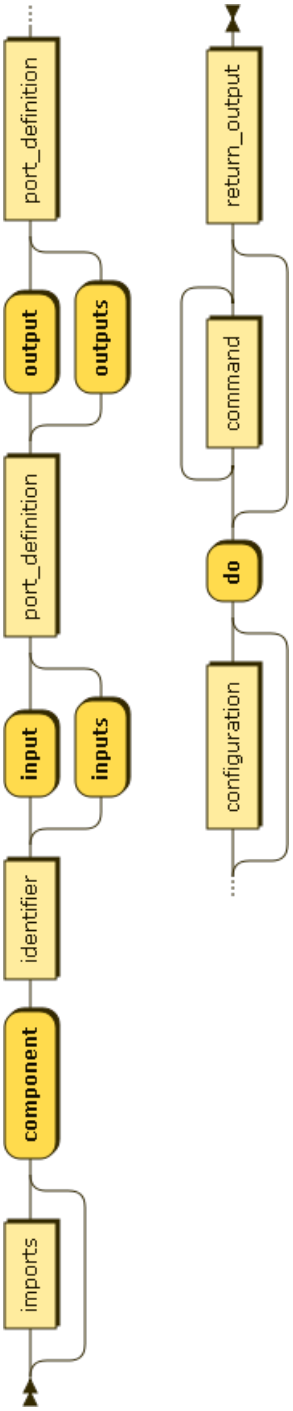


Figure 4.1: Imperative PCL file syntax.

```
import pcl.system.process as process
import pcl.util.list as list

component sleep
  input sleep_time
  output complete
  configuration sleep_command
  do
    cmd <- list.cons(@sleep_command, sleep_time)
    process.callAndCheck(cmd)

    return complete <- True
```

Figure 4.2: Example imperative PCL file.

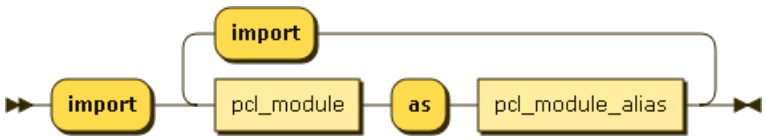


Figure 4.3: `imports` : Importing PCL runtime functions.

of directories from which a search shall take place for the PCL runtime libraries. If this environment variable is not set then the current working directory is used as a starting point for the component search.

Each imported component must specify an alias. This is the name by which this component shall be referred to in this PCL file. E.g., `import pcl.util.list as list` shall import a PCL runtime library called `list` from the package `pcl.util` and shall be refereed to as, i.e. has the alias, `list`.

Figure 4.4 shows how the non-terminals expand in the import syntax.

(a) `pcl_module` expansion

(b) `pcl_module_alias` expansion

Figure 4.4: `pcl_module` & `pcl_module_alias` : Imperative import PCL non-terminals.

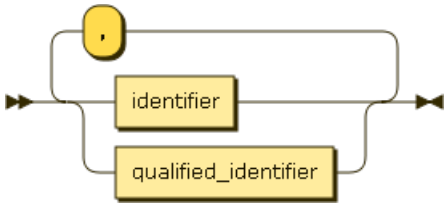


Figure 4.5: `port_definition` : Imperative PCL port definition.

4.1.2 Port Definition

A port definition informs the PCL compiler about the nature of a component's input or an output. Components defined with imperative PCL can only have one input and one output port. Figure 2.3 shows the syntax for this grammatical construct.

Again, ports carry one or more *signals*. A signal is a piece of data that flows through ports and has a unique name to that port, and can be fully qualified. The signal names, for a port, are declared in a port definition. Signal names are read-only.

4.1.3 Configuration

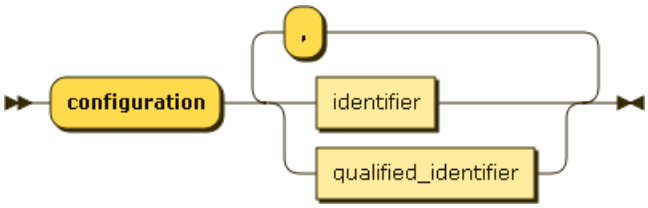


Figure 4.6: `configuration` : Imperative PCL configuration.

A component's configuration is static and read-only data. Configuration data is named using identifiers, which can be fully qualified. Figure 4.6 shows the configuration syntax. Configuration identifiers may be used at any point where a variable, or input signal name can be used, e.g., and *if* command, or function call. In imperative PCL zero or more configuration identifiers can be declared.

4.1.4 Commands

The command syntax is shown in Figure 4.7. Each command yields a value which can, optionally, be assign to a write-once "variable".

Function Calls

The PCL runtime functions can be used in an imperative PCL component using the following syntax shown in Figure 4.8. Functions are called using the import alias, assigned in the import statement, and the function name, e.g., `list.insert(...)`. A function call's arguments can be any one of an input signal, variable or configuration values, e.g., the call `list.cons(filename, @working.directory, extension)` constructs a list using the input signal `filename`, the configuration `working.directory`, and a variable `extension`. This list could be assigned to a variable using the following `pathname <- list.cons(filename, @working.directory, extension)`. The variable `pathname` is readable in all scopes below the current scope.

Runtime functions can be written by users and should be in Python modules, i.e., a directory containing the file `__init__.py`. These files should contain only functions, e.g., see Figure 4.9 for an example of the runtime library `pcl.util.string`. In order for PCLc to "see" these library modules the `PCL_IMPORT_PATH` environment variable must specify the directory in which these Python modules can be found.

Figure 4.10 shows how the non-terminals expand in a function call.

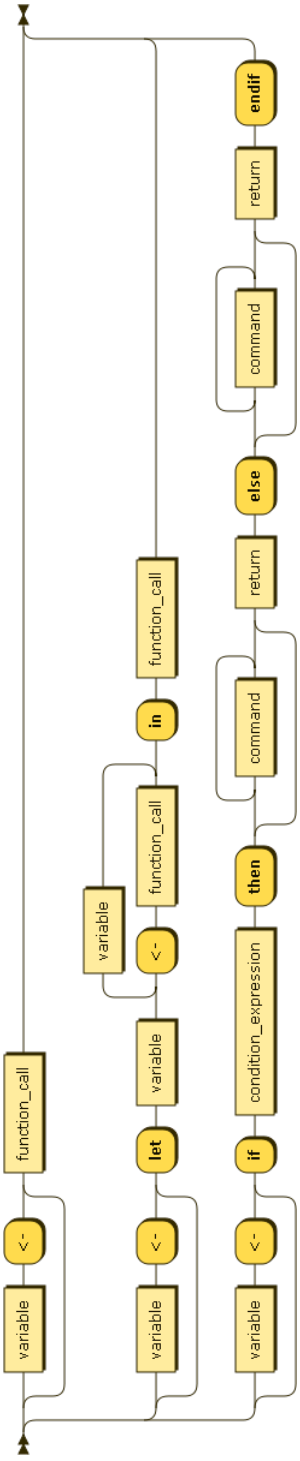


Figure 4.7: `command` : Imperative PCL commmands.

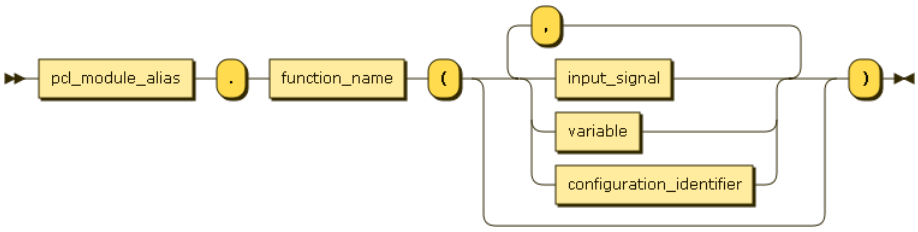


Figure 4.8: `function_call` : Imperative PCL function call.

```
split = lambda s, ss: s.__str__().split(ss)

join = lambda l, s: s.join([str(e) for e in l])

lower = lambda s: s.__str__().lower()

upper = lambda s: s.__str__().upper()
```

Figure 4.9: `string.py`: An example of a runtime library.

(a) `function_name` expansion

(b) `input_signal` expansion

(c) `variable` expansion

(d) `configuration_identifier` expansion

Figure 4.10: `function_name` & `input_signal` & `variable` & `configuration_identifier`: Imperative PCL function call non-terminals.

Let Bindings

Let bindings allow variables to be scoped so that only the function call in the *in* clause has access to them. For example, Figure 4.11 shows a let binding which builds a new pathname for a supplied filename and working directory. Notice all input signals and configuration are accessible inside the let binding. Moreover, any assigned variables before the let binding would be accessible. All variables assigned inside the let binding are only accessible inside of the let binding. Finally, the value of the `path.join()` function is assigned to the `pathname` variable.

```
import pcl.os.path as path
import pcl.util.list as list
import pcl.util.string as string

component pathname_creator
  input filename
  output filename.new
  configuration working.directory
  do
    pathname <- let
                  basename <- path.basename(filename)
                  pieces <- path.splitext(basename)
                  base <- list.index(pieces, 0)
                  ext <- list.index(pieces, 1)
                  bits <- list.cons(base, "new", ext)
                  new_basename <- string.join(bits, ".")
                in
                  path.join(@working.directory, new_basename)

    return filename.new <- pathname
```

Figure 4.11: Let binding example

If Commands

If commands in imperative PCL are similar to ternary operators available in many computing languages. However, *If* commands in PCL are more powerful and allow other commands to be executed from within the `then` and `else` blocks. Since all commands in imperative PCL have a value both the `then` and `else` blocks must be specified, and use the `return` keyword to create a value for when the condition is true or false. If no value is to be generated then the special `return ()` statement can be used. The syntax for the return constructed is shown in Figure 4.12.

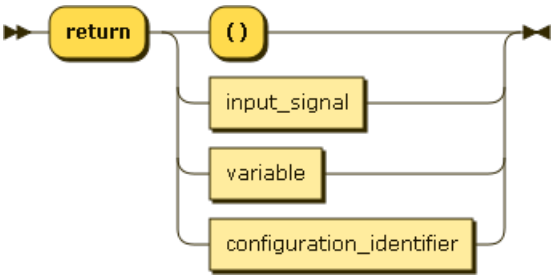


Figure 4.12: `return` : Imperative PCL *If* evaluations.

The `return` command does *not* have the same semantics as return in other imperative languages. In imperative PCL, `return` is used to create a value that shall become the "value" of the *If* command.

Otherwise, the value of a variable can be made available for assignment, e.g., Figure 4.13 shows an example of how to evaluate a value and *null* value. The syntax for the the *If*'s condition is shown in Figure 4.14.

```
import pcl.os.path as path
import pcl.util.list as list

component pathname_creator
  input filename
  output filename.new
  configuration working.directory
  do
    working.directory.exists <- path.exists(@working.directory)

    # If the working directory exists create a pathname
    # using the working directory.
    pathname <- if working.directory.exists == True then
      pathname <- path.join(@working.directory, filename)
      return pathname
    else
      return ()
    endif

    return filename.new <- pathname
```

Figure 4.13: *If* example

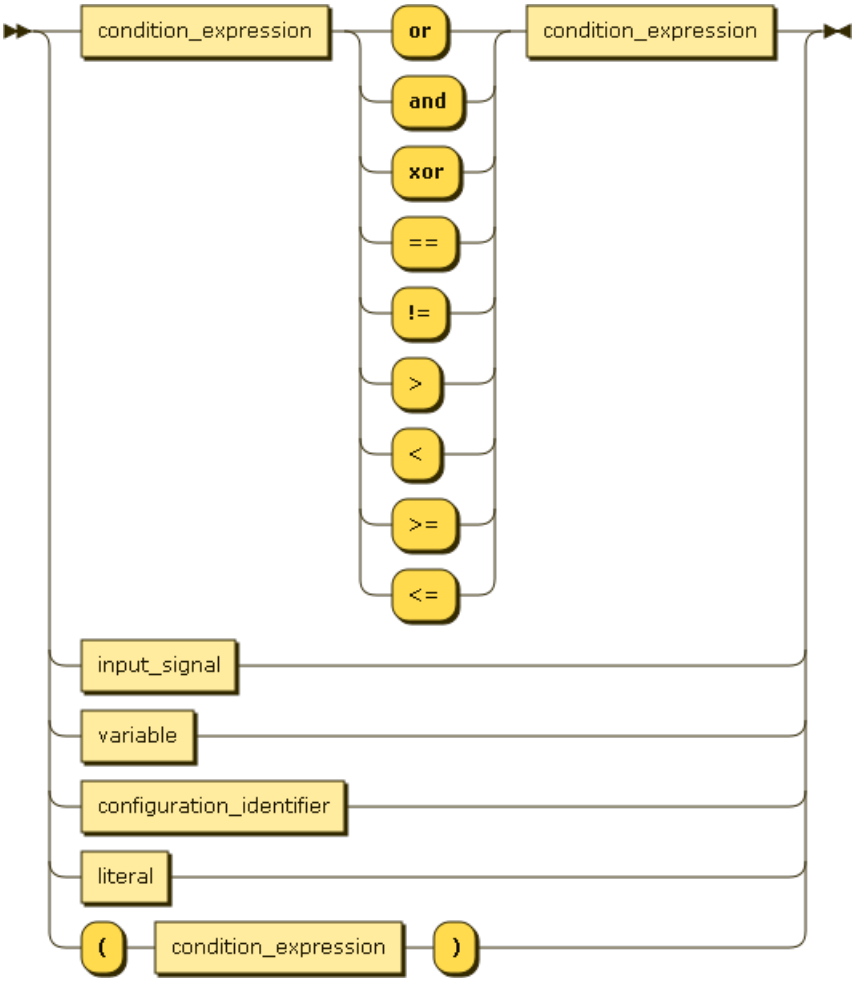


Figure 4.14: `condition_expression` : Imperative PCL *If* evaluations.

4.1.5 Return Output

A PCL component must produce signals on its output port. To do this a mapping is created that uses input signals, variables, or literals and assigns these values to output signals, as shown in Figure 4.15.

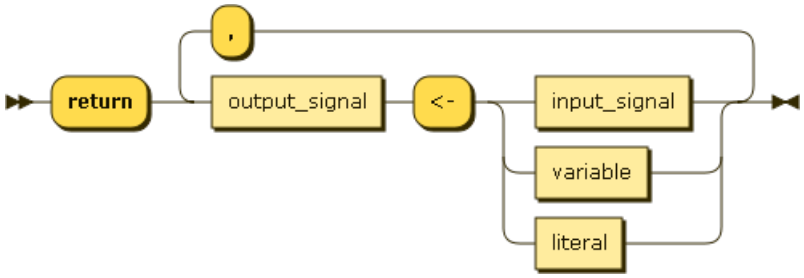


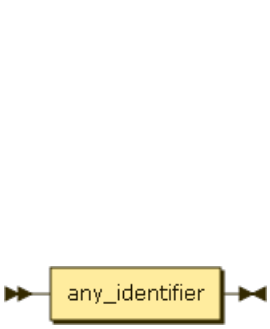
Figure 4.15: `return_output` : Imperative PCL returning component's output.

The output signal is defined as shown in Figure 4.16.

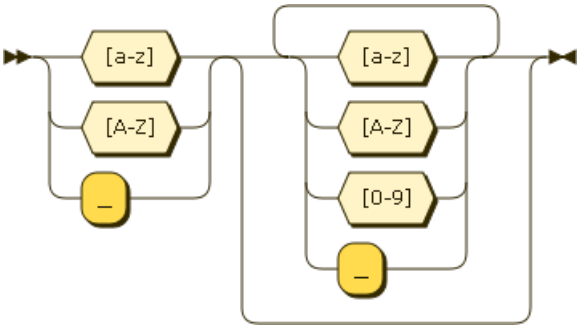
Figure 4.16: `output_signal` : Imperative PCL creating output signals.

4.1.6 Miscellaneous Constructs

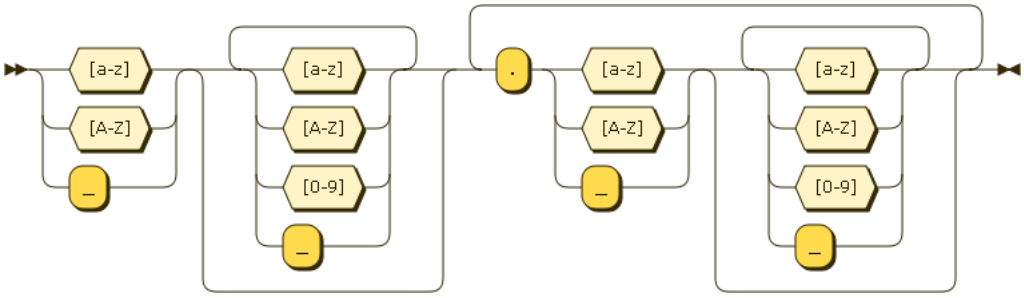
Figures 4.17 and 4.18 show the remaining undefined imperative PCL constructs that were used in the previous sections.



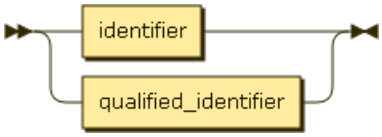
(a) `signal_name` expansion



(b) `identifier` expansion



(c) `qualified_identifier` expansion



(d) `any_identifier` expansion

Figure 4.17: `signal_name` & `identifier` & `qualified_identifier` & `any_identifier`: Imperative PCL miscellaneous non-terminals.

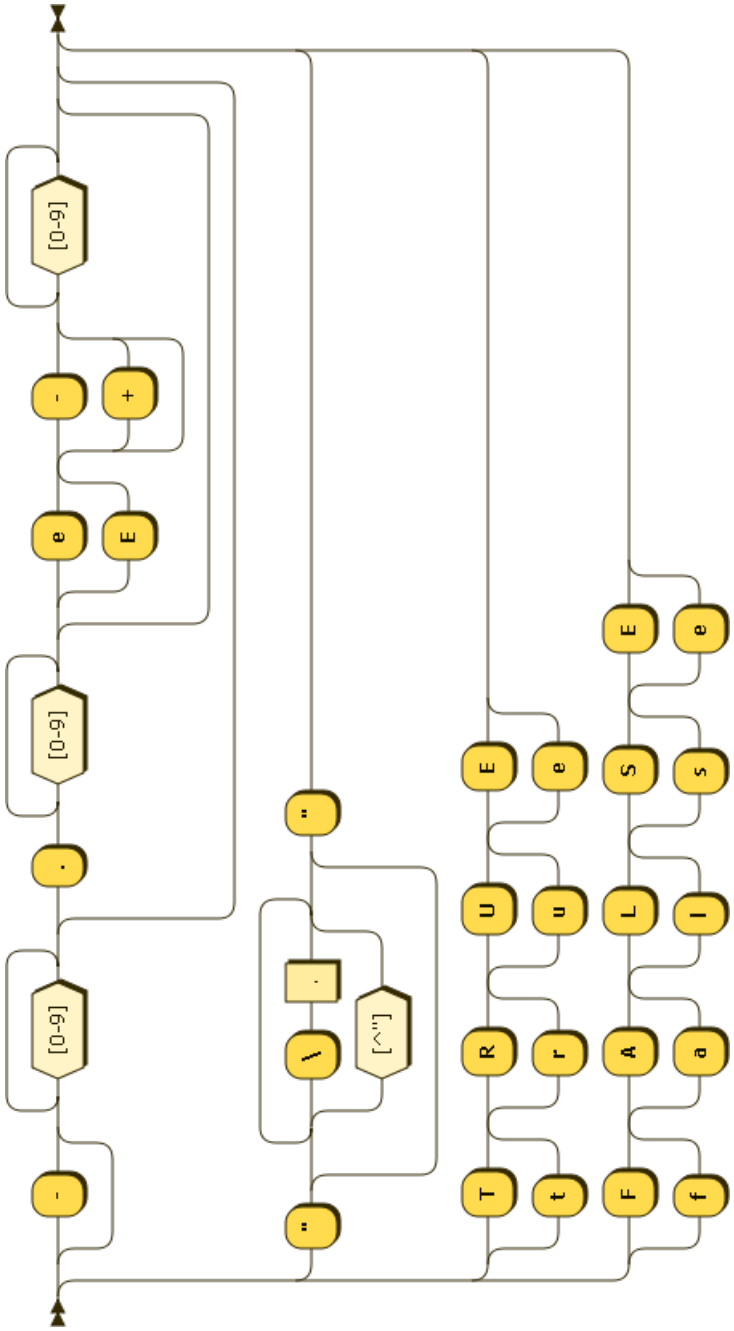


Figure 4.18: `literal` expansion.

4.2 Python Wrapper

The Python wrappers for your programs can inhabit the same hierarchical package structure as your PCL hierarchy. This is because the PCL hierarchy mirrors the Python one[1].

Six functions are required in your Python wrapper, they are:

- `get_name()`: Returns an object representing the name of the component. The `__str__()` function should be implemented to return a meaningful name. E.g.,

  ```
  def get_name():
    return 'tokenisation'
  ```

- `get_inputs()`: Returns the inputs of the component. Components should only be defined with one input port, which is defined by returning a single list of input port signal names. E.g.,

  ```
  def get_inputs():
    return ['port.in.a', 'port.in.b']
  ```

- `get_outputs()` - Returns the outputs of the component. Components should only be defined with one output port, which is defined by returning a single list of output port signal names. E.g.,

  ```
  def get_outputs():
    return ['port.out.a', 'port.out.b', 'port.out.c']
  ```

- `get_configuration()`: Returns a list of names that represent the static data that shall be used to construct the component. E.g.,

  ```
  def get_configuration():
    return ['buffer.file', 'buffer.size']
  ```

- `configure(args)`: This function is the component designer's chance to preprocess configuration injected at runtime. The `args` parameter is a dictionary that contains all the configuration provided to the entire pipeline.

[1] This is the reason why `__init__.py` files must be manually placed in directories in your PCL hierarchy which have no PCL files.

This function is to filter out, and optionally preprocess, the configuration used by this component. This function shall return an object containing configuration necessary to construct this component. E.g. this example returns a dictionary of configuration,

```
import os
def configure(args):
  buffer_file = os.path.abspath(args['buffer.file'])
  return {'buffer.dir' : os.path.dirname(buffer_file),
          'buffer.file' : os.path.basename(buffer_file),
          'buffer.size' : args['buffer.size']}
```

- `initialise(config)`: This function is where the component designer defines the component's computation. The function receives the object returned from the `configure()` function and must return a function that takes two parameters, an input object, and a state object. The input object, `a` in the example below, is a dictionary that is received from the previous component in the pipeline. The keys of this dictionary are the signal names from the previous component's output port. The state object, `s` in the example below, is a dictionary containing the configuration for the component. The keys of the configuration dictionary are defined by the `get_configuration()` function. The returned function should be used to define the component's computation. E.g.,

```
import subprocess
def initialise(config):
  def sleep_function(a, s):
    proc = subprocess.Popen([config['sleep_command'],
                             str(a['sleep_time'])])
    proc.communicate()
    return {'complete' : True}

  return sleep_function
```

The function returned by `initialise()` is executed in the thread pool used by the runtime (see Chapter 3). It is implementation defined as to whether this function blocks, waiting for a computation to complete, or not.

```
import subprocess

def get_name():
  return "sleep"

def get_inputs():
  return ['sleep_time']

def get_outputs():
  return ['complete']

def get_configuration():
  return ['sleep_command']

def configure(args):
  return {'sleep_command' : args['sleep_command']}

def initialise(config):
  def sleep_function(a, s):
    proc = subprocess.Popen([config['sleep_command'],
                             str(a['sleep_time'])])
    proc.communicate()
    return {'complete' : True}

  return sleep_function
```

Figure 4.19: `sleep.py`: An example Python wrapper for PCL.

An example of a complete Python wrapper file is shown in Figure 4.19. This wrapper is the Python implementation of the imperative PCL `sleep` component shown in Figure 4.2.