Version 1.0 February 13, 2016

The ProcessJ Language

by Matt B. Pedersen

1 Introduction

2 The Grammar

2.1 A Compilation Unit

A ProcessJ file constitutes a *compilation unit*. A compilation unit consists of the following four things:

- An optional list of pragmas (of the form `pragma ...`). Pragmas are used to pass options to the compiler when compiling a specific file. Currently, pragmas are only utilized in the library generating system – see Section 2.2.
- An optional package declaration of the form `package <name>`. A package name should, like in Java, correspond to the directory path in which the file is located. See Section **??** for more information on packages.
- An optional list of import statements of the form `import ....`. importing a file allows you to use types and procedures (which are also types) from other compilation units. See Section **??** for general information on imports, Section **??** for implicit imports though the use of qualified named, and Section **??** for more information about how name resolution is handled with the presence of import statements.
- An optional list of type declarations. This includes the definition of records, protocols, and procedures.

compilation_unit → *pragmas** [*package_declaration*]
import_declarations* *type_declarations**

Example:

```
package Example;

import std.io;

public record R {
  int a;
  double b;
}

public proc void foo(int a) {
  a = a + 1;
}
```

2.2 Pragmas

A **pragma** is a statement that is used to pass information to the compiler. Currently pragmas are used to instruct the compiler to generate certain types of code when generating new libraries. A pragma can take one of two shapes. The first is of the form

```
#pragma ID
```

where `ID` is an identifier – a flag – that is passed to the compiler, or

```
#pragma ID "string constant"
```

where `ID` is an identifier that is associated with the `"string constant"` and passed to the compiler.

The complete list of valid pragmas is:

- `#pragma LIBRARY;` – this instructs the compiler that the ProcessJ file is meant to be a library file.
- `#pragma NATIVE "string";` – this declares that the library is mapped directly to a natively written library represented by `"string"`. One example could be `"math.h"`, which would tell the compiler that the implementation of this library can be found in an existing library file for the target language (in this example, C).
- `#pragma LANGUAGE "string";` – this sets the implementation language of the library. Naturally, there is a connection between the language chosen and the correct use of the other pragmas.
- `#pragma FILE "string";` – this sets the name of the output file name in the target language.

Much more information about the use of pragmas can be found in Section **??** on how to write libraries for ProcessJ.

2.3 Package Declarations

2.4 Import Declarations

An import in ProcessJ is a way to make available code written in another compilation unit without re-implementing it. There are two forms of explicit imports in ProcessJ, one where a single compilation unit (file) is imported and one where all compilation untils (files) in a package are imported. It is not possible to import one single type from a compilation unit, but it is possible to explicitly use a type in a non-imported compilation unit through the `::` operator. However, such a use forces an import of the compilation unit (file) in which the type is located, but without making available the remaining types of that compilation unit to the program (apart from to the imported type). The two forms of the import statement look like this:

```
import std.io;
```

which imports the compilation unit `io` found in the `std` package, and

```
import std.*;
```

which imports **all** the compilation units in the `std` package. The `*` option should be utilized with care as it can potentially cause a lot of I/O for the compiler as it has to read and parse a lot of files.

Importing compilation units (which henceforth shall be called just 'files') builds a hierarchy for the name resolution phase to traverse. This hierarchy can be rather complicated to understand, especially for the compiler writer, but the following rules should be noted for the regular programmer:

- Types declared in later imports are preferred to types in earlier imports. This means that the order if import statements determines (in reverse order) which type is found in the name resolution phase when a type name is used. Again, to explicitly override this the `::` operator can be used in declarations or invocations of procedures.
- Files imported by explicitly imported files are not made available to the programmer, so if your program imports a file `A` and `A` imports a file `B` which contains a type `T`, then you cannot refer to `T` without importing `T` explicitly.

Section **??** explain the implementation of the import system in greater detail.

2.5 Type Declarations

A ProcessJ file can contains any number of type declarations. In ProcessJ, a procedure is considered a type as well, the reason is that procedures, when run, are considered processes, and ProcessJ supports mobile processes; that is, processes can be communicated between other processes as a piece of data.

At the top-level ProcessJ supports 5 different declarations:

- Procedure declarations.
- Protocol declarations.
- Record declarations.
- Constant declarations.
- Extern type declarations.

Before we consider the 5 different top-level declarations (of which the first three also are type constructors) we must briefly consider modifiers and primitive types.

2.5.1 Modifiers

ProcessJ currently supports 6 different modifiers. Modifiers are keywords that can be placed in front of type and variable declarations. Not all modifiers can be used on all types. The list of modifiers can be seen in Table 1

If no modifiers are used, the `protected` modifier is the default chosen by the compiler. This means that any type without a modifier will be visible only within the package.

Modifier	Description
`mobile`	The `mobile` modifier can only be used on procedure types, channel, and channel end declarations. A mobile process is a process that can be communicated in a channel that carries the appropriate procedure type. A mobile channel is a channel whose ends can be sent over a channel, and a mobile channel end denoted the channel ends of a mobile channel.
`const`	The modifier `const` can only be used to declare constants at the compilation unit level like `const double PI = 3.1415` or it can be used to declare a local variable or even a parameter constant. A constant local cannot be assigned apart from its initializer, and a constant parameter cannot be assigned to at all apart from the value it gets when the procedure is called.
`native`	The `native` modifier does not appear in normal code. It is used only when writing native libraries for ProcessJ.
`public`	Only top-level types (that is, types declared outside procedures) can use the `public` modifier. The `public` modifier makes a type available to anyone who either imports the file in which it is declared or who implicitly causes an import by referring to the type using the `::` operator.
`private`	Like `public`, `private` is a top-level modifier. Any type declared `private` can only be accessed within the compilation unit in which it was declared.
`protected`	Again, `protected` is a top-level modifier only. A `protected` type is accessible only within the package.

Table 1: ProcessJ modifiers.

Name	Representation	Range
`boolean`		{`true`, `false`}
`byte`	8-bit signed 2's complement	{-128,...,127}
`short`	16-bit signed 2's complement	{-32768,...,32767}
`char`	16-bit Unicode character	{'\u0000',...,'\ufff'}
`int`	32-bit signed 2's complement	$\{-2^{31}, \ldots, 2^{31}-1\}$
`long`	64-bit signed 2's complement	$\{-2^{63}, \ldots, 2^{63}-1\}$
`float`	single-precision 32-bit IEEE 754 floating point	$\pm 1.18 \times 10^{38}$ - $\pm 3.4 \times 10^{38}$
`double`	double-precision 64-bit IEEE 754 floating point	$\pm 2.23^{308}$ - $\pm 1.80 \times 10^{308}$
`string`		–
`barrier`		–
`timer`		–

Table 2: ProcessJ primitive types.

2.5.2 Primitive Types

ProcessJ supports all the typical primitive types known from most other programming languages. These include integral types of various sizes, floating point types, booleans, and strings. In ProcessJ a string is a primitive type. In addition, there are two further primitive types, namely `barrier` and `timer`. These are probably new to you, but they are essential in a process oriented language (or at least the barrier is – the timer is nice to have). Table 2 contains a list of the primitive types. Apart from the `barrier` and the `timer` types, all the other primitive types should be well known from other languages.

Barriers. A `barrier` is a *synchronization point* in a ProcessJ program. A process can `enroll` on a barrier, which means that it may synchronize on it (using the `sync` keyword). When a process synchronizes on a barrier, it will be held at that barrier until every other process enrolled on that barrier also synchronizes on it. Only when all enrolled processes have called `sync` on the barrier in question will every process be allowed to proceed past the `sync` call. If just one of the enrolled processes does not call `sync`, all other processes will be prevented from progressing in their execution. Barriers can be passed like any other primitive value to procedures, but cannot be sent over channels; that is, barriers cannot be declared `mobile`.

Timers. A `timer` is a data type that represents a clock that ticks. A timer variable can be *read*, much like a channel:

```
timer t;
```

```
int time;

time = t.read();
```

It should be noticed that while regular channel reads are synchronous, that is, the sender and the reader must both be ready to communicate (otherwise neither can progress, and the process is suspended), a timer read is always ready and can never cause a process to be suspended. A timer read returns an integer value. The second operation you can perform on a timer is a timeout:

```
timer t;
int time;

t.timeout(100);
```

Invoking a `timeout` on a timer prevents the process from progressing until the specified amount of time has passed. Timeout statements can also be used as guards in an alt-statement. Like barriers, timers cannot be communicated on channels; they cannot be declared mobile.

2.5.3 Record Declarations

A record in ProcessJ is like a struct in C. The EBNF grammar for a record declaration is as follows:

record_type_declaration	→	*modifier** **record** ID **{**
		(*type variable_id* (**,** *type variable_id*)* **;**)$^+$
		}
variable_id	→	ID
		variable_id [ ]

An example of a ProcessJ record is:

```
public record Client {
  string first_name, last_name;
  string address1, address2, city;
  int zip;
  Transaction transactions[];
}
```

2.5.4 Protocol Declarations

The protocol type constructor in ProcessJ has many similarities to a union data type in C. It consists of a number of tag-named variable lists. Here is an example:

```
public protocol P {
  request: { int number; double amount; }
  reply: { bool status; }
}
```

A variable of type `P` contains **either** an integer `number` and a double `amount` while being tagged as `request` **or** a boolean `status` while being tagged `reply`.

One major difference between unions in C and protocols in ProcessJ is that protocols can *extend* existing protocols. For example, consider this example:

```
public protocol P2 extends P {
  deny: { int code;}
}
```

which declares a protocol type `P2` which *inherits* the `request` and `reply` cases from `P` and further extends the type by adding a new case called `deny`.

A protocol can extend as many other protocols as you want, so the following code is legal too:

```
public protocol P7 extends P6, M2, Y6 ;
```

Here, `P7` will contain all the cases from `P6`, `M2`, and `Y6`; if any of those three protocol types have similar tags an error will be produced by the compiler. The grammar for declaring protocol types look like this:

protocol_type_declaration → *modifier** **protocol** ID [**extends** ID (**,** ID)*]
(
{
(ID **:** **{** (*type* ID **;**)* **}**)*
}
| **;**
)

A protocol body can be empty but only if it extends at least one other protocol. However, it is perfectly OK for a protocol case not to have any declarations in its list. That is often useful if a case simply serves to mark a certain choice that does not need to carry any data, like for example an acknowledgment.:

```
public protocol P {
  request: { int package_no; }
  ack: {}
}
```

2.5.5 Constant Declarations

A constant declaration at the top-level is like a local variable declaration with the `const` modifier prefixed. For example:

```
public const double PI = 3.1415;
```

declares a public constant called `PI`. Top-level constants can only be declared of primitive types that are not barrier or timer types. Note, the use of the modifier `const` for local variables or parameters simply means they cannot be assigned, and they can be of any type.

2.5.6 Procedure Declarations

2.5.7 Native Type Declarations

Who knows.

2.6 Other Types

2.6.1 Arrays

2.6.2 Channels

A *channel* is a new concept for many programmers, but is central to the concepts of process oriented design. A channel, at its simplest incarnation, is nothing but a "piece of garden hose" through (though we say 'on') which data can flow in one direction (channels are not bi-directional). Other versions of channels can have shared ends, which in the garden-hose analogy means that ends have spreaders on. Rather than water flowing in the hose think of marbles. If the receiving end is shared the marble will eventually end up going down one of the end of the spreader and only one receive will get it - it does not replicate itself like in a traditional broadcast. Here is an example of a declaration of a channel:

```
chan<int> c;
```

This declares a channel variable called `c` that carries integer values. Note the use of the `<>`. Channels must be declared to carry a type, and almost any type can be used.

Channels cannot be passed as parameters to a procedure and they cannot be communicated on other channels either. For mobile channels see the next section on channel ends.

2.6.3 Channel Ends

A channel consists of two ends: a reading end and a writing end. You can only read from the reading end and you can only write to the writing ends. any channel variable's ends can be obtained by suffixing either `.read` or `.write`. So the following code is legal:

```
public void foo(chan<int>.read in) {
  ...
}

public void main(string args[]) {
  chan<int> c;

  par {
    ...
    foo(c.read);
  }
}
```

We will return to the `par` statement later in Sectionsec:block. Note the declaration of the parameter `in`: `chan<int>.read` is a *type*, namely the reading channel end of a channel carrying integer values. Similarly, the writing channel end type is obtained by the `.write` postfix. To declare shared ends of a channel we have three options: either both are shared, only the reading end is shared, or only the writing end is shared. Here are examples of the declarations of all three types:

```
shared chan<int> sharedReadWriteChan;
shared read chan<int> sharedReadChan;
shared write chan<int> sharedWriteChan;
```

Similarly, in a procedure declaration's parameter list we can specify how a channel's ends are shared – however, the use of the `shared` modifier applies only to the end in question:

```
public proc void foo(shared chan<int>.read sharedReadingChannelEnd,
                     shared chan<int>.write sharedWritingChannelEnd) {
  ..
}
```

The idea of both ends being shared when talking about a channel end makes no sense, and since channels cannot be passed parameters, the "both ends shared" option does not apply for parameters.

Shared channel ends must be *claimed* before being used. This is achieved using the `claim` keyword. See section 2.7.15 for more information.

2.7 Statements

2.7.1 Block Statement

In general, a block in ProcessJ is a number of statements surrounded by a set of { }. A block also introduces a new scope, and blocks can be nested.

A block is implicitly *sequential*, so each statement in a block is executed in order from top to bottom. However, in ProcessJ, it is sometimes possible to declare a block *parallel*. This is done by prefixing the word `par` to the block. For example:

```
...
par {
   foo();
   bar();
}
...
```

The above example shows a par-block with two statements, namely the invocations of the functions `foo` and `bar`. A par-block makes each statement a process that runs concurrently with the other statements in the par-block, and the entire par block does not terminate until every process of the block has terminated.

It is not possible to declare variables in a par-block for a number of reasons: it is not possible to read and write the same variable in a par-block as that causes a race condition on the specific variable. If a variable can only be read or written, then a newly defined variable can only be written through its initializer or an assignment and never read, and since the variable goes out of scope at the end of the block, such an endeavor would be meaningless.

Furthermore, a par-block can enroll its processes on zero or more barriers in the following way:

```
...
par enroll (b,c) {
  foo(b,c);
  bar(b,c);
  baz(b,c);
}
...
```

which executes `foo`, `bar`, and `baz` concurrently while enrolling all three processes on the barriers `b` and `c`. Note, all three processes are passed the barriers on which they are enrolled. If we did not do that they could not synchronize on the barriers, and if just one of them does not synchronize on a barrier it is enrolled on, then none of the other processes enrolled on that barrier can move past their synchronization point, and we have a potential for a deadlock.

A par block can only appear inside the body of a procedure as a block statement.

The grammar rules for regular blocks and par-blocks are:

$$\begin{array}{lcl} \textit{block} & \rightarrow & \textbf{\{}\ (\textit{block\_statements})^*\ \textbf{\}} \\ \textit{par\_block} & \rightarrow & \textbf{par}\ [\ \textbf{enroll}\ \textbf{(}\ (\textit{expression})^*\ ]\ \textbf{)}\ \textit{block} \end{array}$$

The list of expressions in the enroll clause must be of barrier type, and even though the par-block uses the rule for a block, local variable declarations will be prevented by the compiler.

2.7.2 Do Statement

A do statement in ProcessJ is exactly like in Java. The grammar for a do statement is:

$$\textit{do\_statement} \rightarrow \textbf{do}\ \textbf{(}\ \textit{expression}\ \textbf{)}\ \textit{statement}$$

2.7.3 While Statement

Like Java

2.7.4 For Statement

Like Java

2.7.5 Expression Statement

Like Java

2.7.6 Break & Continue Statements

2.7.7 Return Statement

Like Java

2.7.8 Switch Statement

Like Java, except must be used for Protocol access.

2.7.9 Stop & Skip Statements

`stop` and `skip` are two new keywords that most programmers may not be familiar with. `skip` is a no-op operation - it does not do anything – it could mostly be replaced by a semicolon.

`stop` does exactly that - it stops, but does not terminate. `stop` is equivalent to `for (;;) ;` – an infinite loop that never does anything.

2.7.10 Barrier Synchronization Statement

To synchronize on a barrier we use the `sync` keyword. For example:

```
sync(b);
```

synchronizes on a barrier named `b`.

Remember, no process can progress past a barrier synchronization until every barrier who has enrolled on it has

2.7.11 Timer Read / Timeout Statement

Remember, a `timer` is an ever ticking clock. A timer can be *read* much like a channel can:

```
timer t;
long time = t.read();
```

A read of a timer returns a long value. A timer can also be used as an 'alarm clock', that is, it can be set to block until it times out after a preset time. For example, if we want a timer to stop execution for 1 second we can write code like this:

```
timer t;
t.timeout(1000000); // time for timeout is measured in milliseconds
```

Code like the above is typically not how a timeout is used. It is often used as a case in an alt-statement such that if no other case is ready the alt-case can time out.

2.7.12 Suspend Statement

2.7.13 Channel Write Statement

A channel write statement is simply a write to a channel. This is done like this:

```
chan<int> c;

...
c.write(42);
```

The last line writes the value 42 to the channel `c`. Actually, in reality it writes 42 to the *writing end* of the channel `c`, and therefore the correct way to write the last line from the example above ought to be:

```
c.write.write(42);
```

where `c.write` is an expression that evaluates to the writing end of the channel `c` to which we can then write 42 by calling `write`. The compiler allows both way as the double 'write' seems redundant. However, the following code is perfectly legal:

```
chan<int> c;

chan<int>.write cw = c.write;
cw.write(42);
```

2.7.14 Alt Statement

Another new construct that C or Java does not have is *alternation* (or *alt* for short. An alt statement consists of a number of *guarded* statements. To execute an alt, each guard is evaluated, and of the guards that are *ready* one is chosen at random and its corresponding statement is executed.

The grammar for an alt statement is as follows:

$$
\begin{array}{lcl}
\textit{alt\_statement} & \rightarrow & [\ \textbf{pri}\ ]\ \textbf{alt}\ \textbf{\{} \\
 & & \quad ([\ \textbf{(}\ \textit{expression}\ \textbf{)}\ \textbf{\&\&}\ ]\ \textit{guard}\ \textbf{:}\ \textit{statement})^{+} \\
 & & \textbf{\}} \\
\textit{guard} & \rightarrow & \textit{left\_hand\_side} = \ \textit{channel\_read\_expression} \\
 & | & \textbf{skip} \\
 & | & \textit{timeout\_statement}
\end{array}
$$

Each guard can be preceded by an optional Boolean expression - if the Boolean expression is false, the guard is not considered. The selection process for which statement of an alt to execute follows the following algorithm:

1. For each case of an alt statement do the following: Ff the there is a boolean expression and it evaluates to *true*, then check if the guard is *ready*:
   - A skip guard is *always* ready.
   - A timeout is ready if the amount of time given in the timeout has elapsed since the alt was evaluated the first time[1].
   - A channel read guard is ready if there is a committed sender at the other end of the channel, that is, if a communication of a piece of data on the channel is ready to proceed once a read operation is started.
2. From the set of guarded statements for which the guard is ready, if the alt is a prioritized alt, pick the fist one, and if not, pick one at random.
3. If no guards are ready, the alt blocks until such time that at least one guard is ready.

If no guards are ready, the alt statement is blocked until such time that at least one of the guards become ready. Since a skip guard is always ready, it can serve as default option in an alt statement. If the optional *pri* key word is used, the alt then becomes a prioritized alt in which guarded statement that appear lexicographically earlier than others will be chosen if ready over other ready statements appearing later.

```
alt {
  x = c1.read() : { ... }
  y = c2.read() : { ... }
  t.timeout(700) : { ... }
}
```

or

[1]This is highly unlikely to be the case the first time around, but if no other guards are ready, the process is descheduled, and when it gets rescheduled, enough time might have elapsed for the timeout to have happened.

```
pri alt {
  x = c.read() : { ... }
  skip : { ... }
}
```

In the latter example, if channel `c` is not ready to read, the statement following the skip guard will execute by default.

2.7.15 Claim Statements

A claim statement is simply a statement that lists a number of shared channel ends used by the statement following. For example:

```
claim (c, b2[i]) {
  a = c.read();
  b2[i].write(a);
}
```

only shared channel ends should be claimed.

The grammar for a claim statement is:

claim_statement	→	**claim (** *channel_end* (**,** *channel_end*)* **)** *statement*
channel_end	→	ID
	|	*channel_type* ID = *expression*
	|	*expression* **.** (**read** | **write**)

2.8 Expressions

2.8.1