ccaPL: a Programming Language for the Calculus of Context-aware Ambients

François Siewe
Software Technology Research Laboratory
De Montfort University, Leicester, United Kingdom

October 21, 2012

Abstract

The Calculus of Context-aware Ambients (CCA) was proposed as a formalism for modelling context-aware systems. The main features of the calculus include mobility, context-awareness and concurrency. This technical report presents a programming language for CCA called ccaPL. This language provides textual notations for representing CCA processes. A ccaPL interpreter is implemented in java and allows for the simulation and animation of CCA processes.

1 Overview of CCA

The concept of ambient is central in CCA; it is an abstraction of a place where computation may happen. An ambient can be mobile and can contain other ambients called child ambients organised in a tree structure. Such a hierarchy can be used to model any entity in a pervasive system –whether physical, logical, mobile or immobile– as well as the environment (or context) of that entity. In addition to child ambients, an ambient can also contain a process specifying the capabilities of that ambient, i.e. the actions the ambient is allowed to perform, such as mobility capabilities, context-aware capabilities and communication capabilities. Table 1 depicts the syntax of CCA, based on three syntactic categories: processes (denoted by \( P \) or \( Q \)), capabilities (denoted by \( M \)) and context-expressions (denoted by \( \kappa \)). We assume a countably-infinite set of names, elements of which are written in lower-case letters, e.g. \( n \), \( x \) and \( y \). We let \( \tilde{y} \) denote a list of names and \( [\tilde{y}] \) the arity of such a list. We sometimes use \( \tilde{y} \) as a set of names where it is appropriate.

### Table 1: Syntax of CCA

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, Q ) := 0</td>
<td>inactivity process, does nothing and terminates immediately. The process ( P\mid Q ) denotes the concurrent execution of the processes ( P ) and ( Q ). The process ( \nu n \ P ) creates a new name ( n ) and the scope of that name is limited to the process ( P ). The replication ( !P ) denotes a process which can always create a new copy of ( P ), i.e. ( !P ) is equivalent to ( P\mid !P ). Replication can be used to implement both iteration and recursion. The process ( n[P] ) denotes an ambient named ( n ) whose behaviours are described by the process ( P ). The pair of square brackets ‘([\cdot])’ outlines the boundary of that ambient.</td>
</tr>
<tr>
<td>( M ) := ( n \mid \text{out} \mid \alpha \ \text{recv}(\tilde{y}) \mid \alpha \ \text{send}(\tilde{y}) \mid \alpha \ x(\tilde{z}) \mid \text{del} \ n )</td>
<td>Process abstraction ( x\hookrightarrow \kappa )</td>
</tr>
<tr>
<td>( \alpha ) := ( \uparrow \mid \downarrow \mid \uparrow \downarrow \mid :: \mid \epsilon )</td>
<td>A context expression specifies a property upon the environment. A context-guarded prefix ( \nu \kappa.M.P ) is a process that waits until the environment satisfies the context expression ( \kappa ), then performs the capability ( M ) and continues like the process ( P ). The dot symbol ‘( . )’ denotes the sequential composition of processes. We let ( M.P ) denote the process ( \text{true}?M.P ), where ( \text{true} ) is a context expression satisfied by all context.</td>
</tr>
<tr>
<td>( \kappa ) := ( \text{true} \mid \cdot \mid \text{out} \mid n = m \mid n[\kappa] \mid \neg \kappa \mid \kappa_1 \land \kappa_2 \mid \kappa \land \kappa_2 \mid \oplus \kappa \mid \odot \kappa \mid \diamond \kappa )</td>
<td>A search prefix ( \text{find} \ x : \kappa \text{ for } P ) is a process that looks for a set of names ( \tilde{n} ) such that the context expression ( \kappa{x \leftarrow \tilde{n}} ) holds and continues like the process ( P[x \leftarrow \tilde{n}] ), where the notation ( {x \leftarrow \tilde{n}} ) means the substitution of ( n_i ) for each free occurrence of ( x_i ), ( 0 \leq i &lt;</td>
</tr>
</tbody>
</table>
process abstraction can be thought of as a definition of function or procedure in imperative programming languages such as C or Java.

**Capabilities** A call to a process abstraction named \( x \) is done by a capability of the form \( \alpha x(z) \) where \( \alpha \) specifies the location where the process abstraction is defined and \( z \) is the list of *actual parameters*. The location \( \alpha \) can be ‘\( \uparrow \)' for any parent, ‘\( n \uparrow \)' for a specific parent \( n \), ‘\( i \)' for any child, ‘\( n \downarrow \)' for a specific child \( n \), ‘\( : \)' for any sibling, ‘\( n :: \)' for a specific sibling \( n \), or \( \epsilon \) (empty string) for the calling ambient itself. A process call \( \alpha x(z) \) behaves like the process linked to \( x \) at location \( \alpha \), in which each actual parameter in \( z \) is substituted for each occurrence of the corresponding formal parameter. A process call can only take place if the corresponding process abstraction is *available* at the specified location.

Ambients exchange messages using the output capability \( \alpha (z) \) to send a list of names \( z \) to a location \( \alpha \), and the input capability \( \alpha (\langle y \rangle) \) to receive a list of names sent from a location \( \alpha \). The mobility capabilities \( \text{in} \) and \( \text{out} \) are defined as follows. An ambient that performs the capability \( \text{in} n \) moves into the sibling ambient \( n \). The capability \( \text{out} \) moves the ambient that performs it out of that ambient’s parent. The capability \( \text{del} n \) deletes an ambient of the form \( n[0] \) situated at the same level as that capability, i.e. the process \( \text{del} n.P \mid n[0] \) reduces to \( P \). The capability \( \text{del} \) acts as a garbage collector that deletes ambients which have completed their computations.

**Example 1.1**

- The process
  \[ n[\text{in} m.\text{out}.0] \mid m[\text{in} n.\text{out}.0] \]
  describes the behaviours of two sibling ambients \( n \) and \( m \) concurrently willing to move in and out of one another.

- The ambient
  \[ n[(\mathcal{Vat}(m))? :: \text{send}(msg).0] \]
  releases the message ‘\( msg \)’ only when at location \( n \); where the context expression \( \mathcal{Vat}(m) \) holds if \( n \) is a child ambient of the ambient \( m \). The formal definition of the predicate \( \mathcal{Vat} \) is given in Example 1.2.

**Context model** In CCA, a context is modelled as a process with a hole in it. The hole (denoted by \( \odot \)) in a context represents the position of the process that context is the context of. For example, suppose a system is modelled by the process \( P \mid n[Q] \mid m[R \mid S] \). So, the context of the process \( R \) in that system is \( P \mid n[Q] \mid m[\odot \mid S] \), and that of ambient \( m \) is \( P \mid n[Q] \mid \odot \). Thus the contexts of CCA processes are described by the grammar in Table 2. Properties of contexts are called context expressions (CEs in short).

**Table 2: Syntax of contexts**

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C ::= 0 \mid \odot \mid n[C] \mid C[P] \mid (\nu n) C )</td>
</tr>
</tbody>
</table>

**Context expressions** The CE \( \text{true} \) always holds. A CE \( n = m \) holds if the names \( n \) and \( m \) are lexically identical. The CE \( \bullet \) holds solely for the hole context, i.e. the position of the process evaluating that context expression. Propositional operators such as negation (\( \neg \)) and conjunction (\( \land \)) expand their usual semantics to context expressions. A CE \( \kappa_1 \mid \kappa_2 \) holds for a context if that context is a parallel composition of two contexts such that \( \kappa_1 \) holds for one and \( \kappa_2 \) holds for the other. A CE \( n[\kappa] \) holds for a context if that context is an ambient named \( n \) such that \( \kappa \) holds inside that ambient. A CE \( \odot \kappa \) holds for a context if that context has a child context for which \( \kappa \) holds. A CE \( \mathcal{Vat} \) holds for a context if there exists somewhere in that context a sub-context for which \( \kappa \) holds. The operator \( \mathcal{Vat} \) is called *somewhere modality* while \( \odot \) is aka *spatial next modality*.

**Example 1.2** We now give some examples of predicates that can be used to specify common context properties such as the location of the user, with whom the user is and what resources are nearby. In these sample predicates we take the view that a process is evaluated by the immediate ambient \( \lambda \) say that contains it.

- \( \text{has}(n) \equiv \odot (\bullet \mid n[\text{true}] \mid \text{true}) \) holds if ambient \( \lambda \) contains an ambient named \( n \)

- \( \text{at}(n) \equiv n[\odot(\bullet \mid \text{true})] \mid \text{true} \) holds if ambient \( \lambda \) is located at an ambient named \( n \)

- \( \text{with}(n) \equiv n[\text{true}] \mid \odot(\bullet \mid \text{true}) \) holds if ambient \( \lambda \) is (co-located) with an ambient named \( n \)

\(^1\)The symbol ‘\( \equiv \)’ stands for ‘defined as’. 

2
2 Translation of CCA processes into ccaPL

The translation from CCA to ccaPL is straightforward as shown in Table 3, 4, 5 and 6. However, note that ccaPL interpreter is case-sensitive, e.g. “hello” and “Hello” are treated as distinct names.

Table 3: Translation of processes

<table>
<thead>
<tr>
<th>CCA</th>
<th>ccaPL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P, Q ::= 0$</td>
<td>$0$</td>
<td>inactivity</td>
</tr>
<tr>
<td>$n[P]$</td>
<td>$n[P]$</td>
<td>ambient</td>
</tr>
<tr>
<td>$(vn) P$</td>
<td>new $n \ P$</td>
<td>restriction</td>
</tr>
<tr>
<td>$P</td>
<td>Q$</td>
<td>$P\parallel Q$</td>
</tr>
<tr>
<td>$!P$</td>
<td>$!P$</td>
<td>replication</td>
</tr>
<tr>
<td>$\kappa?M.P$</td>
<td>$\langle\kappa&gt;M.\ P$</td>
<td>context-guarded prefix</td>
</tr>
<tr>
<td>$x\bowtie(\tilde{y}).P$</td>
<td>proc $x(\tilde{y})\ P$</td>
<td>process abstraction $x$</td>
</tr>
<tr>
<td>${P}$</td>
<td>${P}$</td>
<td>brackets</td>
</tr>
<tr>
<td>find $x:\kappa\ for\ P$</td>
<td>find $x:\kappa\ for\ P$</td>
<td>search prefix</td>
</tr>
</tbody>
</table>

Table 4: Translation of capabilities

<table>
<thead>
<tr>
<th>CCA</th>
<th>ccaPL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M ::= \text{in}\ n$</td>
<td>$\text{in}\ n$</td>
<td>move into the ambient $n$</td>
</tr>
<tr>
<td>$\text{out}$</td>
<td>$\text{out}$</td>
<td>move out</td>
</tr>
<tr>
<td>$\text{del}\ n$</td>
<td>$\text{del}\ n$</td>
<td>delete the ambient $n$</td>
</tr>
<tr>
<td>$\alpha x(\tilde{z})$</td>
<td>$\alpha x(\tilde{z})$</td>
<td>call to the process abstraction $x$</td>
</tr>
<tr>
<td>$\alpha (\tilde{y})$</td>
<td>$\alpha \text{recv}(\tilde{y})$</td>
<td>receive list of messages $\tilde{z}$ from $\alpha$</td>
</tr>
<tr>
<td>$\alpha (\tilde{z})$</td>
<td>$\alpha \text{send}(\tilde{z})$</td>
<td>send list of messages $\tilde{z}$ to $\alpha$</td>
</tr>
</tbody>
</table>

Table 5: Translation of location notations

<table>
<thead>
<tr>
<th>CCA</th>
<th>ccaPL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\uparrow$</td>
<td>$\uparrow$</td>
<td>any parent</td>
</tr>
<tr>
<td>$n \uparrow$</td>
<td>$n@$</td>
<td>parent $n$</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>$#$</td>
<td>any child</td>
</tr>
<tr>
<td>$n \downarrow$</td>
<td>$n#$</td>
<td>child $n$</td>
</tr>
<tr>
<td>::</td>
<td>::</td>
<td>any sibling</td>
</tr>
<tr>
<td>$n ::$</td>
<td>$n ::$</td>
<td>sibling $n$</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>$\epsilon$</td>
<td>locally</td>
</tr>
</tbody>
</table>
Table 6: Translation of context-expressions

<table>
<thead>
<tr>
<th>CCA</th>
<th>ccaPL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>κ := true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>n = m</td>
<td>n = m</td>
<td>name match</td>
</tr>
<tr>
<td>• this</td>
<td>name hole</td>
<td></td>
</tr>
<tr>
<td>¬κ</td>
<td>not κ</td>
<td>negation</td>
</tr>
<tr>
<td>κ₁ ∣ κ₂</td>
<td>κ₁ ∣ κ₂</td>
<td>parallel composition</td>
</tr>
<tr>
<td>κ₁ ∨ κ₂</td>
<td>κ₁ or κ₂</td>
<td>disjunction</td>
</tr>
<tr>
<td>κ₁ ∧ κ₂</td>
<td>κ₁ and κ₂</td>
<td>conjunction</td>
</tr>
<tr>
<td>⊕κ</td>
<td>next κ</td>
<td>spatial next modality</td>
</tr>
<tr>
<td>⊤κ</td>
<td>somewhere κ</td>
<td>somewhere modality</td>
</tr>
</tbody>
</table>

2.1 Boolean Expressions (ver. 4.0 and later)

In addition to the match operator ‘=’, the following predicates can be used:

- ‘<’: less than; e.g. \( n < m \)
- ‘\(<=\)’: less than or equal; e.g. \( n \leq m \)
- ‘>’: greater than; e.g. \( n > m \)
- ‘\(\geq\)’: greater than or equal; e.g. \( n \geq m \)

2.2 Arithmetic Expressions (ver. 4.0 and later)

Arithmetic expressions can be used in a ‘let’ statement as follows, where \( P \) is a process and, \( x_i \) and \( e_i \), \( 1 \leq i \leq n \) are variable symbols (possibly free in \( P \)) and arithmetic expressions, respectively:

\[
\text{let } x_0 = e_0, x_1 = e_1, \ldots, x_n = e_n \text{ in } P.
\]

An arithmetic expression has the following syntax:

\[
e ::= x \quad \text{a name: string or integer} \\
\quad | - e \quad \text{unary minus} \\
\quad | e_1 + e_2 \quad \text{addition of integers or concatenation of strings} \\
\quad | e_1 - e_2 \quad \text{minus} \\
\quad | e_1 \times e_2 \quad \text{multiplication} \\
\quad | e_1 \% e_2 \quad \text{modulo} \\
\quad | e_1 / e_2 \quad \text{division}
\]

Example 2.1

- \( \text{let } x = 3 + 5 \times 7 \text{ in send}(x).0 \)
- \( \text{recv}(y, z).\text{let } x = 3 + 5 \times y, t = y + 2 \times z \text{ in send}(x, t).0 \)
- \( \text{recv}(y, z).\text{let } x = y + z \text{ in } .x[0] \)

2.3 Example 1: a One-Place Buffer

A one-place buffer is a data structure that operates through two actions \( \text{push} \) and \( \text{pull} \), respectively putting one item in the buffer and taking one item from it. The buffer is full when it contains one item. It is impossible to put one item into a full buffer; it is impossible to take one item from the empty buffer. Let us assume that at the beginning the buffer is empty; the buffer behaves as follows: at first only the \( \text{push} \) operation is possible, after \( \text{push} \) is performed (the buffer contains one item), it is possible to perform \( \text{pull} \). We model the buffer as an ambient \( \text{buf} \) as follows:

\[
\text{buf} \left[ \begin{array}{c}
\uparrow(x).\langle x \rangle.0 \\
\downarrow(y).\uparrow(y).\uparrow(z).\langle z \rangle.0
\end{array} \right]
\]
This ambient waits for its parent to perform push, and once a push action has been performed the ambient waits for its parent to perform a pull action. The corresponding ccaPL program is the following:

```plaintext
buf[
    @recv(x).send(x).0 |
    !recv(y).@send(y).@recv(z).send(z).0
]
```

### 2.4 Example 2: a Network Hub

A network hub is a device for connecting multiple Ethernet devices together and making them act as a single network segment (Wikipedia, 04/12/2011). It has a fixed number of ports and any packet entering any port is simply broadcast on all other ports. A network hub with 4 ports can be specified in CCA as follows:

```plaintext
hub

```

where $p_i$ is the name of the ambient modelling the device connected to the port number $i$.

The corresponding ccaPL program is:

```plaintext
hub[
    !p1::recv(x).{p2::send(x).0 | p3::send(x).0 | p4::send(x).0} |
    !p2::recv(x).{p1::send(x).0 | p3::send(x).0 | p4::send(x).0} |
    !p3::recv(x).{p1::send(x).0 | p2::send(x).0 | p4::send(x).0} |
    !p4::recv(x).{p1::send(x).0 | p2::send(x).0 | p3::send(x).0}
]
```

### 2.5 Example 3: a Context-aware Mobile Phone

Let’s suppose a context-aware mobile phone that automatically switches into silent mode (can vibrate but cannot ring) when in a conference room and to normal mode (can vibrate and ring) when outside.

```plaintext
phone
```

where ‘switch’ is a process abstraction and ‘conf’ is an ambient modelling the conference room.

The corresponding ccaPL program is:

```plaintext
phone[
    <somewhere(conf[<somewhere(this | true)] | true)>switch(silent).0 |
    <not somewhere(conf[<somewhere(this | true)] | true)>switch(normal).0
]
```

### 2.6 Example 4: Mobility

Let’s consider two ambient $n$ and $m$, willing to move in and out of each other as follows:

```plaintext
n[in m.out.0] | m[in n.out.0]
```

The corresponding ccaPL is identical.

### 2.7 Example 5: Context Provision/Acquisition using Process Abstractions

Consider a context-aware system which enables a mobile software agent willing to edit a text file to do so on any host computer using an appropriate text editor for that computer operating system. This can be modelled in CCA using the concept of process abstraction. Suppose there are two types of computer in the system: some running Windows operating systems and others running Linux. Each computer running Windows is configured to use notepad as default text editor while a computer running Linux uses emacs by default. Let’s denote by win an ambient modelling a computer running Windows and by lin one running Linux. Each of these ambients contains a process abstraction edit that maps to the default text editor and the mobile agent just has to call that process abstraction to edit a file using the local text editor as specified below.
• A Windows computer uses *notepad* as default text editor:

\[
\text{win}[\text{edit} \triangleright (f).\text{notepad}(f).0]
\]

• A Linux computer uses *emacs* as default text editor:

\[
\text{lin}[\text{edit} \triangleright (f).\text{emacs}(f).0]
\]

• The mobile agent is willing to edit the text file *foo* on the host computer:

\[
\text{agent}[\uparrow \text{edit}(\text{foo}).0]
\]

• If the host is *win* then notepad will be used to edit the file:

\[
\text{win} \left[ \text{edit} \triangleright (f).\text{notepad}(f).0 \mid \text{agent}[\uparrow \text{edit}(\text{foo}).0] \right]
\]

• Otherwise emacs will be used instead:

\[
\text{lin} \left[ \text{edit} \triangleright (f).\text{emacs}(f).0 \mid \text{agent}[\uparrow \text{edit}(\text{foo}).0] \right]
\]

The corresponding *ccaPL* program when the host is *lin* is:

\[
\begin{align*}
\text{lin}[ & \\ & \begin{array}{l}
\text{proc edit}(f) \{ \\
\text{emacs}(f).0 \\
\mid \\
\text{agent}[ \\
\text{@edit}(\text{foo}).0 \\
\} \\
\end{array}\end{align*}
\]

### 2.8 Example 6: Parameterised Processes

Process abstractions can also be used to represent parametrised processes, i.e. processes that carry a set of parameters. For example, let’s consider a process that connects a set of devices \(p_1, p_2, p_3\) and \(p_4\) into a local network using a hub like the one described in Sect. 2.4. Such a process can be represented in *CCA* as a process abstraction defined by:

\[
\text{connect} \triangleright (\text{hubID}, p_1, p_2, p_3, p_4).\text{hubID} \rightarrow \left[ \begin{array}{c}
\text{connect}(\text{hub}, PC_1, PC_2, PC_3, PC_4) \\
| PC_1[\text{hub} :: \text{recv}(x).0] \\
| PC_2[\text{hub} :: \text{recv}(x).0] \\
| PC_3[\text{hub} :: \text{recv}(x).0] \\
| PC_4[\text{hub} :: \text{recv}(x).0 \mid \text{hub} :: \text{send}(\text{hello}).0]
\end{array} \right]
\]

where the names \(\text{hubID}, p_1, p_2, p_3\) and \(p_4\) are formal parameters, place holders for the hub and the connected devices respectively.

This process abstraction can then be called upon actual parameters to connect a set of devices in a local network. For example, the following process connects 4 PCs through a hub; each PC willing to send or to receive network packets.

\[
\text{connect}(\text{hub}, PC_1, PC_2, PC_3, PC_4) \\
| PC_1[\text{hub} :: \text{recv}(x).0] \\
| PC_2[\text{hub} :: \text{recv}(x).0] \\
| PC_3[\text{hub} :: \text{recv}(x).0] \\
| PC_4[\text{hub} :: \text{recv}(x).0 \mid \text{hub} :: \text{send}(\text{hello}).0]
\]

\(PC_4\) sends a ‘hello’ message to the network; all other nodes will eventually receive that message. The corresponding *ccaPL* program is:
proc connect(hubId, p1, p2, p3, p4) {
    hubID[
        !p1::recv(x).{p2::send(x).0 | p3::send(x).0 | p4::send(x).0} |
        !p2::recv(x).{p1::send(x).0 | p3::send(x).0 | p4::send(x).0} |
        !p3::recv(x).{p1::send(x).0 | p2::send(x).0 | p4::send(x).0} |
        !p4::recv(x).{p1::send(x).0 | p2::send(x).0 | p3::send(x).0}
    ]
}
| connect(hub, PC1, PC2, PC3, PC4)
| PC1[!hub::recv(x).0]
| PC2[!hub::recv(x).0]
| PC3[!hub::recv(x).0]
| PC4[!hub::recv(x).0 | hub::send(hello).0]

3 Structure of ccaPL programs

The general structure of a ccaPL program is depicted in Table 7. So a program has two parts: the declaration part and the body part. The body part is a ccaPL process as described in Section 2. The declaration part is optional and begins with the keyword BEGIN_DECLS and ends with the keyword END_DECLS. Between these two keywords, one can insert zero or more declaration statements and execution directives. A declaration statement defines a context-expression as a Boolean function with a name and a parameter list, while an execution directive tells the ccaPL interpreter how the program must be executed. In addition, any text between the markers "/*" and "*/" is considered a comment and is ignored by the interpreter. So is any text following the symbol "//" till the end of the line.

| Table 7: General structure of a ccaPL program |
| BEGIN_DECLS | declaration block (optional) |
| END_DECLS | body of the program (mandatory) |

Example 3.1 A ccaPL program with declarations:

BEGIN_DECLS
    def has(n) = { this | n[true] | true}
    def isHello(n) = {n = hello}
END_DECLS

send(hello).recv(ack).0
| recv(x).<isHello(x)>send(thanks).0

Example 3.2 A ccaPL program with declarations and executions directives:

BEGIN_DECLS
    def has(n) = { this | n[true] | true}
    def isHello(n) = {n = hello}

    mode random // default mode is 'interleaving'.
    display code // add this line to display code
    // after each reduction.
    //display congruence // add this line to include congruence
    // in the output.
END_DECLS

send(hello).recv(ack).0
Example 3.3 A ccaPL program without declarations nor executions directives:
BEGIN_DECLS
SEND_DECLS
send(hello).recv(ack).0
recv(x).<x = hello>send(thanks).0

or simply:
send(hello).recv(ack).0
recv(x).<x = hello>send(thanks).0

3.1 Declaration statements
A declaration statement has the following syntax:

\[
\text{def } \text{name}(par_1, par_2, \ldots, par_n) = \{\kappa\}
\]

where

- \text{def} is a keyword meaning function definition
- \text{name} is the function’s name
- \text{par}_i, 1 \leq i \leq n is the \text{i}^\text{th} parameter of the function (no parameter if \text{n} = 0)
- \kappa is the context expression denoted by this function and is written in the syntax given in Table 6
- The pair of parentheses and the pair of curly brackets are mandatory.

Here are couple of examples:

- The negation of the context expression \text{true} can be defined as:
  \[
  \text{def } \text{false}() = \{\text{not true}\}
  \]

- The context expressions given in Example 1.2 are defined in ccaPL as follows:
  \[
  \begin{align*}
  \text{def } \text{has}(n) &= \{\text{next (this | n[true] | true)}\} \\
  \text{def } \text{at}(n) &= \{n[\text{next (this | true)}] | \text{true}\} \\
  \text{def } \text{with}(n) &= \{n[\text{true}] | \text{next (this | true)}\}
  \end{align*}
  \]

3.2 Execution directives
There are two directives to control the execution of ccaPL programs: \text{mode} and \text{display}.

\text{Mode:} Because processes of a program are executed concurrently in an interleaving manner, the directive \text{mode} specifies the strategy for fair execution of these processes. Process fairness means that if a process is enabled (i.e. ready to perform a capability) infinitely often then that process is also executed infinitely often. In other words, an enabled process cannot wait indefinitely to be executed. So each process must be given a fair chance of being executed. The directive \text{mode} takes the unique parameter \text{random} in the following syntax: \text{mode random}. When this directive is specified, at each execution step the process to execute is chosen randomly from the list of enabled processes. A pseudo random number generator with uniform distribution is assumed.

However, when this directive is not explicitly specified in a program, the default execution mode is used. In the default mode, at each execution step the process to execute is chosen based on two criteria:

- how long the process has been willing to execute (first-in, first-out), and
- in case of conflict, the sequential order of their occurrence in the program text is used.

Note that the default mode is deterministic, while the random mode is non-deterministic.
Display: The directive **display** dictates how the execution trace is displayed on the screen as follows:

- **display code**
  Display the program code (except comments) after each reduction.

- **display congruence**
  Display, in addition to reduction steps (denoted by --&gt;), the congruence steps (denoted by &lt;--&gt;).

- By default code and congruence are not displayed in output, only reduction steps are.

4 Execution of ccaPL programs

The ccaPL interpreter is a java jar file named ccaPL.jar. It runs on any operating system that supports the Java Virtual Machine (JVM). The command line to execute a ccaPL program is:

```java
java -jar ccaPL.jar myprog.cca
```

where `myprog.cca` is a text file containing your ccaPL program.

To redirect output into a text file say `myoutput.txt`, use the command line:

```java
java -jar ccaPL.jar myprog.cca > myoutput.txt
```

**Example 4.1** Let's consider the program in Example 3.3, where there is no declaration statement nor execution directive. So the default execution mode and the default display settings apply. The execution output is the following:

CCA Parser Version 3.0 : Reading from file prg3.cca . . .
CCA Parser Version 3.0 : CCA program parsed successfully.

Execution mode: interleaving

---&gt; {Local: root ===(hello)==&gt; root}
---&gt; {Local: root ===(thanks)==&gt; root}

**Example 4.2** Let's consider the program in Example 3.2. It contains declaration statements and execution directives. So the execution mode is random and the code will be output after each reduction. The execution trace is the following:

CCA Parser Version 3.0 : Reading from file prg2.cca . . .
CCA Parser Version 3.0 : CCA program parsed successfully.

Execution mode: random

```
BEGIN_DECLS
  def has(n) = {this | n[ true ] | true}
  def isHello(n) = {n=hello}
END_DECLS

send(hello).recv(ack).0
| recv(x).&lt; isHello(x) &gt; send(thanks).0

---&gt; {Local: root ===(hello)==&gt; root}
```

```
BEGIN_DECLS
  def has(n) = {this | n[ true ] | true}
  def isHello(n) = {n=hello}
END_DECLS
```
5 Understanding the execution output

A sample of a program execution output is annotated in Figure 1. In general, the following notations are used:

- The symbol ‘<-->’ corresponds to the structural congruence relation as formally defined in [1]
- ‘-->’ represents the reduction relation (see [1])
- The explanation of each transition is given between a pair of curly brackets.
- The notation ‘A ==X==> B’ means that an ambient ‘A’ has sent a message ‘X’ to another ambient ‘B’.
- Additional annotations such as ‘Child to parent’ and ‘Sibling to sibling’ provide information about the relationship between the sender ‘A’ and the receiver ‘B’.
- The notation \{binding: n -> IN1\} corresponds to the execution of a process of the form “find n:k for ...” and means that the variable n has been bound to the name IN1.

6 Error messages

Programs are first parsed for syntax errors and then executed if no error is detected. When an error is detected, the parser displays an error message then terminates. Consider the following program. There is a syntax error in line 8,
namely “y/....”.

```plaintext
BEGIN_DECLS 1.
def has(n) = { this | n[true] | true} 2.
def isHello(n) = {n = hello} 3.

mode random // default mode is 'interleaving'. 5.
display code // add this line to display code 6.
  // after each reduction. 7.
y//display congruence // add this line to include congruence 8.
  // in the output. 9.
END_DECLS 10.

send(hello).recv(ack).0 11.
| recv(x).<isHello(x)>send(thanks). 12.
```

The interpreter outputs the following error message:

CCA Parser Version 3.0 : Reading from file prg2.cca . . .
CCA Parser Version 3.0 : Encountered errors during parse.
ParseException: Encountered " <NAME> "y " at line 8, column 3.
Was expecting one of:
  "def" ...
  "END_DECLS" ...
  "display" ...
  "mode" ...

Basically, the interpreter identifies the line and column numbers of the error in the program text and suggests a list of expected tokens.

**References**