CellSpeak

Language Manual

May, 2017

Cells run in parallel,
On all cores,
On interconnected systems,
As one program

Table of Contents

1 Introduction

Cells run in parallel,
On all cores,
On interconnected systems,
As one program

1.1 CellSpeak – in one page

CellSpeak is a message based general-purpose programming language. It is designed for programming the present-day computing environment that consists of interconnected multi core computing devices, great and small – cloud servers, workstations, smartphones or tiny battery operated IoT devices.

CellSpeak embraces the microservices-approach to software development. It allows to build applications from interconnected parts that have a clear interface, but no other coupling, resulting in a much greater flexibility in building or maintaining software systems – parts can be replaced – or re-used – without affecting any other part of the application, without the need for re-compiling or re-linking.

The basic design unit of a CellSpeak program is the cell. Cells communicate with each other by sending messages and executing tasks in response to these messages. Cells can be big or small and an application can consist of a few big cells or of thousands of tiny cells.

Cells execute in parallel – by design – and make efficient use of current multi-core processors. Any cell that has work to do will be scheduled on an available thread, without the need for any particular arrangements in the code itself.

Cells of an application can also execute on other machines in a network in a transparent way, i.e. without having to change the architecture of the application. All this makes that programs written in CellSpeak scale very well. A cell can also provide its services to one or to many applications.

Programs written in CellSpeak can easily communicate with other software systems – not necessarily written in CellSpeak. They can 'talk' to the browser, for example to use it as a user interface, and they can use or implement a REST interface in a very straightforward manner, because CellSpeak is message-based.

CellSpeak is also designed to integrate with external libraries written in other languages. CellSpeak is not an island and it was considered essential to be able to integrate in a well-defined way with external libraries written in C/C++.

CellSpeak is geared for performance. The exchange of messages between cells is very efficient and uses several strategies based on the size of the message and the location of the cells. CellSpeak can be used for a high-performance graphics application where elements in a scene may exchange many short messages as part of their interaction or equally well for a distributed application where remote systems exchange large chunks of data over a network.

CellSpeak is different language from many current-day languages, but it is not a difficult language.

CellSpeak is also a modern language and its syntax and constructs implement many of the current day paradigms. CellSpeak is Open Source and available under a BSD license.

1.2 Why another language

The current landscape of computer languages is diverse. The list of programming languages as can bewilderingly long. There are 'old' languages, like C, that have never really gone out of style, there are relatively young languages, like Go or Swift, there are languages aimed at specific application domains, there are scripting languages, the languages of the web, functional programming languages etc. etc.

This, combined with an observation that is frequently made, namely that a good programmer armed with a good development environment, will be able to write a good program in any decent language and it becomes clear that the question – why another language – needs an answer.

Nevertheless, it is a good thing that we still see so much activity in the design of new languages and the evolution of existing ones. Software is eating the world, but software engineering is still challenged by much of the same problems it has been trying to cope with since its beginnings, and, admittedly among many other factors, the choice of a language is still an important element of a strategy in trying build efficient, reliable, maintainable systems.

CellSpeak is born out of a few observations. First observation: over the last twenty years, the computing environment has changed considerably – multicore processors, internet, smart devices everywhere – but still the majority of the languages we use, are based on a single flow of control on a single cpu. We still manage to build big and complicated systems in this way, but we have to rely increasingly on help from outside the language.

For multi-threading and communications for example, we turn to the OS and libraries, and we include all sorts of mechanisms to make sure that we master the added problems of shared data or having to cope with asynchronous events in a program. Enter locks, semaphores, memory barriers, call-backs and so forth. Surely not everything can be part of the core definition of a language, but I am sure we all have had the experience where we felt let down by our programming language.

Second observation: many applications are monolithic and have links and dependencies in unexpected and often unnecessary places – difficult to maintain, difficult to extend. Every developer has experienced the fact that by adding or changing something, a seemingly unrelated part of the system fell over – or the system as a whole crashed.

Having experienced first-hand the difficulties in working with hundreds of programmers, testers and managers, often in different time zones, against stringent time, quality and cost targets, it became clear that more was needed and that also – apart from development environments, practices, standards, training etc.– the programming language must support what you are trying to achieve.

1.3 Design principles

The observations mentioned above - and some other – resulted in the following list of high-level requirements:

- It must be possible to write a program for a distributed architecture, without having to handle the details of scheduling and communicating between components.
- Applications must be able to scale quickly.
- It must be possible to keep dependency between separate components of a system low.
- It must be straightforward to exchange information with external systems via standard protocols.
- Reuse of existing code written in other language(s) must be possible

- A single code base for all processor types – ‘write once, run everywhere’
- No compromise on performance.

It appears that these requirements can be met by using a message-based language. However, the path that has been followed often, i.e. by adding messaging capabilities to an existing language, does away with almost all the benefits that messaging can bring because the type dependencies, data visibility etc. remain in the ‘augmented’ language exactly as in the original.

For that reason, the approach was turned upside down: a new language, CellSpeak, was developed, into which it is possible to integrate code written in another language, but that brings the separation between the different components of a program that reduces dependencies and

1.4 Content of this manual

This manual is intended to explain the CellSpeak language. It helps if the reader has experience in one or more high level computer language. The manual is intended to be used as a step by step guide to the concepts in CellSpeak but also as a reference manual to turn to when writing software in CellSpeak.

This manual does not contain a detailed discussion of how the concepts behind CellSpeak are implemented. For the interested reader, there is another manual, the *CellSpeak Design Manual*, where he can find detailed discussions of the different parts that make up CellSpeak.

However it often helps to understand certain language concepts, if some background is given about how these concepts are implemented. Wherever it makes sense to do so, that background information is included in this manual also.

Before diving into the language aspects of CellSpeak a short overview is given in Chapter two of the main components and terminology of CellSpeak.

2 The CellSpeak compiler and Virtual Machine

2.1 Terminology

The fundamental unit in a CellSpeak program is the cell. Cells are isolated objects, i.e. collections of data and methods, that communicate to other cells only by exchanging messages. All data in the cell is accessible only by the cell itself and another cell cannot invoke a method on another cell.

Once created, cells typically wait for messages and execute ‘handlers’ in response to messages. In the process of handling a message they will also typically send messages to other cells. A running CellSpeak program is thus a group of cells that react to messages being sent by other cells and that possibly send messages to other cells while doing that.

The notion of an application is somewhat blurred in CellSpeak, in the sense that new cells can be added to a group of cells to implement some new features, or that a single cell can provide its services to many cells, that can belong to different ‘applications’.

Building an application with CellSpeak is more like designing an organization - of cooperating cells - than like writing a traditional application.

There are no explicit or explicit constraints with respect to cells. Cells can be big and complicated or can be small and lightweight – and an application can consist of a few cells or of thousands of cells – all determined by the architecture of the application.

Cells are instances of what is called a *design* in the language. The name *design* was chosen over the name *class* because, as CellSpeak is designed to interface with other languages that also use the concept of a class, it makes discussions about the language(s) clearer. In fact, the keyword *class* is used in CellSpeak to refer to external C++ classes that can be used directly from within CellSpeak in the same way as in C++. More on this in the chapter on the integration of external libraries.

The design of a cell consists of data, a constructor, a destructor, functions and message handlers. As already mentioned the data of the cell and its functions are purely internal to the cell and serve to structure the code that defines the behavior of the cell. Nota that all the building blocks of a cell are all optional – it is possible to have a completely empty design.

The functions constructor etc., are written in CellSpeak using the types, statements etc. that the language provides and that are explained in this manual.

The second most important concept in CellSpeak is the message. A message consists of the name of the message and some optional data, the payload of the message. That payload can be anything, from a set of parameters like for a function call to a set of large files and anything in between. The size and structure of a message can vary as any other data structure used in a program.

Sending a message in CellSpeak is always asynchronous, meaning that a cell does not wait until another cell has acted upon the message, but instead the cell continues immediately after sending a message with whatever it still has to do.

When a cell is activated or ‘executed’, the cell will go through all the messages that it has received and when there are no more messages it relinquishes control and waits for the next message(s).

Because Cells have no direct links between them, it is intuitively clear that multiple cells can execute simultaneously on all cores that a processor has avialbale, and that by extending the messaging mechanism to work between systems in a network, cells belonging to the same application can also execute on separate, interconnected systems.

Source code written in CellSpeak is compiled into CellSpeak bytecode. CellSpeak has its own bytecode because it needs specific low level instructions to handle the message switching that is at the heart of the language. The CellSpeak bytecode is, as we will see, also quite extensive because it has included support for many different types, most notably vectors and matrices, that are part of the core of the language.

The reason for including these types is, on the one hand that these types are used in many applications and on the other hand that by including these types at the bytecode level, specific optimized assembly instructions can be used, if available, for each type of processor the programs will execute on.

CellSpeak source code is compiled into bytecode that is grouped in a CellSpeak module. Modules can be grouped together to form a package.

2.2 A CellSpeak Application

A CellSpeak application has no 'main' entry point, but is started by instantiating some selected *design*. Typically, that cell will then start to build its organization by creating other cells and so forth. It is perfectly possible to have several different types of cells where a program could start with, as it is equally possible to create or delete new cells in a running program as required.

Cells have a relationship between themselves. Cells created by another cell are its *children*, and by the same token, *children* have a parent and are *siblings* of each other. The relationship between cells are not cast in concrete. A parent cell can for example *detach* a child-cell and send that cell to another cell, who can attach it in his own hierarchy.

Cells can also destroy children, or create extra children while an application is running, for example because some cell is no longer needed after starting the application or because more cells are needed to cope with a surge in activity of the system.

Cells communicate by exchanging messages. If a cell receives a message it will try to find a handler for that message. If it does not find a handler it can do two things: either discard the message or pass it on to its children – both strategies can have their place in the architecture of a design.

Even when a cell handles a message, it still has the possibility to pass the message to its children. An example of where this is useful is graphics application, where a *draw* message would have to be executed by all the children of a cell.

A cell does not have to do any other maintenance of its message queue. Messages for which a handler was found and that has executed, or messages for which no destination was found, are removed automatically from the message switch.

A CellSpeak application keeps running and will continue to check for messages as long as it runs. An application can be stopped in different ways – for example by sending a request to a cell to stop and destroy itself, but also by destroying for example the top-cell itself directly. When a cell is destroyed all its children cells are destroyed automatically.

2.3 Architecture in a nutshell

The CellSpeak language has two main components: the compiler and the virtual machine.

The compiler compiles the source code to a CellSpeak module that consists of the bytecode for the functions and message handlers, and of the symbol information for that module. The symbol information is saved, by default together with the compiled bytecode.

Because of that, compiled modules can be used as inputs when compiling other modules, because they contain the symbol information – types, variables etc. – for that module. In this way a developer can reuse the types, functions etc. in that package without having to include any source code that was used in the creation of the package. Also, by using the package with included symbols he is always sure that the code that he is using and the definitions that were used to build that code are in sync.

However, modules can also be saved without this symbol information if deemed necessary.

The bytecode generated by the CellSpeak compiler is specific for CellSpeak. The bytecode that the compiler produces is machine independent and can be run on any machine for which a CellSpeak Virtual Machine exists.

The Virtual Machine itself has several important components: the bytecode interpreter, the native code compiler, the scheduler and the message switch.

The interpreter executes the byte code generated by the compiler directly. The interpreter takes an instruction, executes a piece of code for that instruction and then takes the next instruction.

The native code compiler will first compile the CellSpeak bytecode to the specific machine code and then transfer control to the machine code. In most cases this will be significantly more efficient then executing the bytecode directly. Note that compiling bytecode to machine code is much simpler then compiling source code to bytecode, and that for many bytecode instructions there exists a direct equivalent in the machine code.

Executing bytecode can be useful however for example in situations where the native code compiler is not available or for debugging code. In most production environments however, the VM will compile the bytecode to the native processor code before execution.

The message switch takes care of exchanging messages between the cells - cells that can be on the same machine or on other remote machines. The key parameter for the message switch is performance. For that reason the message switch is not monolithic, but instead applies different algorithms to be as efficient as possible under different circumstances. Small messages for example are handled differently than big messages.

It is simply not possible to give a single number with respect to the message switching performance because it depends on the size of the messages, the speed of the machine etc., but for small messages that stay on-board, message switching speeds in the order of millions of messages sent and received per second are not uncommon.

The task of the scheduler in the Virtual Machine is to make sure that the resources of the machine the program is running on, are used efficiently. The scheduler will create as many worker threads as required on a machine depending on the number of cores a processor has.

Note that a cell is not a thread and that switching from one cell to another is not a thread switch. The worker threads will look for cells that have work to do and when finished with one cell, they just take the next. All worker threads operate in parallel.

It is perfectly possible to run unrelated sets of cells, for example belonging to two separate applications, on the same virtual machine, while it is equally possible to run several Virtual Machines on the same system, OS permitting, while the cells in the different VM's can still exchange messages between themselves.

2.4 The CellSpeak compiler

The CellSpeak compiler uses three types of inputs: source code, packages with symbols and packages without symbols.

Source code files are, unsurprisingly, text files that contain the source code for the designs of the cells. CellSpeak does not use include files, but it is possible to have for example a source file that only contains type definitions that are shared by several projects. If the only purpose is to re-use some type definitions, it is often better to use a compiled package that contains the symbols.

Compiled code is stored in packages, together with the symbols for that package. However using a package and its symbols creates a dependency. When a new version of the package is made and the developer wants to upgrade his software to make use of that new version, the developer will typically have to recompile his code against the new package. When a package is loaded by the Virtual Machine, it will check that the versions of the packages, that have this kind of a 'hard' dependency, do match.

In many cases, you will only want to use the cells and interfaces that are defined in the package, and none of the internals - types functions and so forth - that make up these cells. This is called a soft dependency. If the package has an upgrade, because of some corrections or added features, the virtual machine will allow to mix different versions of packages that have this 'soft' dependency. The compiler will check your package against the packages with which it has a 'soft' dependency and warn about changed interfaces, but this will not invalidate the build. The worst that can happen when an interface is broken, is that a message does not get handled.

When building an application it is always better to only have soft dependencies between packages, because of the added flexibility. However for certain types of packages that implement functions like math functions or string functions and so forth, that you would find in what we typically understand by a library, it is often unavoidable to have a hard link. On the positive side, these libraries tend to be stable and do not change often.

It is also for this reason – to avoid unwanted dependencies – that messages parameters are identified by their signature and not by their types. A signature completely defines the structure of a parameter, but avoids reference to user defined types that would automatically create a hard dependency.

2.5 The CellSpeak Virtual Machine

The CellSpeak Virtual Machine executes CellSpeak programs. To start a program the VM is requested to create a cell. Usually, that cell will then start by creating other cells and eventually the whole organization of the application will be created and the cells will do their work and exchange messages with other cells in the process.

When this organization of cells is working, it is perfectly possible to add or remove cells from the running system, for example to add or disable some features of the overall system – or to have two distinct applications executing in the same VM.

The Virtual Machine consists of a group of worker threads that will execute cell-code as soon as a cell has work to do. Note that each cell does not have its own thread, and a switch from one cell to another is a much less expensive operation than a thread-switch.

Another key-component of the Virtual Machine is the message switch. The message switch delivers the messages to the cells and handles the storing and releasing of messages in the process. The message switch also takes care of the routing and sending of messages to interconnected systems.

Also part of the Virtual Machine is the Native Code Compiler. CellSpeak programs are stored in bytecode format and are translated to native machine code when they are loaded to be excecuted.

Finally, the Virtual Machine also provides a number of services to the cells. For example, the Virtual Machine has a connection manager that can be used for establishing connections to other systems. Another example of a service is the timer service that can be used to send messages to cells when a set timer expires for example. The Virtual Machine also has a mechanism to expand the services it provides, so that cells can expose additional services for other cells to discover and use.

In order to access the services of the Virtual Machine, CellSpeak has the reserved keyword *system* which is the cell that other cells can send messages to, to communicate with the Virtual Machine.

2.6 Standard Packages

Apart from the CellSpeak compiler and the Virtual Machine, there is a third component which is essential to build applications: other packages.

CellSpeak comes standard with packages for math functions, string functions, packages to create output windows, graphics packages and so forth. Packages provide functionality to the developer without him having to redevelop these services for each application.

In CellSpeak a package can also be a thin layer around an (existing) library written in C or C++ for example, giving the language immediately access to a vast body of existing code.

3 Overview of the CellSpeak Language

3.1 Introduction

In this chapter we will give an overview of the CellSpeak language. If the reader has already been exposed to other computer languages, particularly object oriented languages, he will have little trouble to follow this chapter. CellSpeak examples are presented in a different type as follows:

```
-.
A very short introduction

CellSpeak is a messaging programming language.
The units that exchange messages are called 'cells'.
Cells have no shared data.
Cells are based on a design.
A cell can run locally or remotely.
.-
```

While the purpose of this manual is not to explain the inner-workings of the compiler or the virtual machine, it is often helpful to have some background information with respect to language features. Where appropriate this is information is given in a *background section* like this:

Background

Information in the background section is there to give more insight into some features of the language. It is not required to read the section to understand the language and the section can be skipped altogether.

3.2 Some syntax choices

Broadly speaking we can say that the CellSpeak language consists of three architectural layers: the first layer is the cell and message passing layer. It is at the heart of the definition of the language and it determines the overall structure of an application in CellSpeak.

The second layer is the choice of language concepts for the language that implements the message handlers, the functions and so forth. It encompasses the type of statements that are supported, the type system and so forth, whose description make up the bulk of this manual. CellSpeak uses an object-oriented approach and incorporates many modern-day language features. Here we have deal with records and methods, closures, functions, exceptions and so forth – all of which are discussed later on.

The third layer of the language is the *syntactic sugar* that is used for the language. These are characteristics of the language that could have been chosen differently without altering the working or intrinsic structure of the language: keywords, punctuation and other structural elements in the language.

For CellSpeak the decision was made to steer away from the classic curly bracket style of coding. Not because this is a bad or inconvenient, but because it makes it instantly clear that you are looking at CellSpeak code, as opposed to for example C++ code, without having to look for specific clues or elements of the respective languages.

In CellSpeak we do not use curly brackets, instead - where required - we use '*do .. end*' pairs or '*is .. end*' pairs. CellSpeak also does not use semicolons to terminate or separate definitions, declarations expressions or statements. The language definition is such that there is no ambiguity about where something starts and another thing ends.

The single most important operation in CellSpeak is to send a message. For this a left pointing arrow is used as follows:

```
cell <- message
```

There are several ways to include comments in CellSpeak source. Line comments start either with a double slash: // or with a double hyphen: -- , whereas block comments are delimited by /* and */ or -. and .-

```
-. -----------------------------------------------
This is an example of a block comment in CellSpeak
This type of comment can be used to include text
Comment into the source code.
-.-----------------------------------------------
```

Comments between /* and */ can be nested, making it easier to comment out blocks of code without having to worry about other code commented out inside. It is up to the developer to use the comment styles as he sees appropriate, but one style (// and /* */) is better suited to comment out code during the development process, whereas the other style (-- and -. .-) is more suited to include textual comments in the source code.

Indentation is not part of the language specification and it is up to the developer to decide about this. Names of variables functions etc. start with an alphabetical character or an underscore and can contain characters, digits and underscores.

3.3 Cells

The fundamental unit of every CellSpeak program is the cell. A cell is defined by its design and instantiated with the *create* keyword as in the following example:

```
design X is
     code and data for the design
end

design Y is
     cell MyCell
     MyCell = create X
end
```

In the example above *MyCell* stores a reference to the cell created from the design X. The reference is used later to send messages to.

To be clear: a cell is a cell. In contrast to classes for example, a cell has no type other than cell. A reference to a cell can take references to any cell.

There is no 'main' cell in a program. A program is started by creating a particular cell. That cell will then typically create the other cells that are required in the program. In this way several alternative entry points for a program can be created.

A design can optionally have parameters, a constructor and a destructor. Designs can also be inherited by other designs. In that case they inherit the data and methods from the ancestor design. The methods of the inherited design can be modified and other data and methods can be added to the new design.

An important feature of cells is that they form a hierarchy. A cell that is instantiated by another cell is a child of the former. We will see that this feature is important with respect to the message flow.

The following code creates the cells as given in figure 1:

```
design Car is
     constructor is
          create Body
          create Engine
          create Wheels
     end
end
design Body is
     create Doors
     create Windows
     create Seats
end
design Wheels is
     create Wheel
       …
end
```

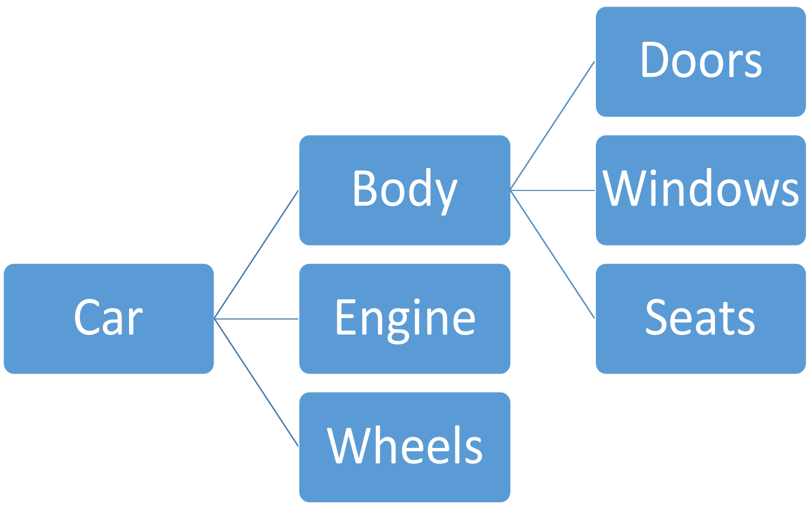


Figure 1 Cell hierarchy

Cells are organized in a hierarchy which in this example can simply be read as: a Car consists of a *Body*, an *Engine* and *Wheels*, the *Body* consists of the *Doors*, the *Windows* and the *Seats*.

The *Car* cell is the parent of the cells *Body, Engine* and *Wheels.* The cells *Body*, *Wheels* and *Engine* are siblings.

Cells are attached to their parent cell irrespective of the fact that the parent cell keeps a reference to that cell or not. In the example above the cell *Car* does not keep a separate reference for the cells of the design *Body, Engine* and *Wheels*, but the cells are attached to the cell *Car* anyhow.

But often it is however useful to have an explicit reference to the cells that are part of a design. Again, all references to cells are of the same type 'cell'.

3.4 Messages

Messages are being sent to cells as follows

```
design Car is
     cell MyEngine = new Engine
     MyEngine <- ShiftToGear(4)
end
```

In this example the message *ShiftToGear* is being sent to the cell *MyEngine* with just one parameter, an integer.

The name of a message is basically a string value that is used at the receiving end, together with the message signature, to pick the right message handler. The message signature is the list of types for the parameter(s), which in our example would simply be *int*. Because the message is handled as a string value it is actually possible to use strings as message names, for example because you want to include white space or special characters:

```
design Car is
     cell MyEngine = new Engine
     MyEngine <- "Is your oil below this level?"(75)
end
```

Names of messages will never collide with names of variables or other identifiers in the language. Later on we will see that it is also possible to use variables of the type string to send a message.

It is also possible to send multiple messages in one line and even to send the messages to multiple cells at the same time:

```
design Garage is
     cell HerCar = new Car
     cell MyCar = new Car
     cell JoesBike = new Bicycle
     HerCar, MyCar, JoesBike <- Fill(100), Clean(), Drive(1.6)
end
```

This sends three messages to each of the three cells *HerCar, MyCar* and *JoesBike*. This allows for a compact, but still readable notation. The above line replaces nine lines if only one message could have been sent to one cell at the time.

Messages are always sent in a non-blocking way. The cell does not suspend its activities until the message is delivered, or acted upon or whatever. Instead, it delivers the message to the message switch and immediately continues. If there is nothing that the cell can do after sending the messages, it should just stop and yield control.

Actions that are executed when a message is received are called *handlers*. In the design of the cell the message handlers define what to do when a message is received. In the example below *StartTheEngine* and *Drive* are two handlers:

```
design Car is

     ..code and data..

     on StartTheEngine do
          statements
     end

     on Drive(float kms) do
          statements
     end

     etc …

end
```

Message handlers start with the keyword *on*, followed by the name of the messages and the optional parameter list. A message handler is different from another handler if either the message name or the parameter signature is different.

A parameter signature is a string that contains one or more bytes for each parameter of the handler. If the handler has a parameter of type integer, then the parameter signature will contain and 'i', for a parameter of type float it will contain an 'f', for a parameter of type 4x4 matrix of double it will contain a 'J' and so forth. If the parameters contain a record, the parameter signature will contain 'R .. #' with a character between *R* and # for each field in the record. In this way each parameter combination has a unique signature that is used in selecting the handler for a message.

Note that the developer never has to deal with the signature of a message directly – that remains purely internal to the compiler.

A note on implementation

Message names and signatures are strings and every compiled design holds a table somewhere to select a message handler based on the message and signature string. For performance reasons the string values are converted by the compiler to a single hash-value, and it is that value that is used to select in the message table, making the process very fast irrespective of the length of the name of a message or of the length of its parameter signature.

This combination of message name and signature is more than enough to create all unique messages that a cell must react upon, but as a signature does not contain the actual type name that is used for a parameter, it is possible that a conflict arises, for example:

```
design Car is

     -- We define two new types based on the type float
     type miles is float
     type km is float
     -- We try to define two new handlers
     on Drive(miles Distance) do
          ..
     end

     on Drive(km Distance) do
          ..
     end

end
```

In the above case the compiler will signal the second handler as a duplicate handler because it does not make a difference between the *miles* and *km* type in a parameter signature. In practice, these cases are rare however and are easily solved by using a different name for one of the handlers.

Message handlers have no return values. If they must return some result to the sender of the message they have to send a message as well. Because this is a common situation, a cell has access to the sender of the message by using the keyword *sender*:

```
sender <- EngineStarted( 3000 )
```

When a message handler has been found for a particular message, then the message is taken from the list of messages to handle for that cell. Messages however that are not handled by a cell automatically flow through the cell's hierarchy until a handler for the message has been found or until there are no more children for that cell. Example:

```
design Garage is
     cell HerCar = new Car
     cell MyCar = new Car
     cell JoesBike = new Bicycle
     HerCar, MyCar, JoesBike <- Fill(100), Honk(), Drive(1.6)
end

design Car is
     cell Body
     Body = new ShootingBreak
     on StartTheEngine do
          statements
     end
     on Drive(float kms) do
          statements
     end

     etc …
end

cell ShootingBreak is
     on Honk do
          …
     end
end
```

The message *Honk* is not handled by the design *Car* so it will flow automatically to all its children where, in this case, the message will be acted upon by the cell *Body* which does have a message handler for *Honk*.

If a cell handles a message but also wants its children to handle the message, then the cell can use the directive *flow* in the message handler. In that case the message will also 'flow' to all the children cells.

```
on TimesUp(time t) do
     if t > 10.15 then
          flow
     end
end
```

In the example above the *TimesUp* message will flow to the children of the cell when *t > 10.15*.

The *flow* directive passes the message to *all* direct child cells of the cell. If the cell just wants the message to be handled by one of its child cells, it has to relay the message to that particular cell. If the message is exactly the same as the one received by the parent cell, the cell can use the keyword *idem* to refer to that message:

```
child <- idem
```

This allows for example a cell to redistribute work easily among its child cells.

If a parent cell does not want that a cell handles any other messages except these that the parent cell directs to it, then it has to create that child cell as a private cell:

```
cell child = create private design
```

Private cells can send messages to the outside world, but they only react to messages sent by their parent cell.

Summarizing:

- A message that is handled by a cell is considered dealt with and does not propagate to the children of the cell unless the cell explicitly does so by using the keyword *flow*
- A message sent to a cell that is not handled by that cell flows automatically to the children of the cell, unless the child cell has been created as a private cell, in which case the child cell will only respond to messages from its own parent cell.

3.5 Types

The CellSpeak type system is extensive. Many types are built into the language allowing the compiler to generate optimized instructions to work with these types.

For example 3D applications or 3D features in apps are ubiquitous, so CellSpeak has built-in types for vectors – with 2, 3 or 4 components - and matrices (2 x 2, 3 x 3 and 4 x 4 – components can also be floats or doubles). Also quaternions and complex numbers are built in.

There are two structured types available in the type system: the array type and the record type.

An array is what you expect it to be. A definition and declaration of an array can look like this:

```
type Alfa int[100]      -- type Alfa is an array of 100 ints
Alfa A              -- A is a variable of type Alfa
```

```
int B[]             -- B is an array of ints, no size given
float C[10,15,27]       -- C is a three dim. array of floats
```

The bounds of an array run from 0 to n-1 where n is the size of the array. Arrays can be multidimensional. The bounds of the array do not have to be given when it is declared, because arrays can be allocated.

A record type is more like a class in other languages: a record can have data fields and methods and can inherit from another record. But to avoid confusion we use the name *record* in CellSpeak.

In CellSpeak the name *design* is used to refer to the code and data of a cell, the name *record* for a collection of data and methods and the name *class* is used to refer to classes written in other languages – e.g. C++ - that can be imported in CellSpeak.

The difference between a record and a cell is that a record is always part of the data in a cell and offers a convenient way to group some data and methods together as classes do in other languages. A cell on the other hand however is never 'part' of another cell and can only communicate with other cells through messages. A cell can hold a reference to another cell, but it cannot invoke any methods on that cell.

The following example creates a record type *Cow* derived from the record type *Animal.*

```
type Animal is record
     float  Weight
     int   OffSpring
     double     Speed
end

type Cow is Animal with
     rename     OffSpring to Calves     -- fields can be renamed
     float  MilkProduced
     function Fodder out float is
          …
     end
end
```

Records can have data and methods, but actually all types can have methods defined for them. It is possible to define for example an array and methods for that type:

```
type Scores is int[] with

     function Mean() is
          …
     End
     function Max is
          …
     end

end

-- We create a var of the type Scores with 200 elements
Scores[200] MyScore

-- We can call the method for this var, e.g.
Int m = MyScore.Max()
```

There is no character type built-in into CellSpeak. Characters depend on encoding and can have varying length. CellSpeak does however have the type *byte* and this type is used as the basis for a standard library that defines the different types of strings (ascii, ansi, utf-8 ..) and their encodings.

A constant between quotation marks is handled as a zero terminated UTF-8 encoded string.

3.6 Variables

Variable are declared by specifying a type followed by the name of the variable:

```
int a, b, c
float x
```

Variables can be initialised in a declaration:

```
xyz Axis = [1, 45.2, 9]
```

In the example above, the components of the vector will be cast to floats as the vector type xyz consists of three floats.

When a variable is initialised, the explicit type can be omitted and replaced by a simple *var* if the compiler can work out the type from the initialisation.

```
var Q = 103.5       -- Q is a float
var H = sin(45)     -- H is of the type that the function returns
```

Constants are defined in the same way, but here the keyword *const* is used:

```
const pi = 3.1415926
```

Sometimes it is more useful to make clear in a piece of code that a given variable is of the same type as some other variable or parameter. This can be done as follows:

```
var x like distance
```

In which case *x* is declared as a variable of the same type as the variable *distance*.

3.7 Pointers

Pointers are both a blessing and a nuisance.

A blessing because they let you manipulate objects without excessive copying and because they allow to build interesting data structures like trees, lists etc. They are also a nuisance because they need attention and are often a cause of bugs in programs (null references, stale references etc.).

In CellSpeak pointers are limited to pointers to records. There are no pointers to floats, integers etc.

Inside the compiler of course, and in the code generated by the compiler, pointers are used extensively, however in a way that is transparent to the developer. If for example an integer parameter is an *out* parameter to a function, what is actually passed to the function is a pointer to the integer, but the notation in the function to work with that parameter is no different from handling an integer that is not a pointer – more on this in the chapter on functions.

Whether a variable is declared as of type record or as of type pointer to the record, does not make a difference in how the variable is dereferenced:

```
type Identity is record
```

```
     utf8   Name
     int    Age
     float  Height
end

-- We define two variables
Identity Me
Identity.ptr He
```

Note that being a pointer or not is a property of the variable, not of the type. It is possible to define a variable or a field of a record as a pointer, but it is not possible to define a pointer type in CellSpeak.

In both cases, fields of the record are referenced like this

```
Me.Height = 1.83
alloc He
He.Height = 1.78
```

Of course in the second case the record must first be allocated before it can be dereferenced. The only notational difference in working with pointers, is for the pointer assignment. Suppose that *R* and *Q* are two pointers to the same record type then we can have

```
R = Q     -- copies the content
R := Q    -- copies the pointer
```

In the first case, the content of what Q points to is copied to R, in the second case the pointer in Q is copied to R. Both now point to the same content.

As another example consider the definition of a doubly linked list in CellSpeak :

```
     type Request is record
          string       Name
          string       Song
          Request.link Next
          Request.link Previous
     end
```

You can then access the fields of these records as follows

```
Request Radio
Radio.Song = “Mountain high, river deep”
Radio.Next.Song = “Yellow Submarine”
```

3.8 Statements

In CellSpeak you will find the same set of statements that you find in other languages, like for example the if-then-else statement.

```
if condition  then
          action
else
     alternative action
end
```

CellSpeak can also use the concise form of the if statement as in this example.

```
Prime > 100 ? return : factor( Prime )
```

This statement does the same as an if-then-else statement, but at many occasions when writing a program, you just want to do a check and a single action, and in these case the readability of the code is enhanced if you can do this in single line instead of having to spread this over three or five lines.

Several flavors of loop statements are available. Take for example the *for* statement, the basic structure is always similar but the loop range(s) can be expressed in several ways. A basic loop statement would be:

```
for i=1 to 100 do
       --i takes values from 1 to 100 (included)
end
```

It is also possible to collapse several nested loops into one

```
for i = 0 to 10, j = 0 to 20, k = 0 to 30 do
     …
end
```

This loop replaces three nested loops:

```
for i = 0 to 10 do
     for j = 0 to 20 do
          for k = 0 to 30 do
           …
          end
     end
end
```

As we will see, in CellSpeak, arrays keep track of their size, so it is possible to have a loop statement in CellSpeak like this

```
for each A in AnArray do
       --A is a reference to every element in the array
       --i.e. if you change A you change the element in the array
End
```

If you need the index values themselves while working with an array you can use this ‘index’ form of the for-loop:

```
for each [i,j] in A do
     -- i,j will take the value of the indices for A starting: [0,0]
     -- [0,1], [0,2] etc. until [n-1,m-1] where n and m are the number
     -- of elements per row/collumn.
end
```

You can also use a for loop to cycle to a list of items

```
for each Name in [“horse”, “Cow”, “pig”] do
       --cycles through the elements in the list
End
```

And finally there is also a general type of loop:

```
for (i= j/2, i < j, i += 7) do
     …
end
```

Apart from *for-loop,* CellSpeak also has the *while-do* and the *repeat-until* loop.

CellSpeak also has a switch statement that works for integer-based types (integers, words, bytes), and for null terminated strings.

```
switch expression

     case v1:   statements

     case v2:   statements

     case v3,v4,v5: statements

     default:   statements

end
```

The compiler sorts the cases into a table, so that even with a large list of values, selecting in the table is efficient (of the order of *log n* instead of *n,* for a table of n entries). Maximum one element in the table is selected and there is no fall-through.

In CellSpeak it is possible to give a name or label to a statement or group of statements, which are called a *task.* That label can then be used then in two other statements: the *leave* statement and the *continue* statement. Even in a moderately complex control structure – for example two nested loops – it is often not clear what loop a *break* breaks out of, or what loop is reiterated by using the *continue* statement. The use of labels can make that clearer.

3.9 Functions

As we have seen before *message handlers* are pieces of code that are invoked to handle incoming messages. CellSpeak also has functions which are, like functions in other languages, a means to structure your code.

The format of a function is uniform, but we can distinguish between several types of functions in CellSpeak:

- global functions – i.e. not linked to a cell or a record. Examples of these functions are mathematical functions, e.g. *sin(x).* Global functions are defined outside of the context of a cell or a type. The only data a global function has access to are its parameters and its own internal data.
- methods for types – Methods are functions that can act on the data of the type. They can be public or private. Methods can be invoked on variables of a given type with the familiar dot-syntax: *variable.method(parameters).* Methods have access to the data in the variable – i.e. the fields of the record, the elements of the array etc.
- cell functions – These are functions that are defined inside the cell and can only be called from inside a cell. Top-level functions defined in in the cell have access to the private data of the cell.

The syntax for defining a function is the same for all functions, example:

```
function Name(int a, float b, xyz c) out float is
      local data and statements
end
```

If the function does not return anything, the out-part can be omitted:

```
function Name(parameters) is
      local data and statements
end
```

Functions can also have multiple return values:

```
function Name(parameters) out float, int, matrix is
      local data and statements
end
```

Multiple return values can help in writing more readable code by allowing to make a clean split between what goes in and comes out of a function.

It can be that also the parameters of function need to be modified inside the function. For that reason, parameters can be passed as *out* parameters:

```
function Name(int a, out float b, xyz c) out float is
      local data and statements
end
```

Modifications to the parameter *b* in the function will be reflected in the calling code.

Functions can be declared without a body simply by adding *to do:*

```
function Name(int a, out float b, xyz c) out float to do
```

and the body of the function can then follow later. This can be useful for functions that call each other.

In CellSpeak it is possible to re-use the declaration of a function as a function type:

```
function MyFunction like SomeFunction
```

Where *MyFunction* will have the same parameter signature and outputs as *SomeFunction.*

It is also possible to declare functions as a variable, by simply putting the keyword *var* front:

```
var function MyFunction(parameters) out float is
      ..
end
```

This fixes the signature of the function, but as the function is a variable now, the body of that function can change.

The body of a variable function can be changed by assigning another function with the same signature (same parameters and return types) to it or by simply changing the body of the function as follows:

```
body MyFunction is
      .. data and code
end
```

There is no difference in calling a variable function or a constant function, and variable or constant functions can be passed as parameters like any other parameter.

Sometimes we just need a short function in our code. It is therefore possible in CellSpeak to define a short function as follows:

```
function distance(float x, float y) => (x^2 + y^2)^0.5
```

The function is one expression long and the return type is derived from that expression.

In CellSpeak, if a parameter is not passed as an *out* parameter, the compiler will decide what is the most efficient and appropriate way for passing a parameter. For example, records will always be passed as pointers irrespective of whether they are out parameters or not.

If the parameter has not been marked an out parameter in the definition of the function, the compiler will not allow that parameter to be changed inside the function.

3.10 Closures

It is possible for a function to take a snapshot of (part of) the variables in its the scope at the moment of its definition. CellSpeak does not use scope-chains or other techniques that keep the entire scope of a function available for the lifetime of that function. However, it offers a straightforward mechanism to capture that part of a scope that a function might want to keep for later reference.

As an example, consider a design for a cube. One of the messages that the cube responds to is *StartRotating:*

```
design Cube(float Rib) is

    on StartRotating(float AngularSpeed, xyz Axis) do

        matrix4 RotationMatrix
        RotationMatrix = MakeMatrix(Axis,AngularSpeed)

        var function SomeAction out nothing
         keep Axis, AngularSpeed, RotationMatrix
        is
         local data and statements
        end

    end -- StartRotating

    on Draw do
        …
        SomeAction()
        …
    end

end -- Cube
```

What happens here is that when the cell gets a message *StartRotating* it defines an action *SomeAction* which stores a local copy of some selected variables, visible at that moment, in the keep section of the function. When the action *SomeAction* will be executed, as the result of the message *Draw* in our example, it will actually refer to these variables and execute the rotation around the axis, with the speed that was defined by the original message.

3.11 Interfacing with external libraries

CellSpeak is a new language, so it is important to make sure that CellSpeak is able to use and interface with software written in other languages. There are several ways to do this.

First, messages can be sent to and received by software not written in CellSpeak. CellSpeak can interact for example with a browser or a web API in the same way it would interact with other cells.

CellSpeak also has mechanisms to import existing external libraries. To understand how this works consider below an example of how a windows DLL is imported in a CellSpeak program:

```
-- import the external library as a group
group d3d11 is lib "..\\system\\Direct3D11.dll"

-- type mapping from the library types to CellSpeak types
library type int, float, double
library type "Vector3<float>" is xyz
library type "Matrix4x4<float>" is matrix4
library type MaterialType is material.rgba

-- library classes - have methods but are otherwise unspecified
library class CameraClass
library class SphereClass
library class MeshClass
```

There are three parts in this example. First of all the library is imported from a file as a group and a name is given to that group. A *group* is just a way to combine related data, functions definitions etc.

The second part establishes the correspondence between types used in the library and the CellSpeak types, and read as follows:

```
library type X is Y
-- The type X in the library corresponds with the type Y in CellSpeak
```

Some are trivial like the line with *int, float*, and *double* simply stating that these types in the library correspond with the same types in CellSpeak, whereas the *Vector3* line, states that a *Vector3* template instantiated with a float type corresponds to the *xyz* CellSpeak type – presumably defined elsewhere.

The third part specifies the classes – in this case the C++ term – that can be found in the library and that you want to be able to use in the CellSpeak application. The –public- methods that exist for these classes will be imported automatically and will be available for use, they do not have to be declared again in CellSpeak.

The use of the imported class in the CellSpeak program is from that moment on almost the same as using a *record* in CellSpeak. You can for example declare a variable of the type CameraClass:

```
CameraClass Camera
```

and invoke methods of that class

```
xyz Direction = [1.0,50.0,100.0]
Camera.PointTo(Direction)
```

With the information provided, the compiler is able to find the requested method in the external library and to call that function using the appropriate calling convention for that method (or function).

3.12 And then there is also…

To finalize this chapter we should still mention some other interesting features of the language – which will be elaborated in later chapters.

3.12.1 Expressions

Expressions are what you expect, with the usual operations defined.

```
int a = 3, b = 5    -- a and b are integers
float c = 7.0, d
d = c * a   -- a is converted to float and then multiplied
d = b       -- b is converted to a float
a,b = b,a   -- a and b are swapped without using a temp var
```

3.12.2 Type casts

Type casts take the form of a type between angled brackets, example:

```
xyz a
a = <xyz>[1, 2, 3] * <xyz>[4.5, 7, 9]
```

In the expression above the expression list *[1,2,3]* will be cast to the *xyz* type, presumably three float values, before being used in the multiplication.

3.12.3 Records

Record can be initialized in several ways, for example

```
record
     int   a
     floatb
     ascii  c
end R = [1, 2.25, “hello” ]
```

But the following gives an identical result, but is probably clearer:

```
var R is record
     int   a = 1
     float  b = 2.25
     ascii  c = “hello”
end
```

In record definitions derived from another record type, CellSpeak lets you rename the fields of that record. An example:

```
type xyz is vec3f
     rename c1 to x
     rename c2 to y
     rename c3 to z
```

```
end
```

The vec3f type is one of the built-in CellSpeak types with vector-operations defined on it. This type has standard *c1, c2* and *c3* as its field names, but the rename feature lets you rename these fields to meaningful names for your application. All operations existing for the *vec3f* type, are immediately usable for the *xyz* type.

3.12.4 Templates

CellSpeak also has templates that let you define a design, a record etc. with generic types and then apply to a specific type. A simple example :

```
template TrigonometryFunctions for TypeA is
     function sin(TypeA Angle) returns TypeA is …
     function cos(TypeA Angle) returns TypeA is …
     function tan(TypeA Angle) returns TypeA is …
end
TrigonometryFunctions for float
TrigonometryFunctions for double
```

Where the functions are written once, but instantiated twice the first time for parameters and result of the type float, the second time for parameters and result of the type double.

3.12.5 CellSpeak Assembly

CellSpeak allows you also to use CellSpeak assembly. In this way you can handcraft specific routines directly in CellSpeak bytecode. Normally this only advised for very short functions. Consider the following example taken from the math library

```
type quaternion_double is vec4d

     rename c1 to a
     rename c2 to b
     rename c3 to c
     rename c4 to d

     function power(double exponent) returns quaternion_double is
          $ V4D_QPOW %sp this exponent
          $ FCT_RETURN %sp size quaternion_double
     end
     function conjugate returns quaternion_double is
          $ V4D_QCONJUG %sp this
          $ FCT_RETURN %sp size quaternion_double
     end
     function norm_squared returns double is
          $ V4D_QNORMSQ %sp this
          $ FCT_RETURN %sp size double
     end
     function inverse returns quaternion_double is
          $ V4D_QINVERS %sp this
          $ FCT_RETURN %sp size quaternion_double
     end
end
```

In this example the double precision quaternion is defined based on a built-in type *vec4d* that has four components, which are renamed. There are also some additional functions defined, typical for quaternions, based on instructions that are part of the CellSpeak bytecode. The lines starting with the dollar sign are the CellSpeak assembly lines.

The compiler will decide when to inline a function or when to include a call to the function in the code. Typically this will depend, amongst other things, on the size of the function. In the example above a call to *q.power(p),* where q is a quaternion and p some power value, will be in-lined and will result in a single instruction to be included in the code, namely the *V4D_QPOW* instruction.

3.12.6 Exception handling

CellSpeak also has exception handling. Exception handling is a way to separate the normal flow of control through the code from the exceptional flow.

CellSpeak does not use *try-catch* to handle exceptions. Instead in CellSpeak any block of code – a message handler, a function, a constructor – can have one exception table. The exception table lists the action to undertake, for each exception that can occur in the code.

Exceptions are either raised by the OS or can be raised also from inside CellSpeak by using the *raise* statement:

```
raise ThereIsNotEnoughMmeory(int used, int available)
```

The exception table the function or handler etc. lists the exceptions it wants to handle followed by the actual code of the exception handler.

```
catch
     ThereIsNotEnoughMemory(int u, int a) do
          …
     end

     CouldNotFindObject(string Name) do
          …
     end
end
```

If no handler is found for the exception at a given level then the stack will be unfolded and the exception is passed to the next level in the calling chain.

3.12.7 Groups

In CellSpeak source code is organized into *groups*, started by the *group* directive followed by the name of a group. The name of a group consists of one or more names separated by periods. Everything that follows a group is supposed to be part of that group until another group begins or until the end of the file. Some examples

```
group Math
group Math.Trigonometry
group String
```

etc. Files that want to make use of a functions, designs etc. of a group have to include a *use* line at the start of the file

```
use Math.Trigonometry
```

This has the effect of making sure that the compiler gets these definitions before compiling the file. Definitions inside that group can also be accessed without having to use the group name as a prefix, if there is no ambiguity. When there is ambiguity the full group names can be used to resolve that ambiguity.

3.12.8 Variable messages

There are situations where we would like to be able to use a message variable. We might want to instruct a timer service to send us a particular message when the timer fires. In order to cater for this the following message format is also available:

```
Client <- (wakeupmessage)(parameters)
```

The name between brackets – wakeupmessage - has to be a variable of the type null-terminated string. If that variable contains value *TooLateForWork* then that message will be sent to the *Client*. Note that the message without the brackets

```
Client <- wakeupmessage(parameters)
```

Would simply send the message *wakeupmessage* to the *Client.*

3.12.9 Interfaces

An interface is a group of messages that have a logical connection. Take as an example a cell that is responsible for setting up connections using either TCP/IP or UDP. That cell could have a group of message handlers for one case and another group of handlers for the other case. Readability will be improved if these handlers can be grouped in interfaces, for example like this:

```
     interface TCPIP
          on Connect(...) do
          on Send(...) do

     interface UDP
          on Send(…) do
```

Cells that use the interfaces must use the complete name of the message, i.e. the name of the message preceded by the name of the interface:

```
Cell <- TCPIP.Connect(…)
```

As we will see later on, interfaces can also contain data that is only accessible to handlers of that interface.

3.12.10 Strings

String values are null terminated sequences of bytes. And constant strings are a series of characters between quotation marks:

```
byte[] a = “Hello World”
```

As we will see later CellSpeak has a library to work with strings of different encoding, but it is worthwhile to note already that in order to make printing, and working with strings in general, easier and more readable, strings can contain expressions between square brackets. These expressions are evaluated and replaced by their string value in the string. Example:

```
int age = 23
float weight = 75
byte[] Fact = “At the age of [age] my weight was [weight] kg.”)
```

will result in the string *Fact* to be:

```
“At the age of 23 my weight was 75.0000 kg.”
```

4 Hello world

Let us have a look at a short ‘hello world’ program to give a first impression of the look and feel of the language. In CellSpeak this might look as follows:

```
design HelloWorld is
     cell Window
     constructor is
          Window = create MenuWindow("Test",<RectangleType>[0,0,900,750])
          Window <- println("Hello World”)
     end
end
```

What happens in the design above is as folows: in the constructor of the design for *HelloWorld*, another design, *MenuWindow* is instantiated and a message *println* is sent to that cell.

The design MenuWindow is part of a package that allows to create output windows and that makes use of the C-libraries provided by the OS to do so.

In order to run the program, you would start the VM with *HelloWorld* as a parameter. The VM will then create a cell of that design and launch its constructor. From then on the cell exists and waits for messages.

The cell instantiated from *HelloWorld* and the cell *Window* remain in existence after the message *println* has been sent. In order to stop the application, the cell instantiated from *HelloWorld* has to be *destroyed*. The cell *Window*, which is a child cell, will then also be destroyed.

5 Keywords

5.1 Introduction

From this chapter we start with a detailed description of the CellSpeak language. In this chapter we give an overview of the keywords used in CellSpeak and their meaning.

The keywords used in CellSpeak can be subdivided in several groups as follows:

- Structural keywords
- Types
- Operators
- Statements
- Directives
- Built-in identifiers

5.2 Structural keywords

Structural keywords are keywords that are used to indicate the components of the language:

KEYWORD	
design	Definition of the code for a cell. Contains data, functions and message handlers.
constructor	Invoked when the cell is created
destructor	Invoked when the cell is destroyed
group	used for grouping similar designs, functions and concepts
lib	for external, i.e. non-CellSpeak libraries
use	to import other groups
interface	to group messages in a logical ensemble
function	to organize code
operator	to define an operator between types
body	to redefine the body of a variable function
template	to define code for unspecified types

5.3 Types

The following types are predefined in CellSpeak.

SPECIAL TYPES	
cell	A reference to a cell. Can be used to send messages to. A cell can also be compared for equality/inequality.
message	A reference to a message.

class	A pointer to an external, i.e. non-CellSpeak class, typically a C++ class
record	A class-like grouping of data and methods in CellSpeak.

Scalar types

int	a 32-bit signed integer
int16	a 16-bit signed integer
int32	alternative name for int
int64	a 64-bit signed integer
word	a 32-bit unsigned integer
byte	an 8-bit unsigned integer
word16	a 16-bit unsigned integer
word32	alternative name for word
word64	a 64-bit unsigned integer
float	a 32-bit floating point number
double	a 64-bit floating point number
bool	a Boolean type, can only take the values true or false
enum	a list of named constant values

Vector types

vec2i	a vector of 2 32 bit integers, components are c1 and c2
vec3i	a vector of 3 32 bit integers, components are c1, c2 and c3
vec4i	a vector of 4 32 bit integers, components are c1, c2, c3 and c4
vec2f	a vector of 2 floats, components are c1 and c2
vec3f	a vector of 3 floats, components are c1, c2 and c3
vec4f	a vector of 4 floats, components are c1, c2, c3 and c4
vec2d	a vector of 2 doubles, components are c1 and c2
vec3d	a vector of 3 doubles, components are c1, c2 and c3
vec4d	a vector of 4 doubles, components are c1, c2, c3 and c4
mtx2f	a 2x2 matrix of floats, components c11 to c22

Matrix types

mtx3f	a 3x3 matrix of floats, components c11 to c33
mtx4f	a 4x4 matrix of floats, components c11 to c44
mtx2d	a 2x2 matrix of doubles, components c11 to c22

mtx3d	a 3x3 matrix of doubles, components c11 to c33
mtx4d	a 4x4 matrix of doubles, components c11 to c44

The vector and matrix types are part of the core types of CellSpeak because there are special instructions defined for these types in the CellSpeak bytecode. These types have standard field names, c1, c2 etc., that in a typical situation would be renamed to more appropriate names. The math library that is part of the CellSpeak distribution, defines for example the following types derived from the built-in vector types:

```
type xyz_float is vec3f with
     rename c1 to x
     rename c2 to y
     rename c3 to z
end

type xyz_double is  vec3d with
     rename c1 to x
     rename c2 to y
     rename c3 to z
end
type xyz is xyz_float   -- the short name xyz stands for xyz_float

-- definition of complex numbers. We change the component names to r and i -- for
real and imaginary respectively.

type complex is vec2f with
     rename c1 to r
     rename c2 to i
end
```

The designer can rename these types and fields as he sees fit, the important point being that in the bytecode special instructions are foreseen to handle these entities in an efficient manner.

Note that there is no special type for *array*. An array is created by using square brackets with the size of the array optionally between the brackets:

```
int A[10]  -- create an array of 10 integers
byte B[]   -- creates an array of bytes - no size given
byte[] B   -- Same thing ) array of bytes
```

5.4 Operators

Operators are used for operations between types. Many of the operators have a meaning for almost all types, scalar, vector and matrix types, some operators have only a meaning for a limited nr. of types, such as the boolean operators.

As we will see later it is also possible in CellSpeak to create new types and to redefine the operators for these types. Examples of this can be found in the math library.

Sending	
<-	Operator for sending a message: cell <- message.
<+	Operator for sending a message with a delivery notification: cell <+ message
<*	Operator for sending a priority message
<!	Operator for sending a message that requests a single action/reply
Arithmetic	
+	Addition.
-	Subtraction.
*****	Multiplication.
/	Floating point division.
%	Remainder or modulo operation.
|	Integer division, only valid for integer types.
#	Vector cross product
^	Exponentiation. Used for scalar and vector types.
Boolean	
not	not true is false, not false is true
and	a and b, true if both true
or	a or b, true if one is true
xor	a xor b, true if a and b are different
Bit-wise	
and	bitwise and
or	bitwise or
xor	bitwise xor
<<	shift left
>>	shift right
Comparison	
==	equal
is	same as ==
!=	not equal
is not	same as !=
>	greater than

<	less than
>=	greater than or equal to
<=	less than or equal to
ASSIGNMENT	
=	Assignment
:=	pointer assignment
~	Delayed Assignment
OTHER	
=>	Lambda arrow. Allows to define single expression functions.
EXTERNAL POINTER	
&	Takes the address – only used for passing parameters to C/C++ methods and functions.
*****	To indicate a pointer type – only used to refer to C/C++ types.

It is also possible to combine the operators above, where that makes sense, with the assignment operator as follows:

```
A += 7
```

With is a shortcut for *A = A + 7.*

The delayed assignment works exactly as an assignment, but in an expression or statement, the value is used before the assignment takes place instead of after. An example makes this clear:

Suppose we have an array A for which we want to return an element, and then increase the index by some value:

```
return A[i += n]
```

What happens in the case above is that the index *i* is first incremented by n, and that result is then used as the indexing value for the array. However if we use the *delayed* assignment:

```
return A[i +~ n]
```

then the value is first used to get the i-th element of the array, and only then *i* is incremented by n.

In many situations this will allow to avoid the use of a temp variable and of an extra statement.

5.5 Statements

The statement keywords used in CellSpeak are very similar to the keywords used for the same purpose in other languages:

KEYWORDS	
is end	Instead of curly brackets CellSpeak uses *is end* as delimiters where required. Where more appropriate, e.g. for message handlers, the delimiters *do end* are used.
do end	See above
if then elif else end	Keywords to construct the if statement. Nested if statements can be made more readable using the elif keyword
? :	Operators to make a concise *if then else* construct.
for each in to	Keywords to construct a for loop. For loops have several variants.
loop	Keyword for making a generalized loop - C style
switch case default	Keywords to build a select statement.
repeat until	Keywords for the repeat statement
while do end	Keywords for the while do statement
continue	Statement to go the next iteration in a loop
leave	Statement to get out of labeled statement group
raise catch	Keywords for exception handling
return	To return value(s) from a function

5.6 Directives

Directives are built-in actions or variables used by the VM or compiler.

KEYWORD	
create	Creates a new cell based on a design.
destroy	Deletes an existing cell. Only child cells can be deleted by a cell.
attach	attaches another cell as a child to a cell
detach	detaches a child cell from a cell
yield	to stop processing in a message handler
flow	to signal that a message should also be passed to the children cells
flush	Clean the message pipeline - no more messages will be handled.
new	allocates data on the heap

delete	releases data from the heap
valid	checks if a link or array element selection is valid (non-null, within bounds)
sizeof	returns the size in bytes of a type or variable
nel	returns the nr of elements in an array
lel	Returns the last element index of an array – same as nel - 1
sel	Sets the nr of elements in an array
out	to signal that the parameter can be modified
public	field/function in record accessible from outside the record
private	field/function in record not accessible from outside the record

5.7 Built-in identifiers

KEYWORD	
this	refers to the record itself - typically used in methods
self	refers to the cell itself
sender	in message handlers refers to the cell who sent the message
system	the local virtual machine
same	The message being handled
all	all children of a cell

6 Structure of a CellSpeak program

6.1 Introduction

In this chapter we look in detail at the different parts of a CellSpeak program.

A CellSpeak program is compiled using the CellSpeak compiler. The CellSpeak compiler is available as a standalone compiler, but is also available as an embedded compiler in IDE's.

The CellSpeak compiler uses as its input a simple CellSpeak project file, itself written in CellSpeak syntax, that contains the following information:

- The source files that need to be compiled
- The packages and their symbols that are needed for the compilation (packages with a hard dependency)
- The packages of which the designs are used, either to create cells or to send messages (packages with a soft dependency)

The compiler will signal any errors, missing definitions, missing externals and so forth in the process of compilation. When the compilation is successful, the output of the compiler is another package that contains one module with the bytecode and the symbol information of the compiled source files. If needed, for example for distribution purposes, several modules can be grouped into one package. If a package is meant only to be used at the level of interfacing with the cell designs, then the symbol information can be excluded from the package.

Packages can be saved in two formats: a binary format and a text format (JSON format). The text format is meant for easy exchange of bytecode files between different processor architectures.

The compiler also generates additional output, meant for debugging and understanding the code generated. For example it can output a mixed listing of source code and bytecode generated.

The compiler also does an analysis of messages sent and available message handlers allowing to spot unhandled messages.

A CellSpeak application is run by starting the Virtual Machine and instantiating one of the designs of the compiled code. A design can however also be instantiated on an already running VM, so several, even unrelated, applications can run in the same VM.

6.2 The project file, modules and packages

The compiler always uses a project file for compilation. If you only want to compile a single file, the compiler can use a default project file - if present - in the directory of the source file. In this way single file programs can be made without having to make – probably the same - project file for each source file.

A typical project file for the compilation of a module might look like this:

```
--------------------------------------------------
--   Project file for an apllication
--------------------------------------------------
-- The group is always called Project
group Project

-- The source files for the project
```

```
const byte[] Source[] = [

     "c:\\somedirectory\somefile1.celsrc”,
     "c:\\somedirectory\somefile2.celsrc”,
     "c:\\somedirectory\somefile3.celsrc”
]

-- Packages for which also the symbols are needed
const byte[] FullPackage[] = [
     "c:\\somedirectory\somefile1.celbyc”,
     "c:\\somedirectory\somefile2.celbyc”
]

-- Packages of which only the cells and their interfaces are used
const byte[] Package[] = [
     "c:\\somedirectory\somefile3.celbyc”,
     "c:\\somedirectory\somefile4.celbyc”,
     "c:\\somedirectory\somefile5.celbyc”
]

const XLib is record
     ALib   = "c:\\somedirectory\\somelibrary1.dll"
     AnOtherLib     = " c:\\somedirectory\\somelibrary2.dll""
end
```

The project file always starts with the group Project.

There are three lists of file names in this project file. *Source* is the list of source file names that need to be compiled. The order of the files in the list is not important, as the compiler will adjust the compilation order as required by possible dependencies. In CellSpeak source files have the extension *celsrc* and packages have the extension *celbyc* if they were saved in a binary format or *celjsn* if they were saved in a text (json) format.

The second list of file names is called *FullPackage* and consists of the list of bytecode files for which also the symbol table has to be loaded, that contains all the information about types, function etc used in the modules in the package.

The third list of files is called *Package* and consists of the list of bytecode files for which only the designs and the interface of the designs will be used. Symbol information for these files is not loaded in the compilation process.

The fourth structure in this file is a record called *Xlib* with as many fields as there will be external libraries used in the source code. In this way all file-names used in the compilation can be grouped into the project file. If in some source file, an external (C++) library needs to be included, one can write then:

```
group NewGroup is lib Alib
```

instead of the equivalent

```
group NewGroup is lib "c:\\somedirectory\\somelibrary1.dll"
```

In this way if the location of some of the files in a project change, then the only place where adaptations have to be made is the project file.

When a package is compiled based on the configuration above, the resulting package does not ‘contain’ the other packages. When running the program the packages used in the compilation will also have to be loaded – if not already loaded by the VM.

Also important to note is that the compiled package does not contain references to the filenames of other compiled packages. All dependencies are stored at the level of the module, and each module has a name, a version and timestamp so that when loading a package with a hard dependency, it can be checked that the module that is loaded corresponds with the module that was used in the compilation process.

Note that for packages that are included in a project without loading the symbols, there is no version or timestamp check, meaning that such a package can be upgraded without needing to upgrade let alone re-compile other packages that use the upgraded package. If some interface in the upgraded package has changed, the worst that can happen is that a cell will not handle a particular message sent to it – but the software that uses the package will not crash. Off course it is always possible – and advisable – to compile affected packages against the modified package to detect elements in the interface that could have changed, but if the result is satisfactory, one can proceed with the installation of only the modified package for a particular system.

In short, packages are containers of modules and are a convenient way to store and distribute compiled code, but the dependency relationships are maintained at the level of the module.

6.3 A typical module

A typical module will have the following outline

```
use math, windows, string
use math.rotation

group MyNewGroup

function f(…) is
end

design d(…) is

    int a, b, c

    constructor is
        …
    end

   destructor is
        …
    end

    function g(..) is
        …
    end

    on msgA(…) do
        …
    end
```

import groups that will be used in this file

give a name to the group in this file. There can be several groups in a file

a global function (outside of a design)

private permanent data of the cell

constructor and destructor are optional, they do not have parameters

a function local to the design, has parameters but has also access to the private data of the cell

a message handler, the message content are the parameters; has also access to the private data of the cell

```
     on msgB(…) do
          …
     end
end
```

A CellSpeak *group* is simply a way to group related definitions and concepts into one entity. The name of a group consists of a name that can contain periods to make it more readable and/or to show the logical structure of several groups.

A source file can contain as many groups and designs as required. A group can also span several files, but the compiler will signal this. In general if a group would be too big to fit in one file, it is better to split it up in several groups, for example

```
group math
group math.complex
group math.matrices.square
```

After importing groups in a source file (*use group-name*), it is no longer necessary to use the fully qualified names for the definitions in that group. If for example the group *math.matrices.square* would contain an entity called *IdentyMatrix*, you can use that name directly and do not have to use the full name. In case of ambiguity, the full name can off course still be used, in this case the full name would be *math.matrices.square.IdentityMatrix.*

6.4 Instantiating a design

Designs are instantiated or, put differently, cells are created, by applying the keyword *create* to a design, for example

```
cell Earth = create PlanetDesign
```

It is not necessary to create an explicit reference to the cell, in this case *Earth*. You can simply create the same cell as follows:

```
new PlanetDesign
```

the cell will be created and will be fully functional, i.e. able to send and receive messages. The cell will in effect be attached to the cell where it has been created as a child-cell.

A second important point is that the design *PlanetDesign* does not necessarily has to be part of any of the groups listed at the start of the file in the use list. As long as the group is part of the overall application, the compiler will be able to resolve this creation request, which is normally the last step in a build process. A failure to find the requested designs will be notified.

A design can also have parameters:

```
cell Earth = create PlanetDesign( Radius, Density, OrbitalPeriod )
```

The parameters in the creation of the cell must match off course the parameters specified in the design, however an important remark applies: parameters in the design of a cell are checked based on their signature, not based on their type. Example:

```
type SizeType is float
type DistanceType is float
```

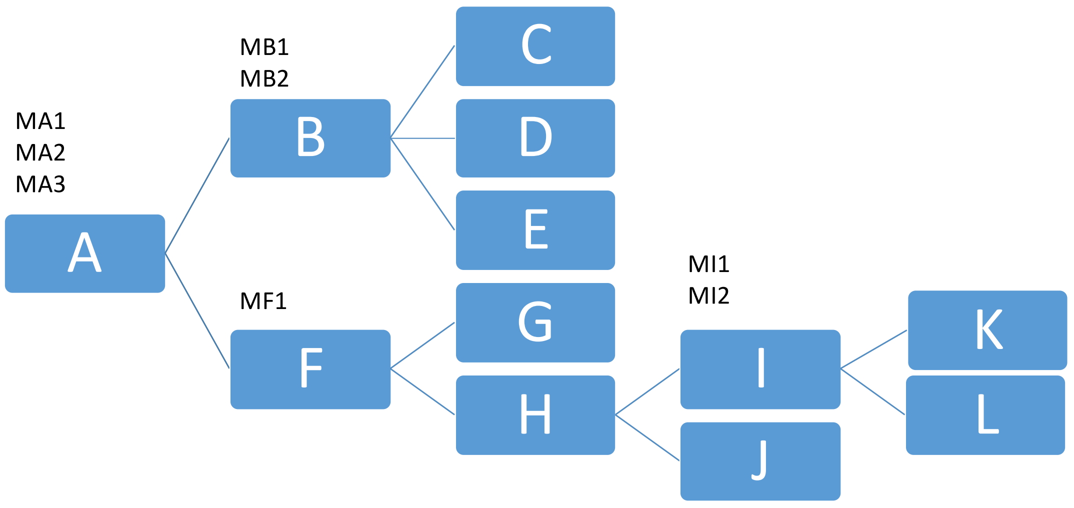
```
design PlanetDesign( SizeType Radius ) is
     …
end

design PlanetDesign( DistanceType Distance) is
     …
end
```

The second design will not be accepted by the compiler because although the types are different, the underlying type, float, that is used in the signature, is the same and the second design will be considered a re-definition of the first.

The reason for this is, that in this way a module can create a cell without having to have a strong dependency link with the package where the cell is defined.

6.5 Message flow

To illustrate the message flow in a CellSpeak program, consider the diagram below. The blue rectangles are cells, named A to L and the lists next to the cell are the messages that have been sent to that cell: the cell A has three messages waiting MA1, MA2 and MA3.



The way the Virtual Machine handles these messages is as follows. It will take the messages for cell A and see if there are any message handlers for the message in the design of A. Messages that were not handled by A or messages that were passed on to the children of A (flow directive) are taken to the next level. Assume that MA1 and MA2 are passed on, then the Virtual Machine will take the message MA1 and MA2 and add these to the list of messages to be handled for cell B. For cell B the VM will then try to find handlers for MB1, MB2, MA1 and MA2. Any messages that were not handled by B will then be passed on to for C,D and E.

In this way it is possible for a complex design consisting of many cells to act as a a single cell to the outside world. Even if the internal design changes, messages sent to that cell will flow to the cell that has the handler to act on it.

Designers should make no assumptions about the order in which messages arrive at a cell. There are however two principles that are respected when handling messages.

First, messages that were sent in one send operation are handled at the receiving end in the same order as they were sent:

```
C <- M1, M2, M3
```

In the example above, the three messages sent to cell C will be handled in that order: first M1, then M2 and finally M3.

Second, when the VM is traversing the tree of cells, the messages for a particular cell are handled in a sort of reverse order. First the messages sent directly to the cell will be handled, then the unhandled or 'flown' messages of the parent, then the message handed down via the grandparent and so forth. Using the example in the figure above, and assuming that all messages are passed on by the cells, the order of handling messages for cell I would be: MI1, MI2, MF1, MA1, MA2, MA3.

Messages can be sent using four types of operators:

```
Cell <-  Message    -- sends a message
Cell <+- Message    -- sends a message and requests a dilivery notification
Cell <!- Message    -- sends a message that will be handled by a single cell
Cell <*- Message    -- sends a priority message
```

Message operators can also be combined:

```
Cell <*+!- Message
```

The message in this send operation is a priority message, a delvery notification is asked for, and only one cell should handle the message.

We will look at the message operators in more detail below.

6.5.1 Cell <- Message

The first operator simply sends the message to the cell. The result of sending a message is the destination cell. If the cell on the left hand does not – or no longer - exist, the result of the operation will be null. This allows for testing if the system was able to send the message - note however it does not mean that the destination has a handler for the message, it might have one or not.

For cells that exist on other connected systems, the result will be ok if the avatar cell for the remote cell still exist – we will come back on this topic later.

6.5.2 Cell <+- Message

Using the second format also sends the message to the destination *Cell*, but also requests a delivery notification. The virtual machine (not the destination Cell itself) will send a delivery notification if it has been able to add the message to the message queue of the destination cell. A delivery notification takes the following form:

```
     OriginalMessageName.DN     -- if successful
     OriginalMessageName.NDN    -- if unsuccessful
```

The notifications do not have parameters. The *sender* of the message is the original destination, although the notification is not actually sent by that destination, but generated by the VM.

Note that in general it is not necessary to request for a delivery notification – the message switch is reliable, and messages do not get lost, and we have seen that if the destination cell does not exist the systems returns null anyhow. But there are circumstances, where the use of an explicit delivery notification can be useful, for example for critical messages between remote systems with an unreliable connection.

6.5.3 Cell <!- Message

If a message is sent to a group of cells, assuming each of the cells has a message handler for the message, that handler will be invoked for each of the cells. This is not always what you want. Suppose you have group of identical cells, and you just want that whatever cell is available, picks up the message and acts on it, then you would first have to identify a free cell and then send the message only to that cell.

This message operator simplifies this: it makes sure that a message is only handled by a single cell, and is a useful construct for example to distribute work over a pool of identical cells.

6.5.4 Cell <*- Message

The fourth format allows to send a priority message. Normally new messages are added to the end of the message queue for a given cell, but by sending a priority message, the message gets added at the front of the message queue for the cell. Obviously if all messages are sent as priority messages, this has no effect.

6.6 Message selection

Message handlers are selected based on the name of the message *and* the parameter signature, i.e. the parameters for that message.

As we have seen the name of a message can be a simple identifier or a string literal. The following messages are all valid:

```
Car <- ShiftToGear(4)
Car <- “Is the engine’s oil level above this ?”(90.0)
Car <- Bluetooth.Connect.Radio(“WKCC”)
```

The message signature of a message is internally represented as a string of bytes with one byte per parameter. If we assume for example that a integer is represented as ‘i’ and a float as ‘f’ then the parameter signature for the message handler below would be “iff”

```

on Drive(int gear, float speed, float distance) do …
```

Off course message signatures are internal to the CellSpeak program and are never manipulated directly by a designer, but there is an important point that needs to be made: signatures do not record user defined types as explained in the case of cell creation. Example:

```
type A is int
```

There is no difference between the type A and an int as far as the message signature is concerned. The message handlers below

```
on Honk(int NrOfTimes) do …
on Honk(A NrofTimes) do …
```

will be seen as having exactly the same name and signature and the compiler will signal this conflict.

Signatures do however take the structure of the data into account: sending two integers is different from sending a record with two integer fields or from sending an array of two integers. The signatures for these three cases would be:

```
     ii
     Rii#
     Ai
```

The size of an array is not part of a signature, and a message handler that has an array of integers as a parameter will be selected irrespective of the size of the array.

Also when you send a record in a message, the names of the fields or the methods that are part of the definition of that record are not sent along. Messages are just structured packets of data that are exchanged between cells.

The fact that the format of messages is standardized by using a predefined set of data-identifiers makes it possible to keep the links between cells independent of type definitions. A designer can choose however to make the link between cells strong by using the same type(s) in the sending and receiving cell, but this is not required.

For the sake of clarity, parameter handling for functions and methods inside a design is, as in other languages based on the user defined type specs and strict type checking.

6.7 Interfaces

Interfaces are a group of message handlers that have a logical connection. A cell can support one or more interfaces.

The message handlers for a particular interface are grouped between *interface name* and *interface end* pairs (or when a new interface starts):

```
interface Payment

     private permanent data for the interface

     on Initiate(…) do

     on Confirm(…) do

     on Abort(…) do

interface end
```

Messages sent to the interface are preceded by the interface name:

```
Cell <- Payment.Initiate( … )
```

Data in the design of a cell that is declared inside the interface is only accessible by the handlers of that interface.

6.8 Rerouting and Default Handlers

Often a cell will only handle part of the messages it receives and delegate some of its tasks to other cells, for example to its children. This can be done by rerouting a message.

A message can easily be rerouted using the keyword *same:*

```
on CalculateDistance(…) do
     RoutePlanner <- (same)
end
```

In the example above the message *CalculateDistance* is re-routed to the cell *RoutePlanner.* The message is rerouted with exactly the same parameters as the original message.

Note that the keyword *same* has to be placed between parentheses, because otherwise it would mean that a message with the name *same* and no parameters is sent to the destination cell. The parentheses force the compiler to treat what is inside them as an expression.

A message that is rerouted is not copied, so even for large messages, rerouting is efficient. The message is in most cases simply attached to another message queue.

In CellSpeak, it is also possible to specify a default handler for a given interface. In combination with re-routing this is a powerful mechanism to distribute work among cells or groups of cells:

```
on Payment.? do
     Cashier <- (same)
end
```

In the example above, any message string with *Payment.* will be rerouted to the cell *Cashier*.

It is also possible to handle part of an interface and delegate other parts to other cells:

```
interface Payment

     private permanent data for the interface

     on Initiate(…) do

     on Confirm(…) do

     on Abort(…) do

     on ? => Bank <- (same)

interface end
```

Another example: if a cell wants to delegate all messages that it does not handle itself, to its children, this can be done in the following way:

```
on ? => all <- (same)
```

Note that the question mark in the message is not pure pattern matching – it has to come at the end of an interface and stands for a complete message, name and parameters.

7 The CellSpeak type system

7.1 Introduction

The CellSpeak type system is designed to be flexible and efficient. Flexible because it allows the designer to construct the types and containers required for his application and efficient because it has many built-in types that make efficient use of processor architectures and the possibly optimized instruction set of the processor.

The number of types in CellSpeak is relatively extensive because of the inclusion of the vector and matrix types. These types were included because their use in graphics, robotics, maps etc. is ubiquitous and the compiler can often generate optimized instructions for handling these types.

Off course nowadays a lot of the processing of vectors and matrices is done by GPU's, but even then there is often considerable vector processing required on the CPU. And off course the use of GPU's in an application is completely compatible with the use of CellSpeak.

CellSpeak has vector types for ints, floats and doubles, each with 2, 3 or 4 components, and to work efficiently with vectors also matrices were added in the 2x2, 3x3 and 4x4 variety for ints, floats and doubles.

The basic constructs to build other data types are the array and the record. Together with templates, they allow full flexibility in constructing new types. Many examples are included in the CellSpeak distribution.

Records are a convenient way to group some data with operations on that data, but as already noted, in CellSpeak any data type can have methods defined of it.

General pointers are not part of the CellSpeak type system, but pointers to record types can be defined.

7.2 Scalar types

The following scalar types are defined in CellSpeak:

SCALAR TYPES	
int	a 32-bit signed integer
int16	a 16-bit signed integer
int32	alternative name for int
int64	a 64-bit signed integer
word	a 32-bit unsigned integer
byte	an 8-bit unsigned integer
word16	a 16-bit unsigned integer
word32	alternative name for word
word64	a 64-bit unsigned integer
float	a 32-bit floating point number

double	a 64-bit floating point number
bool	a Boolean type, can only take the values true or false
enum	a list of named constant values

Operations between scalar types are also defined and work as expected. The table below lists the operators in descending priority. Operators with the highest priority are executed first, operators with the same priority are executed left to right:

Operator(s)	
- not	inversion and Boolean inversion, tilde is reserved
^	exponentiation
* / % |&	multiplication, division, remainder, integer division, ampersand is reserved
+ -	addition, subtraction
<< >>	shift left, shift right
== != < > <= >=	comparison operators
and or xor	Boolean and bitwise logical operators
= ~	Assignment and delayed assignment

Where it makes sense, operations between different types are allowed, for example the multiplication of a float with an integer or the addition of a byte to an int. The type of result of such an operation is always the most precise type:

```
float a = 7.0
double b = 3.2
int x = 145

a * b       -- result is a double
x^b        -- result is a double
x + a/b     -- result is a double
```

In an assignment, if the left-hand side does not have the same type as the right hand side, the right hand side is usually automatically converted to the type of the left hand side, unless where this conversion gives a loss of precision. In that case an explicit cast is required.

```
b = a*b         -- ok, result is double
a = x       -- x is converted to a float before the assignment
x = a + b       -- not possible - explicit cast required
x = <int>(a + b)    -- ok
```

An explict cast takes the form of the type to be cast to between angled brackets.

Arithmetic operators and assignment can be combined in one operation:

```
+=  -=  *=  /=  %=  |= ^=
```

The delayed assignment is the same as a normal assignment, however the result of the assignment expression is used before the assignment is done:

```
int i = 7
A[ i +~ 1]  -- this selects element 7 and then increments by 1
A[ i += 1]      -- this increments i to 9 and then selects element 9
return v = v.next   -- this first takes the next field and returns that
return v ~ v.next   -- this will return v and then sets v to v.next
```

7.3 Numerical Constants

Numerical constants in CellSpeak can be written in different ways:

CONSTANTS	
decimal integer	7
	1_256_789_365 : Underscores can be used to improve readability
word	556w : w signals that the constant is a word – unsigned int.
hex integer	0xff00a9 or 0xFF00A9 : Starts with 0x and can contain a-f and A-F
hex byte	0xff : like a hex integer but with only two positions
hex byte as char	0x'a' = 0x61 , 0x':' = 0x3A etc.
float	3.1415
	3.1415568957e
	6.626069e–34 : exponentiel notation
hex float	0x.7f45ad84 the 0x is followed by a decimal point and 8 hex characters
double	3.1415926535897932d
	6.626069d–34 : exponentiel notation
hex double	0x.01457a4f55de23a6 the 0x is followed by a decimal point and 16 hex characters

The word notation for a constant, with the terminating w is useful for passing word constants – unsigned ints – as parameters. If a function has a word parameter, it can be called as follows:

```
f( 100w )
```

This is equivalent to

```
f( <word>100 )
```

The character notation *ox'a'* is a byte with the ascii value of the character.

7.4 Enumerators

the *enum* type allows to create a type with a series of named elements:

```
enum Colors = [black, white, green, red, blue, maroon]
```

Values of enumerators are always used in combination with their type:

    var FavoriteColor = Colors.blue

The enumerators must be unique in a given set. Enumerators are not numbers but they can be used as indices for an array and in a loop:

```
rgba Pallete[Colors]        -- an array of rgba records for all Colors

for each color in Colors do …   -- Will execute the loop for each color
```

Enumerators can also be used in a switch statement.

Internally the representation of enumerators are integers, but there are no default operations defined on enumerators, such as + * etc. The designer can however, as for other types, define these operations as he sees fit.

7.5 Vector and matrix types

The following vector and matrix types are part of CellSpeak:

VECTOR TYPES	
vec2i	a vector of 2 32 bit integers, components are c1 and c2
vec3i	a vector of 3 32 bit integers, components are c1,c2 and c3
vec4i	a vector of 4 32 bit integers, components are c1,c2,c3 and c4
vec2f	a vector of 2 floats, components are c1 and c2
vec3f	a vector of 3 floats, components are c1,c2 and c3
vec4f	a vector of 4 floats, components are c1,c2,c3 and c4
vec2d	a vector of 2 doubles , components are c1 and c2
vec3d	a vector of 3 doubles, components are c1,c2 and c3
vec4d	a vector of 4 doubles, components are c1,c2,c3 and c4
mtx2f	a 2x2 matrix of floats, components c11 to c22
mtx3f	a 3x3 matrix of floats, components c11 to c33
mtx4f	a 4x4 matrix of floats, components c11 to c44
mtx2d	a 2x2 matrix of doubles, components c11 to c22
mtx3d	a 3x3 matrix of doubles, components c11 to c33
mtx4d	a 4x4 matrix of doubles, components c11 to c44

As mentioned before, the name of these types and of their components is generic and usually the designer will use more telling and appropriate names. We will discuss how this is done in later sections, but the example below will make it already quite clear:

```
type xyz is vec3f with
     rename c1 to x
     rename c2 to y
     rename c3 to z
```

```
end
```

From then on the type xyz can be used iso the type vec3f. All operations defined for vec3f are then also inherited by xyz. Components of variables of the type xyz are also addressed as expected:

```
xyz Position
Position.x = 1.0
Position.y = 12.0
Position.z = 5.0
float Length = (Position.x^2 + Position.y^2 + Position.z^2)^0.5
```

Possible Operations between vectors and scalars are multiplication and division:

```
xyz Pos
float Scale
Pos = Scale * Pos   -- every component multiplied by scale
Pos = Pos * Scale   -- idem
Pos = Pos / Scale   -- every component divided by scale
Pos = Scale / Pos   -- Scale/x, Scale/y, Scale/z
```

The following table gives an overview of all operations. A scalar can be any of the scalar types, vector is any of the vector types and matrix any of the matrix types.

TYPE	OPERATION	TYPE	REMARKS
scalar	*** / % |**	**vector**	integer division only valid for integer types
vector	*** / % |**	**scalar**	idem
vector	**+ -**	**vector**	component wise addition / subtraction
vector	*****	**vector**	dot vector product - result is double or float
			both vectors must have the same dimension (2,3 or 4)
vector	**#**	**vector**	cross vector product - result is a vector of the same dimensions as the two vectors in the operation
matrix	*** / % |**	**scalar**	component-wise operation on each element of the matrix
scalar	*** / % |**	**matrix**	idem
matrix	*****	**vector**	vector transform - vector as a column. Vector and matrix must have compatible dimensions, e.g. only a 3 x 3 matrix can transform a vector with 3 components.
vector	*****	**matrix**	vector transform - vector as a row - result identical as above.
matrix	**+ -**	**matrix**	component-wise addition / subtraction of two matrices of equal dimension

matrix	*****	**matrix**	matrix multiplication - not commutative

The cross vector product and the dot vector product have the same priority, so suitable brackets have to be placed when applicable.

The following example is a function from the math group that returns a 3x3 matrix for the rotation around the x-axis over a given angle:

```
function x_axis(float angle) out matrix3 is

     float s = sin(angle)
     float c = cos(angle)

     return [   1,  0,  0,
            0,  c,  -s,
            0,  s,  c]
end
```

The *matrix3* type is just the mtx3f type that has been renamed.

Vector and matrix types have a number of methods defined for them, as shown in these examples for vectors with three floating point components and 3x3 floating point matrices. The functions also exist for the other matrix and vector types.

```
xyz Vector = [1.0,2.0,3.0]
float L = Vector.length()       -- return (1 + 4 + 9)^1/2 = 3.74
xyz Norm = Vector.norm()        -- returns a normalised vector
Vector.normalize()              -- normalizes the vector

matrix3 M
float Determinant = M.det()     -- determinant of a matrix
matrix Trans = M.transpose()    -- transposed matrix: i,j = j,i
matrix Inverse = M.inverse()    -- the inverse of a matrix - if it exists
```

7.6 Methods for types

Any type in CellSpeak can have methods defined on it. Example:

```
type height is float with
     function km => this / 1000
     function cm => this * 100
end

const height EifelTower = 344
println(“The EifelTower is [EifelTower.cm()] cm high”)
```

Also operators can be defined on new types, example

```
type Mod17 is int
operator + (Mod17 a, Mod17b) out Mod17 is
     return (a + b) % 17
end
```

7.7 Record type

7.7.1 Data and methods

Record types are important building blocks in CellSpeak. As already mentioned, they are Object Oriented constructs similar to classes, but to avoid confusion the *class* keyword is reserved for external - e.g. C++ - classes.

Records can have data fields and methods and can inherit from other records. Because CellSpeak with the *cell* construct is already object oriented, the records are more lightweight in that respect: a record does not have multiple inheritance and does not have a constructor or a destructor.

A record can also be sent as a message parameter, in that case only the data members of the record are sent in the message.

Example of a record:

```
type WeatherStation is record with
     xy     Coordinates     -- xy is a vector
     utf8  Name             -- utf8 is a string
     float Temperatures[]   -- array of unspecified length

     function MeanTemperature out float is -- method
          float Total
          for each T in Temperatures do
           Total += T
          end
          return Total / nel Temperatures
     end
end
```

In the example above *nel* returns the number of elements in the array.

7.7.2 Public and private

By default, functions and data in a record are *public*, unless the qualifier *private* is used. A qualifier remains in effect until a new qualifier is applied - *public* or *private. Public* in the case of CellSpeak in any case means only accessible from within the cell, as it is not possible to access data in other cells.

Sometimes it can be more clear to separate the implementation of the methods from the declaration. The following is allowed:

```
-- define the record in one file ..
type WeatherStation is record with
     xy    Coordinates      -- xy is a vector
     utf8  Name             -- utf8 is a string
     float Temperatures[]   -- array of unspecified length
     function MeanTemperature out float to do
end

-- ..and possibly in a different file, the implemenation of the methods
function WeatherStation.MeanTemperature out float is
     float Total
     for each T in Temperatures do
          Total += T
     end
     return Total / getel Temperatures
```

```
end
```

Note that in a method - or function - definition it is not required to put brackets for parameters if there are no parameters.

The size of the array in the record can but does not need to be specified - we will come back to this in the discussion about arrays.

7.7.3 Inheritance

To create a new record that inherits fields and methods from an existing record you simply write:

```
type BiggerWeatherStation is WeatherStation with
     float Precipitation[]
     function MeanPrecipitation out float is
          …
     end
end
```

where in this example, the record to inherit from is *WeatherStation*.

A record that is derived from an ancestor record can add data fields and methods. In CellSpeak it is also possible to rename inherited fields, for example to resolve ambiguities in the new definition. Renaming is done by using the keyword *rename:*

```
type BiggerWeatherStation is WeatherStation with
     rename Temparatures to GroundTemperatures[]
     float Precipitation[]
     function MeanPrecipitation out float is
          …
     end
end
```

7.7.4 Shortcuts

Sometimes you just want a record as a grouping of some other data in your program. In that case, you can take a shortcut to define the record type as follows:

```
-- keep in a record
type Subscription is record
     keep Name, Age, Magazine
end
```

In the case above a record type will be created with three fields - *Name, Age* and *Magazine* - where these three are known variables in scope of where the definition takes place. The record type will have three fields with the same names and of the same type as the variables.

Sometimes you even don't want to bother to create a type for the record. You can then simply create and initialize a record in one go as follows:

```
var Painting is record
     ansi Title = "Victory Boogie Woogie"
     ansi Artist = "Piet Mondriaan"
     int NrOfColours = 3
end
```

This also works with *keep*:

```
-- keep in a record
var Subscription is record
     keep Name, Age, Magazine
end
```

Where in this case we have no created a type, but a variable with three fields modeled after three variables in scope. This can also be used for constants:

```
const Beatles is record
     ansi Basist = "Paul McCartney"
     ansi RythmGuitar = "John Lennon"
     ansi LeadGuitar = "George Harisson"
     ansi Drums = "Ringo Starr"
     NrOfRecords = 600000000
End
```

Note that in CellSpeak you can define variables by using the *like* keyword:

```
var Artwork like Painting
```

This can relieve the designer for having to come up with type names when not really required.

7.8 Arrays

In CellSpeak arrays are defined by using square brackets as follows:

```
int A[10]  -- an array A of 10 integers
float A[]  -- an array of floats of unspecified size
```

Elements of arrays can be accessed as expected:

```
int A[10]
A[6] = 1568
```

The first index of an array is 0, so in the example above the seventh element of the array is set. The index range of an array is thus 0 to n-1, where n is the number of elements in the array.

An array type is defined as follows

```
type RainFall is float[]
```

This type can then be used to define actual arrays as follows

```
RainFall A          -- declares an array of the type RainFall
RainFall[100] B     -- idem, but fixes the size at 100
RainFall B[10]      -- very different: creates an array of ten arays of the
                    -- type RainFall
RainFall[100] B[10] -- Creates an array of ten arrays of the type RainFall
                    -- each of the size of 100
```

Arrays can have one or more dimensions:

```
xyz Surface[100, 200]
```

If a size cannot yet be given for a multidimensional array, the sizes can be omitted:

```
xyz Surface[,]
```

Note that in CellSpeak a multidimensional array is not the same as an array of arrays:

```
xyz Surface[][]
```

In this case Surface is an array of one-dimensional arrays of vectors.

In CellSpeak arrays know their size. It is therefore possible to use loops for arrays as follows

```
for each Vector in Surface do …
```

where Vector will iterate through all elements of Surface (see also the *for* statement).

7.8.1 Dynamic

Arrays in CellSpeak are always allocated dynamically – more about this later. This means that you can use variables or parameters to set the size of an array:

```
function RollDice(int NrOfTrials) out int is
     int Trials[NrOfTrials]-- NrOfTrials should be 0 or positive
     …
end
```

The array *Trials* of the required size will be created, and cleaned up afterwards, automatically. Trying to create an array with a negative size has no effect and creates an array of size 0.

7.8.2 Array Assignment

Another way to initialize an array is to assign an expression list to it. An expression list is comma-separated list of expressions between square brackets. For an array the elements in the list must evaluate to values compatible with the array.

```
int A[] = [1,85,5,8,457]
```

This initializes the array A to an array of 5 elements.

Arrays can also be assigned to one another:

```
int B[]
B = A       -- copies all the values of A to B
```

Arrays can also be assigned using the reference assignment:

```
B := A
```

In that case content of A is not copied to B, but a reference to the content of A is copied to B.

7.8.3 Slices

Arrays can also be 'sliced'

```
B[3:4] = A[1:2]     -- elements 1,2 of A are copied to 3,4 resp.
int C[] = A[3:]     -- C copies the elements 3 until the end of A
B[:2] = C[3:]       -- the first two elements copy element 3,4 from C
```

There are no negative slice indices. When assigning a slice to an array where the sizes of the slices do not match, the smallest range of the slices will be used, and the copy will still start at the beginning of the ranges.

If at a given moment an array has received a fixed size, i.e. by specifying the indices, then the size of the array will be respected, otherwise the array will take the size of the array or expression list that is assigned to it.

7.8.4 Array directives

There are three array specific identifiers:

```
getel      -- gets the nr of elements in the array
setel      -- sets the nr of elements in the array
lstel      -- returns the index of the last element in the array
```

Given the following declaration

```
int A[] = [7, 5, 45, 9, 16, 21, 877]
```

then

```
nrel A
```

will return a value of 7. Because it often happens that we have to access the last element in an array, the directive *lstel* allows to do that. So in the example above

```
lstel A
```

will return 6. Off course *lstel* is always equal to *getel – 1.*

The size of an array can be modified by using the directive *setel*:

```
setel A[20]         -- A now has 20 elements
```

When an array is resized the previous content of the array is lost.

7.8.5 Arrays of records

We have used simple arrays in our examples so far, but off course it is possible to create arrays of more complex types such as records and initialize these with an expression list:

```

enum PlanetEnum = [ Mercurius, Venus, Earth, Mars, Jupiter, Saturn, Uranus,
                Neptune, Pluto]

type PlanetRecord is record
     cell Cell
     ansi Name
     floatMass      -- In Earth Masses
     float Distance    -- From Sun, in AU = distance earth sun
end

-- The list of data for the planets
PlanetRecord PlanetList[ PlanetEnum ] = [
     ["Mercurius",0.06, 0.39],
     ["Venus",0.82,0.72],
     ["Earth",1.0,1.0],
     ["Mars",0.11, 1.52],
     ["Jupiter",317.8,5.2],
     ["Saturn",95.2,9.54],
     ["Uranus",14.6, 19.22],
     ["Neptune",17.2,30.06],
     ["Pluto",0.00218,39.54]   -- rehabilitated
]
```

7.9 Strings and string literals

String literals are null terminated utf-8 encoded strings between quotation marks :

```
“This is a string literal”
```

There is no built-in character type in CellSpeak because characters and strings can have different encodings and byte ordering. There is however a group *strings* that is part of the standard distribution that has implemented the string types for *ascii, ansi* and *utf8*

ASCII is a string of bytes where bit 7 is not used to encode a character.

ANSI is equal to ASCII for the first seven bits and depends on the system settings for the 128 characters with bit 8 set to 1 (on US and Western European default settings, “ANSI” maps to Windows code page 1252).

UTF-8 is a multi-byte character encoding of Unicode that is backward compatible with ASCII. String literals are constants, and thus immutable, but strings are arrays of bytes and can be changed.

The basis for these three types is the null or zero terminated byte array.

The base type for all three string types mentioned above is called *zt*, for zero-terminated. The declaration, together with some of the functions that operate on the zero-terminated byte array are given below.

```
group string

type zt is byte[] with

     function len out int is
          …-- returns the length of a string. len of “abc” is 3
     end
     function lenz out int is
          …-- returns the length of a string, including the terminating
           -- 0 byte. len of “abc” is 4
     end
     function find(byte b) out int is
          …-- returns the first position of byte b in a string
           -- find b in abc returns 1
     end
     function find(zt s) out int
          …-- returns the position of the substring in a string
           -- find “efg” in “abcdefghijkl” returns 4
     end
end
```

There are additional functions for appending strings, comparing strings, converting strings to upper and lower case etc.

Strings are very often used in formatting output, for example to output to the screen. CellSpeak allows for easy formatting in a string literal:

```
ascii Name = “Ozymandias”
int Age = 73

ascii Announcement = “My name is [Name] and I am [Age] years of age”
```

This will produce a string literal : “My name is Ozymandias and I am 73 years of age”. This offers an easier way of formatting output then by having to put the format specifiers in the line and the variables after the format line (C-style). If the variable has to be fomatted in a special way, e.g. to specify the exact nr of digits, the format specifier can be added after the variable seperated by a comma.

```
ascii Announcement = “My name is [Name] and I am [Age,%3d] years of age”
```

In the examples above we have just put a simple variable inside the square brackets in the string, but you can actually put any expression between the square brackets.

When square brackets are needed inside the string without substitutions, the opening square bracket has to be escaped with a back-slash: \[. The closing square bracket can also be escaped, but that is not necessary.

To get a flavor of how to work with strings in CellSpeak consider some examples below. The cell *Window* is a cell that can write text to a window on the screen. Other cells communicate with this cell by sending the message *println.*

```
-- This the cell that does the tests
design StringTest is

     -- Create the window cell
     cell Window = new OutputWindow

     constructor is
          Window <- Print("\nSome tests with strings")

          -- 1. Let's start with a simple case

          Window <- Print("\n\n***Test1: simple case\n")

          -- Print the string and its length.
          -- Note that quoatation marks inside the string are escaped

          ansi a = "Short string" --12 bytes
          Window <- Print("\nThe string \"[a]\" is [a.len()] bytes long")
          Window <- Print("\nThe string \"[a]\" is [a.lenz()] bytes long,
                         \0 included”)

          -- 2. Multiline Strings

          -- Multiline strings are also possible
          -- You can allign a multi-line string in the source code
          Window <- Print("\n\n***Test2: multi-line strings\n")
          ansi b =   "\nThis is a longer string\
               which spans several lines\
               and is alligned with the first line"
          Window <- Print(b)
          Window <- Print("\nThe string '[b]' is [b.len()] bytes long")

          -- 3. String assignments

          a = "first"
          b = "second"

          -- print the current values of a and b
          Window <- Print("\nNow a = '[a]' and b = '[b]' ")

          -- we swap a and b
          a,b = b,a
          Window <- Print("\nSwapped a and b -> now a = '[a]' and b = '[b]' "

          -- 4. String comparisons

          -- 0 means equal, negative earlier and positive later in the alphabet
          a = "abcdef"
          b = "abcdghij"
          Window <- Print("\ncompare([a],[b]) returns [compare(a,b)]")
          Window <- Print("\ncompare([b],[a]) returns [compare(b,a)]")

          if a is b then
           Window <- Print("\n'[a]' and '[b]' are equal")
          else
           Window <- Print("\n'[a]' and '[b]' are not equal")
          end

          a is a ? Window <- Print("\n'[a]' is always equal to itself !")
          a > b  ? Window <- Print("\n'[a]' > '[b]'")

          if a is <ansi>"abcdef" then
           Window <- Print("\n'[a]' and 'abcdef' are equal")
          else
           Window <- Print("\n'[a]' and 'abcdef' are not equal")
          end

          -- 5. Modifying a string

          Window <- Print("\n\n***Test5: Modifying a string\n")

          -- 0x58 is a byte - careful with non-ansi strings when doing this
          a[3] = 0x58

          -- as we want to print a[3] we put an escape before the [
          Window <- Print("\na\[3] in '[a]' was changed to [a[3]]")
          -- the line above prints “a[3] was changed to X”

          -- converting to upper / lower case
          a = "Kathy and Peter lived in Spain for 2 years."
          Window <- Print("\nString:     '[a]'")
          a.upper()
          Window <- Print("\nUpper case: '[a]'")
```

```
-- finding an replacing a substring
a = "Bob and Alice share the same password :'1234' !"
var i = a.find("1234")

-- overwrite the 1234 - we copy only 0:3, not the terminating 0
a[i:i+3] = "abcd"
Window <- Print("\n[a] - much better now.")

a = "Neil Armstrong was the first man on the moon."

-- searching for a byte
Window <- Print("\nFound 'A' at position [a.find(0x41)] in '[a]'"

-- 6. Appending strings

Window <- Print("\n\n***Test6: Appending strings\n")

-- there are several ways to append strings ...
-- some are shorter more readable then others
-- what to use depends on the circumstances
-- we just explore a few here

ansi Intro = "My name is"
ansi Name = "Ozymandias"
ansi Occupation = "king of kings"
ansi Motto = "look on my works, ye Mighty, and despair!"

-- 6.1 first method - create a formatted string - commas and spaces
-- added where necessary - simple and sweet

ansi First = "[Intro] [Name], [Occupation], [Motto]"
Window <- Print("\nFirst method (formatted string): [First]")

-- 6.2 second method - use the append function for strings
ansi Second
Second.append(Intro)
Second.append(" ")
Second.append(Name)
Second.append(", ")
Second.append(Occupation)
Second.append(", ")
Second.append(Motto)
Window <- Print("\nSecond method (append method): [Second]")

-- as append returns an ansi string we can also
-- do the following..
Second = ""
Second.append(Intro).append(" ").append(Name).append(", ")
     .append(Occupation).append(", ").append(Motto)
Window <- Print("\nSecond method (append chain): [Second]")

-- 6.3 third method : use the append operator
ansi Third = ""
Third = Intro + " " + Name + ", " + Occupation + ", " + Motto
```

```
          Window <- Print("\nThird method (append operator): [Third]")
     end
end
```

7.10 Class type

The *class* type is used for an external C++ - class. A variable of the type *class* contains a pointer to an object defined in a library that has been imported.

Below you can find a typical mechanism by which class names are imported into a design:

```
--------------------------------------------------------------------------
-- 3D groups
--------------------------------------------------------------------------
use math
use windows
use color, material

group d3d11 is lib XLib.3d11Lib

-- type mapping from the lib types to CellSpeak types
lib type int, float, double, void
lib type "Vector3<float>" is xyz
lib type "Matrix4x4<float>" is matrix4
lib type "Matrix3x3<float>" is matrix3
lib type colour is color.rgba
lib type MaterialType is material.rgba

-- lib classes - have methods but are otherwise unspecified
lib class CameraClass
lib class MeshClass
lib class CylinderClass
lib class SphereClass
```

The lines *lib class* import a number of classes from that library, and these classes can later be used as types for class variables in the CellSpeak code, for example:

```
SphereClass MySphere, YourSphere, HisSphere
MeshClass BigMesh
```

These class variables can then be used to call the methods on them provided by the external library. You can assign classes of the same type to one another and you can also compare classes for equality/inequality.

We will come back on the use of the type class when we discuss the integration with external libraries, but for the moment it suffices to say that external calls types can be used in CellSpeak as if it were local types.

7.11 Declarations

Types are used in declarations for variables, parameters and constants. A declaration starts with the type followed by the name(s) of the variable of that type.

```
int a, b, c               -- creates three int variables
int a = 4, b = a + 1, c = a * b -- variables can be initialised
```

A constant declaration starts with the qualifier const followed by the type, name and initialisation of the constant.

```
const float LightYear = 9.46e9   -- light year in km
```

If the type of the constant is clear from the initialization, then the type can be skipped.

```
const LightYear = 9.46e9        -- same as above
```

The same goes for variables. If they are initialized in the declaration, the compiler can normally deduce the type from the type of the initialization. The variable is in that case preceded by var.

```
var Speed = 0.0 -- a floating point variable
```

It is also possible to initialize variables using a type cast

```
var Axis = <xyz>[1,2,3]
```

As we have seen a variable can also be declared based on another variable

```
var Speed like Axis
```

Where the variable Speed is now of the same type as Axis.

7.12 Type casts

The CellSpeak compiler will convert a value of one type into a value of another type where this is logical and clearly the intended behavior.

In an operation with two operands of a different type, one of the operands will be promoted to the most precise type. Consider for example

```
byte b = 200
int I = 1000
float f = 1.7
i*b              -- 1200, an int value
f*I              -- 1700.0 a float value
```

In the expression i*b the byte will be converted to an integer before doing the multiplication. In the expression f*i, the integer will be converted to a float before doing the multiplication.

In an assignment the compiler will convert the right hand side to the type of the left hand side. With the variables defined as above this gives for example:

```
i = 2*b    -- the result is converted to an int = 400
f = i + b  -- the result is converted to a float = 201.7
```

However, if there is a loss of precision an explicit cast will be required by the compiler:

```
double d = 2.1
f = d * i   -- error - loss of precision, from double to float
f = <float>(d*i)    -- f = 2100.0 a float value
```

In the expression above, the integer will be converted to a double, multiplied with d and then converted to a float. Because you need to cast the result and not d, brackets are required. In this case not putting the brackets would actually give the same result.

In most cases the numerical behavior of mixed expressions is as expected, however it is always wise to pay some attention to expressions with mixed types. Consider the following

```
i = 2 * b   -- gives an integer result of 400 in our example
i = <byte>2 * b -- gives an integer result of 144
```

The result of the second expression is 144 because the cast transforms the constant 2 into a constant of the type byte and the multiplication becomes a multiplication between two values of type byte - effectively a multiplication modulo 256. The byte result is then transformed to an integer result of the same value.

Because of the types used in CellSpeak, the compiler will also attempt to convert expression lists of numbers to the predefined types:

```
xyz Axis = [0,0,1]
```

Here the compiler will convert the expression list of three integers to a vector type of three floats. Alternatively, one can also write

```
var Axis = <xyz>[0,0,1]
```

A type cast in general will require some operation to be executed on the data that is being cast. As type casts are operations, they can also be defined by the designer to modify - or not - the data. As an example consider the following:

```
word w = 1000
float f = w     -- implicit conversion from word to float, f=1000.0
```

7.13 Naming and scope rules

Now that we have seen how to define the types and variables that can be used inside a CellSpeak program, we also have to look at the scope of a type definition and a declaration.

Names are always part of a namespace. Designs, functions and message handlers each have their own namespace. Another way to create additional namespaces in a CellSpeak program is through the use of groups. The entity named *alfa* in a group *A* is different from an entity with the same name *alfa* in another group *B.*

To illustrate how names are resolved w.r.t. the different namespaces in a program, let us have a look at an example:

```
use group Z               -- assume group Z has a constant beta
group A
     const alfa
     const gamma
group A.B
     const alfa
     const beta
group C
     const alfa
group A.B.C
     const alfa
     function f() out nothing is
          const alfa
```

```
        -. Selection of some names in this function .-

        alfa             -- const alfa local to the function f
        beta             -- const beta from group Z
        gamma            -- const gamma from group A
        A.B.C.alfa           -- const alfa from group A.B.C
        A.B.alfa         -- const alfa from group A.B
        A.alfa           -- const alfa from group A
        A.B.beta         -- const beta from group A.B
        C.alfa               -- const alfa from group C
    end
```

The compiler will look for the name of a type or

- look for the name in the local namespace (in this case the function)
- look for the name in the use list (group Z in this example)
- look for the name in the ‘surrounding’ group hierarchy, working your way ‘outwards’

8 Designs

8.1 Introduction

The most important building block of the CellSpeak language is the *design* of a cell. The design of a cell specifies what a cell does and how it interacts with its environment and the other cells that make up the system or application.

The design of a cell can be very simple and small, as it can be big and complex. It all depends on the functionality of an application and how its tasks can be segmented and distributed over the cells. Some applications are better made from a small group of bigger cells, while for other applications the choice for many small cells is more logical.

Building an application or a system in CellSpeak is first about designing the *organization* of your application: how can the application be segmented in cells, what type of activities should each of the cells have, how will the different cell interact. When this has been sketched out – the process will probably take some iterations – the process of the design of the cells can start to see how they can provide the services they have to contribute to the overall application.

Writing the design of a cell is much like writing a traditional application – making exception for the message handling framework – because it is about deciding about the data and objects that you are going to use in the program, it is about writing the code that implements the functionality that the cell has to provide and so forth.

A cell design has several components

- The header containing name, parameters and inheritance.
- The private data section. This is the data in the cell that is accessible by the message handlers and the functions of the cell, and that has the same lifetime as the cell. We could also have called this the ‘global cell data’ section, because it is accessible from almost everywhere in the cell. However the name private was chosen, because the data is private to the cell, i.e. not accessible from the outside.
- An optional *constructor* and *destructor*. If present these are called at creation and destruction of the cell respectively.
- Local functions. These functions are defined at the highest level in the cell and have access to the private data of the cell.
- Message handlers. Message handlers also have access to the private data of the cell, and are executed in response to messages received by the cell.
- An exception table

Functions are useful, as in any program language, to structure the code that is used in the design of a cell, but it is perfectly acceptable, as is often the case with smaller designs, just to have the message handlers in the cell design.

The schematic example below contains the parts listed above:

```
design Planet(ansi Name, float Distance, float Mass)
                        like HeavenlyBody( Mass ) is

     keep Name, Distance, Mass
     float OrbitalPeriod = 2.73
     matrix3 RotationMatrix
     function StartRotationAround( xyz axis ) out bool

     constructor is
          xyz axis = [1.0, 0.0, 0.0]
          StartRotationAround( axis )
     end

     destructor is
          Sun <- Destroyed
     end

     function StartRotationAround( xyz axis ) out bool is
          RotationMatrix
     end

     on SayHello(cell To) do
          To <- Hello("Hello from planet [Name]")
     end

     on CollisionAhead(cell OtherPlanet) do

     end

     catch
          DivisionByZero do
          end
     end
end
```

8.2 Header

The header of a design consists of the keyword *design,* followed by the name of the design followed by the parameters that are used in the construction of the design:

```
design Planet(ansi Name, float Distance, float Mass)
```

The constructor of a design itself does not have parameters, but when it is called it has access to the parameters of the design. When you just want to declare a design and postpone the implementation to later or put it in a different file, you just put *to do* at the end of the declaration above:

```
design Planet(ansi Name, float Distance, float Mass) to do
```

Anything else that follows, including the inheritance, is part of the implementation.

8.3 Inheritance

A design can inherit from one or several designs. When one design inherits from another, it basically inherits all the data, functions and message handlers from the ancestor design. The design that inherited can then add additional data, functions and message handlers, but it can also redefine functions and message handlers of the ancestor design.

In our example the design *Planet* inherits from the design *HeavenlyBody*:

```
design Planet(ansi Name, float Distance, float Mass)
                        like HeavenlyBody( Mass )
```

The inheritance part starts with the keyword *like* followed by the comma separated ancestor design(s). We also have to pass the required parameters to the ancestor design, that will be used when the constructor for the ancestor is called.

A design can inherit from multiple designs. The compiler will complain of course if there are conflicts between the designs inherited from - either in the form of identical data field names or identical functions or message handlers.

Inheritance is a powerful mechanism to derive cells with more complex or specialized behavior from simpler or more general designs.

Note that a constructor in CellSpeak has no parameters, so inheritance is also used to derive cell designs that have a different parameter signature and/or initialization.

8.4 Private data

The content of the design that follows the header sits between the *is end* pair. As we have seen that content can be data, functions and message handlers.

Data that is defined at the highest level in the design , i.e. not in functions or message handlers, is permanent. It exists for the lifetime of the cell.

When data is declared in the design, that data can of course be initialized with an expression, so in many cases it will not be necessary to have a constructor for the cell because all of the initializations can be done in the declarations. The body of a cell – outside of the functions, handlers and the constructor/destructor - can actually also contain code – statements etc. That code is executed when the cell is created – before the constructor – if any – is called. But of course it is often better, if for example you need temporary stack based variables during the initialization, or just for the sake of clarity, to do the initialization of a cell in the constructor.

We have seen that in CellSpeak the keyword *keep* can be used as a shortcut in the definition for example of record fields. The keyword keep can also be used here for parameters as in the example above:

```
keep Name, Distance, Mass
```

The effect of keep is that three permanent variables are created and that they have the values of the parameters assigned to them. Any references to these names in the rest of the program, will be references to the permanent variables. The advantage of this construct is of course that if you want to keep parameters passed to the design, you can do so without additional declarations and assignments. You just 'keep' the parameter.

In situations where there is a conflict between for example a parameter name of a function and the field of a design, the construct *self.name* can be used to make the distinction.

8.5 Constructor

If the design has a constructor, then the constructor is called at the creation of the cell after the initialization code in the cell itself has been executed. The constructor has access to the parameters in the design header, and therefore the constructor has no header of itself. A constructor also has no return value. A definition of a constructor is thus always simply as follows:

```
constructor is
     …
end
```

When a design inherits from another design then the constructor of the ancestor design is executed first, and if there are multiple ancestors, the constructors are executed in the same order as they are declared, from left to right.

Note also that if the ancestor design has at its turn a predecessor and so forth, then this will result in a cascade of constructors being executed, starting with the 'deepest' constructor – the one furthest down the inheritance chain - first.

It was a design choice in CellSpeak to have only one constructor for a design, and not offer the possibility of a series of alternative constructors with different parameter signatures. The same effect can simply be reached by creating a new design, derived from a previous one, but with a different parameter signature:

```
design Sphere(xyz Centre, float Radius) is ..
design Sphere(float Radius) like Sphere([0,0,0], Radius) is end
```

As a final remark, the constructor is not a function and as such cannot be called directly from anywhere in the design itself.

8.6 Destructor

The counterpart of the constructor is the destructor. The destructor is called when a cell is destroyed. A cell can be destroyed either directly by using the *destroy* directive or indirectly because its parent cell is being destroyed. When a cell is destroyed, all of its children are destroyed first.

The syntax for invoking the destructor is

```
destroy CellRef
```

Where *CellRef* is a variable of the type *cell.*

A cell itself cannot just destroy any other cell in an application – it can only destroy its direct children. Invoking *destroy* on a cell that is not a child-cell has no effect and fails silently.

After the destructor is called for a cell, the system will release all the memory held by the cell and the cell will be removed from the cell-tree. The cell itself does not have to do any memory management in the destructor itself and does not have to destroy any of its children – it's not dangerous or harmful, but it is a waste of time because it is taken care of automatically.

Typical use of a destructor might be for example to send a final message to other cells - e.g. its customers - before disappearing.

If you destroy the top cell of an application, the complete application will be removed from the VM.

As a final remark, a cell can also auto-destruct:

```
destroy self
```

Which has the same effect as described above.

8.7 Local functions

Local functions are functions inside the design. Local functions have parameters and can have one or more return values. The main difference between local functions and global functions, i.e. functions that are not part of a design, is that local functions have access to the private data of the design.

Note that functions defined inside other functions do not have access to the private data of the design in which they are defined. For a more detailed discussion about functions, see the chapter on functions.

8.8 Message Handlers

Message handlers are invoked as a result of a message received by the cell. The message handler is selected based on the name of the message and its parameter signature. Remember that the parameter signature is based on generic type indicators and not necessarily on the types defined in the cell.

The name of a message can be a simple name, but can also be a string literal. Both headers for a message are acceptable:

```
on ReturnAddress( int CustomerNr ) do
     ..
end

on “How are you today ?”(int DayOfTheMonth, float Temperature) do
     ..
end
```

The name of a message can also be a list of names separated by a period. In

```
sender <- ReturnAddress.Reply(“Brussels”)
```

ReturnAddress.Reply is the name of the message.

The name of the message or the size of the parameter signature of a message have no influence on the selection process for the handler. The table with the handlers that every cell contains, is indexed using a hash key based on the message name and the parameter signature and is calculated at compile time – this results in a fast selection process, irrespective of the length of the name or the parameter signature.

Message handlers have no return value – if there is a result to return, they have to return the result by sending a message. The keyword *sender* is available inside a message handler to allow to return a message to the originator of the message being handled.

The message handler is not allowed to change the content of a message. It can take a copy, pass them on to another cell, use them in statements, but the message itself remains read only:

```
on Accelerate( float Speed ) do
     if Speed > 70.0 then           -- ok
          sender <- TooFast( Speed )       -- ok
          Speed = 70.0              -- not ok – msg is changed
     end
end
```

The reason for this is that messages are not needlessly copied for every destination that might need the message, so if some child cell would be allowed to change the message, it could have a hard to detect effect on the behaviour of its sibling cells.

If a message handler wants to return control before reaching the end of its code, it uses the keyword *yield* instead of *return*, which is reserved for functions.

A short message handler can also be written using the lambda notation we have already introduced for functions:

```
     on Get.Speed => sender <- Speed
```

Where *Speed* would presumably be some variable of the cell.

If a cell wants a message, that it has handled itself also to be handled by its children, it has to use the directive *flow:*

```
on Draw( cell Camera ) do

     .. drawing instructions

     flow

end
```

The position of the flow directive has no importance – it can be at the end or at the beginning of the handler and it can also be made conditional for example by using it in an *if then else* statement.

Internally the *flow* directive sets a flag to signal that the message has not be taken from the message queue, but must be passed on to the children cells.

Flow implementation

The way message handling works in CellSpeak, makes the flow directive a relatively inexpensive operation. Suppose that a cell say ten children, then the flow directive does not result in making ten copies of the message and sending a copy to each of the child cells. The thread that is executing the message handler simply keeps the message on its queue of actions and takes it along as it starts executing the handlers for the child cells.

8.9 Interfaces

It is also possible to group several message handlers together in an interface. An interface starts with the keyword *interface* followed by the name of the interface:

```
interface Service

            on Request() do
              ..
            end

            on Release() do
              ..
            end

end
```

An interface can either end by using the keyword *end* or by starting a new interface (or both). If a cell wants to send one of the messages of the interface, then the name of the interface must precede the name of the message:

```
ServiceProvider <- Service.Request(..)
```

An important feature of interface is that they can contain data that is only accessible by the message handlers of the interface:

```
interface Service

     float Cost

     on Request() do
          ..
     end

     on Release() do
          ..
     end

end
```

In the example above the variable *Cost* is permanent data – it has the same lifespan as the cell – but is only accessible by the code inside the interface.

8.10 *system* keyword

We have already seen two reserved keywords for cells : *self* and *sender.*

self is a keyword used to refer to the own cell and *sender* is a keyword that refers to the sender of the message being handled.

A third keyword that refers to a cell is the keyword *system.* This keyword actually refers to the local virtual machine. If a service is requested from the system, that service is obtained from the local virtual machine by sending a request to *system.*

Note that not everything that can be done on a given machine has to be obtained through the system. Functionality is mostly made available through the use of libraries that present a suitable API to the designer. For example working with files or displaying on the screen or using graphics are made possible by using the appropriate libraries.

There are however situations where the use of a cell that provides a service is more appropriate. A typical example of this is the *Connection Manager*. The connection manager is a cell that provides communication services to other cells. Because setting up a comms link can be a lengthy process where all sorts of things can go wrong, it is very convenient to request the connection manager to set up a connection, and return results or error conditions in the form of messages to the requesting cell – it provides for a clean and simple interface (see also the examples in the CellSpeak distribution).

The *system* cell is the very first cell that is created when the VM is started and the functionality of that cell is part of a package like for any other cell and can thus be modified as required for a given environment.

The system cell also usually provides a mechanism for other cells to offer services that can then be discovered by the cells in an application. Services can be anything – timers, access to the browser, communication protocols etc.

The line below can be found in the code of the constructor of the timer cell:

```
-- Offer the timer service to system (request a delivery notification)
system <+- Service.Offer( "Timer", self )
```

Where the cell signals to the *system* that it offers a service *Timer*. The second parameter is the id of the cell offering the service, in this case the cell itself - it could also have been one of its children cells for example.

Another cell will then typically request a service by sending a request to *system:*

```
-- Request a timer service from the system
system <- Service.Get("Timer")
```

And to capture the response, the requesting cell will have a message handler to handle the *Service.Provider* message from *system:*

```
     -- The timer service
     on Service.Provider( ansi Name, cell Provider) do

          -- we only requested a timer ..
          Name is "Timer" ? Timer = Provider   : yield

          …
     end
```

Where *Timer* is a local variable to store the reference to the timer cell.

9 Statements

9.1 Introduction

CellSpeak uses the same sort of statement used in many programming languages, with the occasional twist. While strictly speaking not a statement, we start this discussion with the expression.

9.2 Expression

An expression is a list of operations on the variables and constants of a program. In CellSpeak the assignment is also an expression meaning that the result of an assignment, the left hand side of the assignment, can again be used in an expression. In the examples below, we assume that the variables are of a suitably defined type:

```
a = b + 100     -- addition and assignment
c = (a=10)*(b=50)   -- a is 10, b is 50 and c is 500
xyz v = u # q       -- cross product between vectors
matrix4 M
xyzw v,q
v = M*q     -- matrix multiplication of a 4 comp. vector
```

Because an assignment is an expression, it is possible to do a chain-assignment :

```
var a=b=c=177       -- initialize three int variables to 177
```

The example below initializes a rotation matrix, given the axis *u* and a rotation angle *angle,* using the well-known Rodrigues formula:

```
function AxisRotation(xyz u, float angle) out matrix3 is

     float s,c,k,l

     s = sin(angle)
     c = cos(angle)
     k = 1 - c
     l = u.length()

     return [   c+u.x^2*k,      u.x*u.y*k-u.z*s,    u.x*u.z*k+u.y*s,
            u.x*uy*k+u.z*s, c + u.y^2*k,        u.y*u.z*k-u.x*s,
            u.x*uz*k-u.y*s, u.y*u.z*k+u.x*s,    c + u.z^2*k     ]

end
```

As we have seen, CellSpeak has two sorts of assignments: the normal assignment and the delayed assignment:

```
int a = 7, b = 5
c = (a ~ 2) * b -- result is 35 because a = 7 before the assignment
c = a * (b = 8) -- result is 16 because a is 2 and b is 8
```

The delayed assignment can be useful in loops and return statements where it can allow for a more concise notation by saving a temp variable and an extra assignment.

9.3 Multiple left hand sides

Several variables can be set in one assignment without having to use temporary variables as follows:

```
a,b = b,a  -- a and b are swapped
```

This can also be extended to any number of variables

```
a,b,c,d,e = b,c,d,e,a   -- permutation of the values among the variables
```

The number of variables at each side of the assignment must be equal of course, and pairwise of the same type.

9.4 Conditional statements

The basic form of an *if-then-else* statement CellSpeak is as follows

```
if condition then
     statements
else
     statements
end
```

The condition can be a Boolean expression, a link or an integer expression. If the condition resolves to a true value, a non-null pointer, or to a non-zero integer value, then the statements following *then* are executed. In the other cases the statements of the *else* part will be executed. If there is no need for an else part, then it can be left out. The if statement is terminated by a simple *end*:

```
if condition then
     statements
end
```

if then else statements are often used in a cascade:

```
if condition1 then
     statement1s
else if condition2 then
            statements2
      else if condition3 then
                 statements3
             else if condition4 then
                 statements4
                 else
                 statements5
                   end
              end
       end
end
```

This can be made more readable by using the *elif* keyword, whereby the cascade of *end* at the end of each *if then else* can be avoided:

```
if condition1 then
     statement1s
elif condition2 then
     statements2
```

```
elif condition3 then
     statements3
elif condition4 then
     statements4
else
     statements5
end
```

Note that the condition in an if-statement does not have to be put between brackets:

```
if Name is "Verstraete" then
     ..
```

Very often in program, you just need to do a simple check for which writing a complete if-then-else structure might be a bit too much, therefore in CellSpeak also the conditional expression can be used:

```
condition ? action if true : action if false
```

Again the condition can be an expression resolving to a Boolean value, an integer value or a pointer value. The conditional expression can be used in other expressions, e.g. and assignment:

```
max = a > b ? a : b
```

In CellSpeak the actions after the question mark are not limited to expressions, they can also be a directive or a *leave*, *continue* or *return* statement.

For example, a simple check of a pointer at the start of a function can be written as:

Pointer ? return

As seen in the example, if the second part of the conditional expression is not needed, it can be omitted. This allows for succinct expressions where appropriate:

```
a is 7 ? leave
R.next is null ? return
```

When statements are used in the conditional expression, it cannot be used as the right hand side in an assignment:

```
x = a is 0 ? return : a -- will generate an error
```

The example below is a function for transforming a null-terminated ascii string to upper case

```
function upper out nothing is
     for each b in this do
          b is 0x00 ? return
          b > 0x60 and b < 0x7b ? b -= 0x20
     end
end
```

This short form of the if-then-else statement is very handy and allows for a compact and clear notation. Because of this and because a single expression is often just not enough, it is possible to add a second part to the expression that can be a *return, yield, leave, continue* or *raise* statement:

```
a > 99 ? a = 0 & return
```

which is the short form of

```
if a > 99 then
     a = 0
     return
end
```

9.5 Switch Statement

The general format of the select statement is as follows:

```
switch expression

     case constant: statements
     case constant: statements
     case constant, constant: statements
     default: statements

end
```

Simple example:

```
string.utf8 Greeting
switch Age
     case 0,1,2:    Greeting = “ah boobie boobie baby”
     case 14:       Greeting = “hey whazup”
     case default:  Greeting = “Goodmorning,”
end
```

As you can see from this example not all possible values need to be supplied. Values that are not in the table can be caught by an optional *default* entry in the table. If no entry is found control just continues after the switch statement.

Also multiple constant values are possible in the select statement, separated by comma’s. The values to choose from can be expressions, but they must evaluate to a compile time value.

The values in a switch statement can also be string literals, and the selection variable can be any variant of a null-terminated string:

```
string.utf8 Name

switch Name
     case “Dostojevski”:    ..
     case “Shakespeare”:    ..
     case “Withman”:        ..
     case “Proust”:     ..
end
```

There are two important characteristics of a selection table: first there is no fall through. The statements behind the selected constant are executed and then the control transfers to the next statement after the *switch* statement.

Second, the compiler optimizes access to the table and the compiler sorts the select table. This makes sure that the selection process even in large selection tables remains short, namely of the order of *log(size of the table)*.

9.6 For loop

Loops are important in programs in many different situations. In order to allow loops to be written in a compact way and to minimize common types of bugs often associated with loops, CellSpeak has several variants of the *for-loop*.

The basic *for-loop* in CellSpeak is straightforward:

```
for variable = startvalue to endvalue  do
     statements
end
```

The loop works as expected: the variable is given a starting value and the statements between *do* and *end* are executed, then the variable is incremented (add 1) and the statements are executed again, and so forth. When the value of the variable reaches the *endvalue* the statements are executed one last time and then control transfers to the first statement after the loop.

This means that if the *startvalue* and the *endvalue* are identical, the loop will be executed exactly once. In the following example i is the loop variable:

```
for i = 7 to 45 do
     …
end
```

The loop variable does not have to be declared beforehand, it will be declared automatically as an integer. You can declare or re-use a suitable variable off course. The simple for loop also allows to write multiple loops in one line if required. The following:

```
for i=5 to 10 do
     for j=8 to 45 do
          for k=2 to 17 do
            ..
          end
     end
end
```

can be conveniently re-written as

```
for i=5 to 10, j=8 to 45, k=2 to 17 do
     ..
end
```

The second format of the *for-loop* is to loop over every element of an array.

```
for each v in array do
     statements
end
```

Example:

```
type HealthRecord is record with
     ansi  Name
     float Length
     float Weigth
     float BMI
end
```

```
HealthRecord     Population[n]

for each H in Population do
     H.BMI = H.Weigth / H.Length^2
end
```

In the example above the variable H will cycle through all the elements of the array, one at the time starting at 0 and ending after handling the n-1 th element.

The advantage of using the *for each* loop is that you not have to worry about the size of the array you are applying this loop to, because in CellSpeak an array keeps track of the number of elements it has.

Also here, the variable H does not have to be declared beforehand, as its type can be deduced from the type of the elements of the array.

The variable in the loop effectively refers to the element in the array, meaning that it is not a copy of the element. Modifying the variable means that also the element in the array is changed.

Sometimes also the indices of an element in an array are needed. For this reason another variant exists, the *index* variant. The format of that loop is:

```
int A[n,m]
for each [i,j] in A do
     A[i,j] = i*j
end
```

In this case A is a two-dimensional array, and i and j will take on all possible *index* values for that array as follows:

```
[0,0]..[0,m-1]
[1,0]..[1,m-1]
..
[n-1,0]..[n-1,m-1]
```

Again, the two index variables i and j, need not be declared beforehand. They are placed between square brackets to indicate that they loop over the index range of the array.

Another variant of the for loop allows to loop over every value in an enumeration:

```
enum Planets = [    Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus,
                Neptune, Pluto]

for each Planet in Planets do
     ..
end
```

It is also possible to loop over the values in an expression list (a list of values)

```
const pi = 3.141569
for each x in [pi, pi/2, pi/3, pi/4, pi/5, pi/6] do
     y = sin(x)
end
```

9.7 While do and repeat until

The while-do and the repeat-until loops allow for loops for which the end depends on a condition, rather than on a fixed number of repetitions.

The format of the *while do* loop is as follows:

```
while condition do
     statements
end
```

The condition is evaluated before each execution of the loop and the statements are executed if the condition is true, non-zero or not equal to *null*.

```
repeat
     statements
until condition
```

The repeat statement is the same, with the exception that the condition is evaluated at the completion of each loop. The *repeat* statement is thus executed at least once.

9.8 Loop Statement

For these situations that cannot be handled conveniently by the above loop statements, CellSpeak also offers a generalized loop statement that has the following format:

```
loop (expression, expression, expression) do
     ..
end
```

or put differently

```
loop (initialisation, condition, update) do
     ..
end
```

Before the loop starts the initialization expression is executed, then before the execution of each loop the condition is checked and at the end of each loop the updates are executed. The initialization part and the update part do not have to be present, but the condition part is required. A few examples:

```
loop (float v = -9.23, v < 12.6, v += 0.01) do
     ..
end

loop (,x<0.5,) do
     x = randf()       -- randf returns a random float between 0 and 1
end
```

Multiple initializations and updates are possible by putting these between square brackets:

```
loop ([i=1,j=7], i*j < 512, [i+=3,j+=2]) do
     ..
end
```

There can off course only be one conditional expression.

9.9 Leave and continue

Closely linked to the loop statements are the *leave* and *continue* statement. The *leave* statement will transfer control to the statement following the loop it is executed in. The *continue* statement will restart the next iteration of the loop, without executing whatever follows the *continue* statement.

```
for each L in List do
     L.Height > 180.0 ? continue
     ..
     L.Name is “George” ? break
end
next statement
```

In this example if the first condition is fulfilled, the loop will restart with L taking the next value in the array (if any). If the second condition is fulfilled, then the loop is terminated and control is transferred to *next statement.*

Statements in CellSpeak can be grouped together and given a label. A statement is given a label by preceding the statement with the keyword *task* followed by the name of task. That label can then be used in the *leave* and *continue* statements to make it clear where control will go to. The folowing example calculates the first 100 primes:

```
     const n = 100
     int Primes[n]
     int NrOfPrimes = 1
     int SquareRoot

     -- The first prime is 2
     Primes[0] = 2

     -- Start at the next candidate, 3
     task FindPrimes
     for (i=3,true,i+=1) do

          -- Calculate the integer square root of i
          SquareRoot = sqrt( <float>i )

          -- Check with the primes already found
          task TestPrimes
          for j=0 to NrOfPrimes-1 do

            -- Check if the prime divides i
            i % Primes[j] is 0 ? continue FindPrimes

            -- We can stop if the prime is bigger then sqrt(i)
            Primes[j] > SquareRoot ? leave TestPrimes
          end

          -- found another prime, add it
          Primes[ NrOfPrimes +~ 1 ] = i

          -- If we found enough, we stop
          NrOfPrimes is n ? leave FindPrimes
     end
```

A label can also be given to a group of statements by enclosing the statement group between a *do* – *end* pair:

```
task SomeCalculation do
     …statements
end
```

The *leave* statement can then be seen as a sort of return with which to leave that block of statements.

9.10 Return Statement

The return statements returns a value from a function. A detailed discussion will be given in the next chapter on functions. The format of the statement is mostly straightforward:

```
     return expression
```

As functions can return several values the most general form of the statement is

```
return expression, expression, ..
```

with an expression for each value that needs to be returned, in the order of their declaration in the header of the function.

9.11 Raise Statement

The *raise* statement raises an exception. Raising an exception starts a process whereby the call stack is searched for a function a handler or a design that handles the exception. More details about this process can be found in the chapter on exceptions.

The format of a raise statement is

```
raise exception_name(parameters)
```

The syntax of an exception resembles the syntax of a function call, but the handling of an exception is very different.

10 Functions

10.1 Introduction

Functions are a way of structuring code into small pieces in order to increase readability and reuse. In CellSpeak where we can make a distinction between four types of functions

- Functions defined outside of a design.
- Functions inside a design.
- Nested functions (inside other functions)
- Methods of records.

In CellSpeak functions can be constant functions or variable functions. Much like any other constant, a constant function has its body defined once and afterwards that definition cannot change anymore. In contrast, the body of a function variable (or variable function) can be changed as required.

Functions can be passed as parameters, and when a formal function parameter is defined, you can pass a constant function or a variable function as the actual parameter in exactly the same way. The same as you would expect for, say, an integer parameter. When a function is passed as a parameter, the only thing that is checked is that the parameter signature of the formal and actual function parameter match.

As is the case for other constants, constant functions do not take up data space in the design or record that they are part of, whereas variable functions, like variables, do take some data space.

10.2 Function format

A function consists of a header and a body. The function header specifies the name of the function, the parameters and the return type. The body of the function contains the statements to be executed when the function is invoked.

```
function name (type1 param1, type2 param2) out type3 is
     statements
end
```

Function parameters cannot be modified inside the function by default. If you want to be able to modify a parameter inside a function you have to use the qualifier *out.*

Functions can also have multiple return values, so you can group all output parameters in the output part of a function.

```
function name (type1 param1, out type2 param2) out type3, type4 is
     statements
end
```

You can also give a name to the output parameters as in this example,

```
function Count() out (int Result, bool Error)
     statements
end
```

but the names of the output parameters are purely for documentation puposes, as these identifiers are not used inside the function – values are returned with a *return* statement.

If a function has no parameters, you can omit the brackets and id a function does not produce an output, you can also leave out the *out* section:

```
function name is
     statements
end
```

When calling a function you always have to use the brackets, even if there are no parameters, to allow the compiler to make the distinction between a function call and the use of the function in an assignment or as a parameter for example.

A short function with a body that consists of a single expression can be defined using a *lambda arrow.* The return type is deduced from the type of the expression:

```
function name(parameters) => expression
```

Example:

```
function Cardioid( float t ) => <xy>[ 2cos(t)-cos(2t),  2sin(t)-sin(2t) ]
```

The definition of a method in a record is identical to a function definition, but then inside a record:

```
type recordtype is record with
     type1fieldname1
     ...
     function methodnname (type1 param1, out type2 param2) out type3 is
          statements
     end
end
```

If, for reasons of readability for example, you do not want to keep the body of the function inside the record definition, you can declare the function but have the implementation of the body of the function later:

```
type recordtype is record with
     type1fieldname1
     ...
     function methodname (type1 param1, out type2 param2) out type3 to do
end

function recordtype.methodname(type1 param1, out type2 param2) out type3 is
     statements
end
```

Functions can have the same name. Functions with the same name will be considered different as long as their parameter signature is different. Functions with the same name and the same parameter signature but only a different return value will be signaled as an error by the compiler.

10.3 Function type

In CellSpeak functions can be modelled after other functions by simply using the *like* keyword:

```
function TriGon( float angle) out float to do
function sin like TriGon is
end
function cos like TriGon is
```

```
end
```

In this way an existing function – or just the header of a function - can be used as the model for a series of functions.

10.4 Function as a parameter

The following example shows how a function parameter is specified and how an function is passed as a parameter:

```
-- We define an array type with member functions as follows
type NumberList is int[] with

-- a function to put numbers in the list
function Populate is
     for each [i] in this => this[i] = i
end

-- A function to apply a function to all element
function Apply(function f(int) out (int), float a) is
     for each N in this => N = a*f(N)
end

-- We define a variable of the type with size 20
NumberList[20] MyList

-- We set the elements of the list
MyList.Populate()

-- We define a simple function
function Square(int x) => x^2

-- And call apply
MyList.Apply( Square, 10.0 )
```

The result of the example above is that all elements of the array are squared and multiplied by 10.

Note that in the definition of the function parameter, it is not necessary to give names to the parameters of the function (you can if you want to). It would also have been possible to use function type

```
function f like SomeFunction
```

to specify the type of the function, as explained in the previous section.

10.5 Scope rules for functions

Functions have access to their own local data and to the parameters passed to that function.

Functions at the highest level inside a design, have access to the private data of the design.

Methods of records have access to the fields of the record for which they are defined.

Nested functions, i.e. functions defined in other functions or inside a message handler, only have access to their own data and their parameters.

10.6 Functions with fields

In CellSpeak it is possible to define data, at the time of the creation of a function, that will remain available to that function each time it is called. It allows for example for taking a snapshot of variables that are in scope when the function is created (closure) and to use these variables when the function is invoked.

The fields, that only the function will have access to, are defined between the header of the function and the body:

```
function f( parameters ) with
      int a = 8
      float b = 3.1415
end is
      .. body of the function
end
```

The two fields introduced, *a* and *b*, get a value at the definition of the function, and are accessible from within the function each time the function is called.

Suppose that you want to capture the value of some variables in scope when the function is defined, then you can do this as follows:

```
int A = 5
float B 2.2

function f( parameters ) with
      keep A, B
end is
      .. body of the function
end
```

In the example above, the function will make a copy of the two variables in scope, A and B, and will be able to use these variables when the function is called.

If you only capture variables in scope using the *keep* keyword, then you do not have to use the *with end* brackets.

```
function f( parameters )
      keep A, B
is
      .. body of the function
end
```

Data that is captured in this way, is released when the function gets out of scope.

An example to clarify:

```
design CubeWithAction(float Rib) is

      -- we create a simple function variable
      var function Action out nothing
      matrix4 WorldMatrix

      -- message to start rotating: we will initialize the Action
      on StartRotating(xyz Axis, float AngularSpeed) do
```

```
          matrix4 RotationMatrix   -- declare a 4x4 rotation matrix
          Axis.normalize()     -- the axis must have unit length

          RotationMatrix = <matrix4>axis(Axis,AngularSpeed)

          -- here we define the body of the action
          body Action
           keep Axis, AngularSpeed, RotationMatrix
          is
           WorldMatrix = RotationMatrix*WorldMatrix
          end
     end

     -- message to draw the cube for the next frame
     on Draw(int FrameCount) do
          Action() -- execute the function that was defined beforehand
          ..       -- do some other stuff to draw the cube
     end
end -- design
```

In this example, we define the body of the variable function when we receive a message to start rotating, and as part of that definition we also keep a copy of some of the data that is visible inside that function at the moment of definition.

At a later stage when we call the function, we just pass the parameters to the function as defined - in this case the function has no parameters - and the function will use the captured data as it was at the moment of the definition of the function. Note that inside the function the captured variables (or function fields) can be modified like any other data in the function, the difference being that they preserve their values between function calls. Function fields cannot be changed or accessed from outside the function.

Both constant functions and variable functions can have function fields.

10.7 Variable functions

As we have seen, constant functions are pieces of code that do not change after having been defined. Variable functions are also containers for code, but here the code can change.

The body of a variable function can be defined in three ways:

First it can be defined when the function is declared, as for a constant function:

```
var function Quadratic(float x, float y) out float is
     return x^2 + y^2
end
```

The function Quadratic is a variable function and it is initialized with the code following the header.

The second way to change the body of a variable function is to assign another function to it - another variable or constant function:

```
Quadratic = Ellipsoid
```

where Ellipsoid is some function with the same parameter signature and return value as the function variable *Quadratic.*

Functions can be parameters as we have seen, and Functions can be returned as results

```
function GetQuadratic out function F(float x, float y) out float
Quadratic = GetQuadratic()
```

The third way to change the body of a variable function is by redefining the body of the function - *body* is a keyword:

```
body Quadratic is
     return x^2 - y^2 + xy
end
```

This simply redefines the body of the function Quadratic. As a function variable is unique, there is no need to repeat the parameter signature of the function.

10.8 Sending functions in a message

Functions are code and not data, and cannot be sent as parameters in a message. However, as we will see it is possible to send bytecode files between cells, that can then be loaded and executed by the receiving cell.

11 Templates

11.1 Introduction

Templates in CellSpeak are straightforward. Templates allow to write code that can be re-used for different types. A template is written with one or more types in the template left undefined, and then the template can be instantiated by specifying the actual types for the code.

It allows for example, to write code for functions that have to work for float and double values, only once.

In CellSpeak not only the type(s) can be changed from one instantiation to another. The parameters to the template can actually be any identifier used in the template (type, variable, function etc.)

11.2 Example

The following example defines a template for a generalized hash table. The heading starts with the keyword template followed by the parameters of the template:

```
template HashTable for NewType, Type, Field, Size is
```

When the template is instantiated for a specific value of the parameters, it is done as follows:

```
use template HashTable for HashTableType, MyRecord, Name, 10
```

Which matches the parameters in the template definition one by one. The compiler will then do the replacements in the template definition and compile the template.

The code for the template starts after the keyword *is,* and the template itself ends with the keywords *template end.*

The following is a small part of an example of a template (it can be found in the CellSpeak distribution).

```
template HashTable for NewType, Type, Field, Size is

-- The type that is stored in the table - we add a field Next
type Type##InList is Type with
     Type##InList.ptr   Next
     int key
end

-- The table type
type NewType is record

     const NrOfTables = 8

     Type##InList.ptr Table[ 1 << Size ]       -- The hash table
     Type##InList     BackStore[NrOfTables][]   -- The table with the records
     Type##InList.ptr FreeList          -- The list of released rcrds
     int     CurrentTable           -- table in use
     int    NextFree             -- the next free element

     -- we include an output cell if we want to test the structure
     <#test cell Out #>
```

```
-- a function to initialise the hash table
function Init is
     CurrentTable = 0
     NextFree = 0
     for each R in Table => R := null
     FreeList := null
     setsize BackStore[CurrentTable][ 1 << Size ]
          <#test Out <- print("\nBackstore table nr [CurrentTable] has [elemin
          BackStore[CurrentTable]] elements.") #>
end

…
template end
```

There are two additional things to point out in this template file:

```
Type##InList
```

The double hash will make sure that the name that is substituted for *Type*, is put together with *InList*, to form one identifier, in our example this would give *MyRecordInList.*

The second element is:

```
<#test    cell Out #>
```

In a template you can include code conditionally. In the line above it means that if *test* is defined, then the line *cell out* will be included in the instantiation of the template.

Which code has to be conditionally included, is determined when the template is instantiated:

```
use template HashTable for HashTableType, MyRecord, Name, 10 with test
```

where the *with test* tells the compiler to include all the code between *<#test … #>* brackets. Note that code between conditional brackets is not limited to a single line of course. If there are many conditional compiler flags they are separated by comma's.

The implementation of templates in CellSpeak is straightforward and simple, but nevertheless provide a powerful mechanism to write reusable code, for example for generalized container classes like lists, hash tables etc.

12 Memory management in a CellSpeak program

12.1 Introduction

In CellSpeak, when we discuss memory management, there are several topics we have to address:

CellSpeak allows to have pointers to records and sizes of arrays can be set and changed dynamically. This means that there is allocation and de-allocation of memory and so there must be some type of memory management.

The choice for using pointers in CellSpeak was made because it allows to build interesting data structures. Working with pointers in CellSpeak has been made as easy as possible to relieve the designer of much of the maintenance that is typically associated with pointers. But it remains a fact that working with pointers introduces additional risks and requires designer attention

Of course, inside the CellSpeak bytecode, pointers to all types are used extensively, but this is done transparently to the designer. An *out* parameter of a function for example is obviously a pointer, but this can be transparent to the designer, in the sense that no special notation is needed to dereference that parameter as compared to another parameter.

The second topic we have to address is that CellSpeak can call methods and functions in C/C++ libraries, so there must also be a mechanism to deal with C/C++ pointers.

As we have seen, cells in a CellSpeak program do not share data, and any data allocated by a cell is only accessible to that cell. Cells can also not send pointers between them – actually they can, but the pointer is set to null – because different cells can live on different systems or processes, where a pointer from another system or process does not have a meaning. This means that pointer references are limited to a single cell, and that memory management can be done at the level of the cell.

On the one hand this fact simplifies things, because, if a cell is being destroyed, all the memory held by that cell can simply be returned to the system without the risk of leaks or leaving stale pointers around.

On the other hand, it complicates matters because there can be many cells in a CellSpeak program, and if memory management functions have to be done for each cell, this could give a considerable overhead. The CellSpeak solution to this problem is the use of scratchpad memory.

12.2 Scratchpad memory

As we have seen, the behavior of cells is that after their creation, they wait for incoming messages and execute message handlers as required. This also means that any allocation that has been done during the processing of the message and that is not needed afterwards, can be released automatically.

This situation arises often – in the process of handling messages, often structures, strings and arrays have to be created to process the message, but are no longer needed when the message has been handled. Functions can be called as required in the handler while preparing the response and these functions that will have access to the dynamically allocated data as required, but once the message handler has been completed, the scratchpad memory is simply reset.

The maintenance of this type of memory is very simple – you do not have to keep track of all the allocations that you do – at the end of the handler the scratchpad memory is simply set to its initial value again – no losses, no maintenance.

Of course, if the cell wants to keep some data permanently allocated, it can do as well. We will see examples later on, but how the difference is made is simple: allocations to global cell data are permanent – they exist until released or until the cell has been destroyed, while allocations to local variables in handlers and functions are done on the scratchpad. The designer does not have to specify himself where the allocation has to take place, this is done automatically in a transparent way.

12.3 All records and arrays are dynamic

All records and arrays are allocated, even the record variables that are not declared as pointers or the arrays with a fixed constant size.

```
type Address is record
     ascii  Street
     int   Number
end

Address    MyAddress
Address.ptr     HerAddress
```

The difference between *MyAddress* and *HerAddress* is that the compiler will automatically also generate the allocation instruction for the *MyAddress* variable.

The same goes for arrays:

```
int Numbers[10]
```

In the code above, the compiler will add code to allocate the space for the array of 10 integers.

In the following declaration:

```
Address Directory[100]
```

The compiler will allocate an array that can hold 100 records of the type *Address*, not a record of 100 pointers to a record. If that is what you want, you have to declare it as follows:

```
Address.ptr Directory[100]
```

The fact, that *all* arrays and records are dynamic, is of no further concern to the designer, but it makes the handling and reasoning about records and arrays in the CellSpeak compiler uniform.

12.3.1 The reference assignment

The only difference in a CellSpeak program between a pointer variable and a direct variable is in the use of the reference assignment:

```
HerAddres := MyAddress  -- HerAddress holds a pointer to MyAddress
MyAddres := HerAddress  -- Illegal, MyAddress is not defined as a pointer
```

The reference assignment can also be used for arrays:

```
int A[10]
int B[]
```

```
B := A      -- B refers now to the same array as A
A := B      -- also ok
```

Some other examples:

```
Address    D[ 100 ]
Address.ptr E[ 100 ]

D[ n ] = HerAddress     -- makes a copy
D[ n ] := HerAddress    -- illegal, D is not an array of pointers
E[ n ] := D[ m ]        -- E holds a pointer to D[ m ]
```

Let us have a look now at how variables are allocated in CellSpeak.

12.4 Allocating memory for records

A record is allocated using the *new* keyword followed by the name of the variable, not by the name of the type as is the case for example in C/C++. Example

```
type SongDescription is record with
     utf8  Title
     utf8  Performer
     float Duration
     int   ChartsTop
end

SongDescription.ptr Song

new Song
```

The *new* directive allocates memory for a record of the type *SongDescription*, and puts a pointer to that allocated space in the variable Song.

If *Song* is a global variable for the cell, then the allocation will be done on the heap, if *Song* is a local variable in a function or message handler, then the allocation will be done on the scratchpad memory.

Suppose we want to make a linked list of songs, we can do this as follows:

```
type SongDescription is record with
     utf8       Title
     utf8       Performer
     float      Duration
     int        ChartsTop
     SongDescription.ptr    Next
end

SongDescription SongList
```

To add a song to the *SongList*, we simply allocate as follows:

```
new SongList.Next
```

We can then start to assign some fields like this:

```
SongList.Next.Performer = “Neil Young”
SongList.Next.Duration = 35.50
etc..
```

If we have prepared another variable, we can also assign to the link:

```
SongDescription.ptr NewSong

-- set some value to NewSong
NewSong.Performer = “Outkast”
etc..

-- assign to the link
SongList.Next := NewSong
if NewSong != nothing then ..
```

Pointers can be checked against the built-in constant *null:*

```
if SongList.Next is null then ..
```

This in turn is equivalent to

```
if SongList.Next then
```

which, as we have seen, can be written as

```
NewSong ? ..
```

If there is only a limited action to be executed after the question mark.

Pointers are always initialized to *null* and dereferencing a null pointer fires an null-pointer exception that, if required, can be caught by an exception handler.

De-allocating is done using the *delete* keyword:

```
delete NewSong
```

Delete will also set the pointer to null.

Deleting a *null* pointer has no effect.

12.5 Allocating memory - arrays

As we have seen, arrays are always dynamic in CellSpeak, even if they have a compile time fixed size, arrays are only allocated at run time.

Arrays can be given a size when they are declared and that size can be a constant or a variable:

```
int A[]
int B[5]
int C[n]   -- n is a variable
```

An array in CellSpeak Is not just a pointer, but also carries some extra information, such as the size of the array. Accessing an element that is not part of the array causes a *range exception*.

We have seen that arrays can be initialized by assigning an expression list (list of values) to the array:

```
A = [1,2,3,5,7,11,13,17,19] -- A is an array of 9 elements
```

When an array has not been given a size, the array will take the size of whatever is assigned to it (of the right array type off course). In our example the size of *A* is not specified, and so A is flexible w.r.t. its size.

The assignment

A = [1,2,3]

will work and changes A in an array of three elements now, with the previous elements being lost. An array that has been given a fixed size in the declaration, will not change size as a result of an allocation.

Arrays can however be resized:

```
resize B[10]        -- B has 10 elements now, previous elements are lost
```

When you do a resize, the previously allocated space is released automatically.

Arrays themselves are not pointers, but hold a pointer and can be assigned using the reference assignment:

```
A := B
```

A and B will now point to the same array. If B is resized, A will still hold the pointer to the original array.

Also for arrays the distinction between global arrays and local arrays – arrays in functions and message handlers – hold. Global arrays are allocated on the heap of the cell, whereas local arrays are allocated on scratchpad memory and have virtually no memory management.

12.6 Scratchpad memory revisited

Does the use of scratchpad memory and a heap make the whole memory allocation business in CellSpeak more opaque or simpler?

Well, we think it definitely makes it a lot simpler – also for the designer. Suppose you see a line in a program as follows:

```
new variable
```

For a designer the situation is immediately clear, if that variable is a global cell variable, then that variable will be allocated on the heap of the cell, and if the variable is a local variable, the variable will be allocated on the scratchpad. Only the variable on the heap might require afterwards some extra attention and needs to be deleted explicitly if no longer needed. Variables allocated on the scratchpad memory do not need further attention – they are deleted automatically when a message handler completes. So the difference between a variable that is allocated on the heap and a variable that is allocated on the scratchpad memory, is as clear as the difference between a global variable and a variable on the stack.

But what about parameters of functions? Suppose that a function is declared as follows:

```
function f( out SongList.ptr R) is

     …
     new R
```

```
end
```

Where will R be allocated? Well the answer simply is: if the parameter that is passed to the function is a global variable then R will be allocated on the heap, if it is a local variable, R will be allocated on the scratchpad.

This information is available to the function at run time, because when the pointer is pushed on the stack for the function, there is also some extra information pushed on the stack, such as where to allocate for that particular parameter, so the allocator always knows where to allocate the parameter.

13 Interfacing with external libraries

13.1 Introduction

CellSpeak is a young language with not a lot of software written in it, so it is essential that programs written in CellSpeak can interface with libraries written in other languages. CellSpeak was therefore designed to interface with external systems.

There are two ways that a CellSpeak program can interface with other systems. First of all, through its messaging interface. An external system can send messages to a CellSpeak Virtual Machine. For example if a cell makes a connection to the browser using the TCP/IP address and port number, then that cell will be able to communicate with the browser and the communication manager of the Virtual Machine will take care that messages sent and received are converted to the format used by both participants in the communication.

Messages can also be sent from inside a C/C++ library by using a SendMessage function that is supplied by the VM to the library, when that library is loaded.

A second way that a CellSpeak program can interface with and external library is by calling directly functions and methods that are exposed by that library.

CellSpeak has defined the built-in type *class* for that, variable of the type *class* are C++ objects that can be manipulated directly in C++.

A binary library contains the information necessary to identify the classes and methods, and the parameter signature of these methods, but it does not contain a description of the types itself. Therefore, the interface to a library also has to specify the correspondence between an external type and the CellSpeak type.

In the following sections we will show by example how an interface to an external library is made. The examples that we use are based on the DLL's available under the Windows OS.

13.2 DLL Interface specification

An external library is declared as a group by itself in CellSpeak and the library is imported as follows

```
group d3d11 is lib "..\\TestBench\\Debug\\Direct3D11.dll"
```

This simply states that the library *Direct3D11.dll* will be known as the group d3d11 in the CellSpeak environment. The group *d3d11* can still contain other CellSpeak types, designs and functions, but it can only contain one library.

We have seen that a project file can contain a constant that is set to the name of the library file, so that we do not have to use explicit file names in the CellSpeak source file:

```
group D3D11 is lib Direct3D11Lib
```

When the compiler reads this statement, it will read the entire DLL and analyze what is in the DLL: the classes, the methods and the names of the types. There is an option in the compiler to print what it has found in the library – this can be useful when debugging.

The next step that needs to happen is that the CellSpeak program tells what classes and functions it wants to use from the external library, but in order to use the methods and functions, there must first a link be established between the types used in the external library and the types in CellSpeak. The general format of an equivalence is:

```
lib type LibraryType is CellSpeakType
```

which states that the two types are compatible. Compatible means that when an external method is called with a parameter of type *CellSpeakType*, then the compiler will search the function in the external lib using the type *LibraryType*. The following example comes from the d3d11 library

```
-- type mapping from the xlib types to CellSpeak types
lib type int, float, double, void
lib type Colors is color.rgba
lib type MaterialType is material.rgba
lib type "Vector3<float>" is xyz
lib type HWND__* is Hwnd
```

If the names of the types used in xlib (external library) are the same as the names of the types in CellSpeak, then the type name does not have to be repeated. The first line simply states that the xlib and CellSpeak know the types *int, float double and void* and that these types are pairwise equivalent in both languages.

The second line specifies that the type Colors in the xlib corresponds to the type *color.rgba* in CellSpeak.

C++ types can be derived from templates – the fourth line specifies that the xlib type *Vector3 of floats* corresponds with the xyz type in Cellspeak (also a vector of floats). Note that in this case the xlib type had to be put between quotes, because of the use of angled brackets in the C++ type.

The last line of the example tells the compiler that the type Hwnd is equivalent to a C++ pointer to HWND__. Because CellSpeak has no generalized pointers, we have to see how this works later on.

Some important remarks:

First of all, not all types that are used in the xlib need to have an equivalent in CellSpeak. Only the types that will be used in the CellSpeak program need an equivalent.

Suppose the following library function is called in the CellSpeak code

```
Fill(MyColor)
```

The compiler will determine that the type of the parameter to that function is *color.argb* and that the DLL type that corresponds with that type is *Colors.* He will then use that information to localize the function *Fill* in the DLL and call the function with the parameter specified.

Second important point is that in a DLL the compiler does not have access to the layout of the data, so the compiler cannot check if the data correspondence is correct – if size and/or structure are actually different between both types, this can cause problems, so it is up to the designer to check this.

And finally, one xlib type can only be mapped one CellSpeak type, more than several xlib types can be mapped to the same CellSpeak type.

For calling a method this poses no problem – based on the parameter signature of the function, the compiler can establish a link, but for return values of functions it can be necessary to cast these to the correct type.

13.3 Specifying classes

Classes are library constructs that are otherwise opaque to a CellSpeak program. What you can do from within the CellSpeak program, is call the methods of the class. The xlib classes that you want to use in the CellSpeak program have to be listed. It is also possible to change the name of the xlib classes, for example because they collide with the name of a type already in use. Some examples of class declarations:

```
lib class CameraClass
lib class MeshClass is D11MeshClass
lib class "VC::CubeClass" extends MeshClass is VC_CubeClass
```

The first line simply states that entities in CellSpeak the external class CameraClass can be used. This type can then be used to declare variables of this type in CellSpeak:

```
CameraClass MyCamera
```

And methods of this class can be invoked:

```
MyCamera.Initialize(..)
```

Internally the variable *MyCamera* is a pointer, but in CellSpeak the same dot syntax applies for calling a method on this class.

The second line says that the class *MeshClass* in the library is called *D11MeshClass* in the CellSpeak code, possibly to avoid a name clash or to adhere to a different naming convention.

The information in a DLL header is what it is and does not contain all the details that one would like in order to make the interface as automatic as possible. The format of a DLL does not duplicate the methods for a class that inherited the methods of another class. If that is the case it has to be specified manually in CellSpeak, which is what the third is about: it says that there is an xlib class *VC::CubeClass*, derived from *MeshClass*, that in CellSpeak is called *VC_CubeClass.* Note that we also have to put the external type between quotes, because of the double colons.

Classes in CellSpeak are always pointers, this means that the classes must always be created at the C++ side of the interface and passed as a pointer to CellSpeak. CellSpeak can also not access members of the class directly, only through the use of its methods.

Note that, as with the types, classes that are part of the library, but that are not going to be used in CellSpeak, do not have to be listed in the list of classes.

13.4 The pointer type

Many functions and methods in C++ expect a pointer as a parameter. CellSpeak does not have a pointer type and even the type that most resembles a pointer, the *ptr* type, has an internal format and is only used for records. However, CellSpeak allows to take the address of an actual parameter pointer for an actual parameter of a method or function from an external library. An example:

```
     -- now calculate the new view matrix and set the camera matrix
     CalculateViewMatrix()
     CameraMatrix = ProjectionMatrix * ViewMatrix
     D3DCamera.SetCameraMatrix(&CameraMatrix)
```

The variable *CameraMatrix* is a CellSpeak variable of the type *matrix4*, and *D3DCamera* is an external class. The method *SetCameraMatrix* expects a pointer to a matrix, not the matrix itself, so the pointer of the matrix is taken by using the familiar & operator. Note that this operator only works for parameters in calls to external functions.

13.5 Calling a function

Not only methods of classes, but also functions in a DLL can be called directly from CellSpeak. If the DLL contains a function that one wants to call directly from CellSpeak, the following has to be included in the interface definition:

```
lib function Printf
```

which simply states that *Printf* is a function that could be called. Note that the parameter signature does not have to be specified. In case of a name conflict, the function can be renamed:

```
lib function Printf is CppPrintFunction
```

Note that a function of a given name is either external or a CellSpeak function, i.e. once *Printf* is defined as an external function it is not allowed to define a CellSpeak function with the same name but with a different parameter signature.

13.6 Heap and stack

The heap and the stack on the CellSpeak side and on the C++ side of the interface are completely separated. The most important consequence of this is that memory that is allocated on one side, should be de-allocated at the same side of the divide. As *new* and *delete* in CellSpeak are only valid for CellSpeak items, any attempts to allocate or de-allocate classes will be caught as an error by the compiler.

13.7 Threads

CellSpeak is thread-safe by design. Cells are isolated and exchange data only through messages. C++ Libraries on the other hand can be thread-safe or not. When including external libraries in CellSpeak, this is something the designer has to consider.

If an external library is not thread-safe, the CellSpeak application can still be executed on a single thread, which can affect the performance of the application, but the application will run without the crashes it would have if run in multithread mode.

The second solution will require some re-engineering of the application, in order to make sure that requests/calls to the library are serialized, for example by interposing a dedicated cell between other cells and the library they want to use.

13.8 Examples

The CellSpeak distribution has several examples of integration with external libraries where the above is applied, for example the integration with d3d11 libraries to create graphical applications in CellSpeak.

14 Compiling a CellSpeak application

14.1 Introduction

In this chapter we will look at some of the practicalities of compiling a CellSpeak application. We will look at the compiler and its options, but we will not discuss the use of the IDE - the most important tool for a programmer - because CellSpeak is being integrated in several tools, and it would take us to far to discuss all of them. In an appendix we have listed however one - short - overview of one tool, very in which CellSpeak has been integrated, the Notepad editor. This integration can also be used as guideline for people who want to integrate the CellSpeak compiler in their favorite package.

CellSpeak has no REPL (Read Eval Print Loop) because it make less sense for a language like CellSpeak built around message sending. But as we will explain in the next chapter, the user has the possibility to interact with a running application directly by sending messages from the console to the running cells. This possibility can off course also be switched of when this is not desirable.

14.2 The project file

The basis for each CellSpeak project is the CellSpeak project file. The project file is very straightforward and contains two arrays of strings:

```
--------------------------------------------------
--   Project file for the cube of cubes application
--------------------------------------------------
use strings
group Project.CubeOfCubes

-- The source files for the project

const string.ansi Source[] = [
     "..\\Applications\\CubeOfCubes.cellsrc",
     "..\\Groups\\Camera.cellsrc",
     "..\\Groups\\Light.cellsrc",
     "..\\Groups\\Shape.cellsrc",
]

-- the bytecodes for this project

const string.ansi ByteCode[] = [
     "Math.cellbyc"
]
```

The first array of strings is the list of source files that are part of the project, and the second list of files is the list of compiled bytecodes that this project needs for compilation. Note that the bytecode files contain at least a list of exported designs which can be used in the source files to create cells from these designs, but the byte code file also contains the symbol table information for the objects in that file. The source code itself is off course no longer part of the bytecode file , but type information, constants, function signatures etc. are and can be used in the source files for the project.

Note that however, for example for the distribution of an application, that the developer can decide to strip the symbol table information from the bytecode file. For simply running a program, the symbol table information is not required.

The result of a compilation - if successful - is always another bytecode file that will contain the CellSpeak bytecode for the sources in the project. This means that there is no one on one link between a source file and a bytecode file. All source files are compiled together and the result goes into one bytecode file.

For small projects - just one source file - it is also possible to compile directly the source file without having to create a project file.

14.3 The compile command

The following command is used to compile a CellSpeak project:

```
cellspeak -p <project file> [ -b <byte code> -o <output file> -c -m -x -?]
```

or in case only a single source file needs to be compiled:

```
cellspeak sourcefile [ -b <byte code> -o <output file> -c -m -x -?]
```

The name of the compiler is cellspeak and the options have the following meaning:

option	*explanation*
-p	<project file> if no extension .cellprj is assumed
-b	binary bytecode file, if not present the project filename with extension .cellbyc is used.
-o	<output file> (errors etc), if not present stdout is used.
-c	output bytecode in a readable format - goes to the output file
-s	output bytecode in a readable format mixed with source code
-m	message check - check send and receive profiles
-x	xml output - goes to bytecode filename but with the extension .xml
-?	prints this usage overview

In the -p option the project file is specified, if no extension is given then the default extension *cellprj* is assumed.

The -b option allows to specify a different byte code file then the default one proposed by the compiler. The default proposed is *projectname.cellbyc*.

The output generated by the compiler can be redirected to a file with the -o option, if not present output is directed to *stdout.* In the next section we will have a detailed look at what output the compiler produces.

The -c option instructs the compiler to print a readable version of the bytecode produced. The -s option instructs the compiler to print the source code interspersed with the byte code generated. This can be helpful when debugging the code. The -c and -s option are mutually exclusive.

The -m option is to instruct the compiler to do a message check, namely to check whether each message sent in the source has a message handler somewhere in the packages. The check is not full proof, in the sense that the compiler cannot check if a message will arrive at the intended design - a message to a cell could have been injected in the cell tree at any point above the intended cell and still find its way to that cell - but at least it will detect orphan messages, i.e. message for which it cannot find any message handler, and orphan handlers, message handlers for which - in this source code - never a message is generated. None of these two conditions is an error, but it can help to spot obvious (typing) mistakes.

The -x option instructs the compiler to output the compiled code also in an XML format. The bytecode is normally output in a binary format, but the VM can also read the XML format when loading a file for execution. An example of the XML format is shown in the chapter *Running a CellSpeak program.*

14.4 Compiler output file

In this section we have a closer look at the output generated by the compiler.

Below are some extracts of the result of a compilation - not all the output is shown in this example. When no extra options are requested, the compiler will output just the files it is compiling, followed by the errors and warnings generated during the compilation. In the first part the compiler prints the name of the file it is currently compiling:

```
Compiling C:\Development\CellSpeak\Groups\Mesh.cellsrc
Compiling C:\Development\CellSpeak\Groups\Camera.cellsrc
Compiling C:\Development\CellSpeak\Groups\Light.cellsrc
Compiling C:\Development\CellSpeak\Groups\Shape.cellsrc
Compiling C:\Development\CellSpeak\Applications\CubeOfCubes.cellsrc
Compiling C:\Development\CellSpeak\Groups\Camera.cellsrc
Compiling C:\Development\CellSpeak\Applications\CubeOfCubes.cellsrc
```

It is not unusual to see the same file name appear several times. The first thing the compiler does when opening the file is reading the *use* clauses to see what groups the file needs for its compilation. If there are groups missing from that list, the compiler will start another file and work his way down the list until all files are compiled. If there are errors or warnings they will also be output. errors and warnings have the format as shown in the following examples. To find the file where the error occurred, one has to search the closest *Compiling <file>* above the error message:

```
Compiling C:\Development\CellSpeak\Applications\CubeOfCubes.cellsrc
-!- Error DCL 002 at line 59.Unknown token:  RotationMatrox
```

Error references have a name consisting of three characters, *DCL* for declaration in this case, followed by a number, *002* in this example. Then follows the line number in the source where the error occurred and a description of the error. In this case simply an unknown (misspelled) token, which is also given in the error message.

Some error messages can be more substantial, such as error messages for a syntax error, as the example below, where the compiler prints the token it has a problem with, together with the syntax rule(s) of the language showing where the parser had arrived - indicated by the star - and showing also what the parser expected to see at this point.

```
-!- Syntax Error at line 62.
    'for' (FOR) cannot be handled here. [Parser State 495].
    Rule 570 statement : BODY variable * capture_list IS END
    Rule 571 statement : BODY variable * IS code_block END
    Rule 574 statement : BODY variable * capture_list IS code_block END
    Rule 576 statement : BODY variable * IS END
```

Warnings have a similar format, but here the leading characters different and the word *error* is replaced by *warning.* A compilation with only warnings is still successful.

```
-=- Warning EXT 007 at line 25.The library type cannot be found in the
     library. Type name : CLSPK_ByteArray
-=- Warning EXT 013 at line 27.Note that external return types will be
     converted to the first equivalent type found.Type cell is also linked
     to type word32 - (new link to unsigned int)
```

After the compilation is over, the compiler will also show some compilation statistics. This is purely informational.

```
-- Compiler Timings ------------------------------------------------------
CellSpeakClass::CompileStream was called 21 times total duration 56 ms
ProjectClass::LoadAllPackages was called 1 times total duration 0 ms
ProjectClass::CompileEachFile was called 1 times total duration 65 ms
ProjectClass::AfterCompilation was called 1 times total duration 0 ms
```

The following line just confirms that the compiler has found all the designs that are used in the newly compiled bytecode. These designs might have been part of the source or part of the bytecode packages included in the project. If not all designs have been resolved, the compilation will still be successful, but when loading the package to run the missing designs will have to be part of some of the other packages already loaded or that are loaded at run time.

```
-- There are no unresolved designs in bytecode module CubeOfCubes
```

Then follows the readable output of the compiled files if that was requested. The simple byte code output has the following format:

```
---------------------------------------------------------------------------
-- ByteCode CubeOfCubes (1)
---------------------------------------------------------------------------
<INFO> Function sin unit 1 bytecode at: 0x08EDA088 size 52 bytes
<0000> FCT_DESC    unit 1 ip  0040 ip  0000
<0016> FLO_SIN     st  0024 st  0020
<0028> FCT_RETURN  st  0024 int 4
<0040> FCT_END     sin
---------------------------------------------------------------------------
<INFO> Function cos unit 2 bytecode at: 0x08EDA0BC size 52 bytes
<0000> FCT_DESC    unit 2 ip  0040 ip  0000
<0016> FLO_COS     st  0024 st  0020
<0028> FCT_RETURN  st  0024 int 4
<0040> FCT_END     cos
```

The number at the left hand side is the offset into the code unit (function, message handler, constructor) except for the first line of every unit which gives some general info about the unit: the name, the unit number, the address in the module and the size of the unit.

Each line in the unit represents a CellSpeak bytecode instruction followed by the arguments for that instruction. It would lead us too far to go into details about the bytecode instructions here, but there is an appendix to this manual where more details are given about the CellSpeak bytecode. The full explanation about the code is given in the manual *The design of the CellSpeak compiler and Virtual Machine.*

At the end of the file, the compiler will list the *exported designs* for this file, i.e. the designs that can be used by other packages that use this module

```
---------------------------------------------------------------------------
-- 21 Exported Designs
---------------------------------------------------------------------------
Design Cube.SharedMesh(fP) Unit  157
Design CubeOfCubes(iP) Unit  201
Design CubeWithAction(fP) Unit  195
Design Cylinder.SharedMesh(ffP) Unit  160
Design LightsForTheScene() Unit  192
Design Shape.Cone.SharedMesh(fP) Unit  163
Design Sphere.SharedMesh(fP) Unit  166
Design SystemGroup.ConnectionManagerDesign(C) Unit  113
Design SystemGroup.SystemDesign() Unit  103
Design SystemGroup.TimerDesign() Unit  123
Design camera.Basic() Unit  169
Design camera.FromTo(Rfff#Rfff#) Unit  172
Design light.Behaviour() Unit  142
Design light.Directional(Rffff#Rffff#Rffff#Rfff#) Unit  145
Design light.Point(Rffff#Rffff#Rffff#Rfff#ff) Unit  147
Design shape.SharedMesh(P) Unit  151
Design world() Unit  203
```

The exported designs list contains the name of the design, the parameter signature for its creation and the unit number in this module. The characters and explanation of the parameter signature are listed in an appendix.

15 Running a CellSpeak application

15.1 Introduction

Developing with CellSpeak is easy and follows the same path whether you are developing a small program to test some features or whether you are developing a big application.

CellSpeak is incremental in a way that many languages are not. If you have built some designs then another cell will be able to communicate with cells based on that design without having to compile with an external library or include some stuff. You just send the message(s) the cell responds to and you will get replies etc.

Also if a design is changed and the existing message interface is kept, you simply update the bytecode file in your system for that design and the rest of the system does not need to be touched. An architecture of micro-services is much easier to implement in CellSpeak then in any other language.

Load balancing with a system written in CellSpeak becomes also much easier - you can decide to run cells on different systems with no change to your source code.

Also if different protocols need to be used between systems, this can be dealt with in a CellSpeak program in a way that is transparent to the rest of the application(s).

The truth is off course that good and bad designs of a system are possible in all languages, but CellSpeak offers an approach to programming that recognizes the programming environment of today as compared to that of decades ago when most of the currently used programming languages were used.

15.2 What is an application in CellSpeak

When you run an application in CellSpeak, you ask the VM to add one or several cells to its database of running cells. What that cell, or cells, do then is up to the details of its design, but typically it will create other cells and start exchanging messages to get some work done. Note that the new cells will also be able to interact with the cells that are already running on that VM.

The user has also the choice to start up a new VM for running his application. That cell will still have the possibility to communicate with the other cells running in a different VM - the VM's will use some form of inter-process communication for that, transparent to the applications - but the applications will be more shielded from each other. Often this is the preferred way of running applications that cannot yet be fully trusted: if the application would crash the VM, the rest of the system will not be affected.

15.3 Starting up a CellSpeak application

The general format to start a CellSpeak application is as follows

```
cellrun -c -l -v -? -o <outputfile> -b <bytecodefile> -d <design parameters>
```

option	*explanation*
-c	creates a new virtual machine. Otherwise the command will be executed by a running VM.
-l	forces a load of the bytecode - even if it is already loaded in a running VM.

-t n	specifies that the virtual machine should use n threads for running the program. Setting the value to 0 or 1 makes it run single threaded. Not specifying it will schedule a default nr. of threads.
-i	forces the virtual machine to run as an interpreter. Normally bytecode is compiled to native code before execution.
-o	sets the optional output file for a new VM (existing VM keep their current output file)
-v	sets the verbose flag - more informational messages will be output to the output file
-b	is folllowed by the bytecode where the design to run can be found. Without extension .cellbyc is used.
-d	-d is followed by the cell to instantiate (if any) and the parameters: group.design param1 param2 param3 etc. parameters are not comma separated.
-?	prints this usage overview

remarks

- The -l option require the -b option to be present.
- Cellrun with only a -d option will create the cell in the running VM
- -t only has effect together with the -c option

Some examples:

```
cellrun -b CubeOfCubes -d World
```

In this example the running VM will load the bytecode file *CubeOfCubes.cellbyc* and create a cell based on the design *World*, which has no parameters.

```
cellrun -d PrintGroup.PrinterServer
```

The running VM will search a design called *PrintGroup.PrinterServer* in the bytecodes it has already loaded and if found create a new cell based on that design.

```
cellrun -c -t 4 -i -v -b Animals -d Hurd.Cows 20 white
```

This command will start of a new VM that will load the file *animals.cellbyc*, search the design *Hurd.Cows* and create a cell based on that design using the parameters *20* and “blue”, an integer and a string. The VM will start 4 threads to run the application but will not compile it to native code before running it - it will run in interpreted mode.

If a design or a file cannot be found *cellrun* will return an appropriate error message.

15.4 Files and dependencies

When loading an application, you have to specify which bytecode package contains the design(s) as for the cells you want to create.

A *bytecode package* is a file with the extension *.cellbyc*

A bytecode package can contain one or more *bytecode modules.* A bytecode module is the result of a compilation of a group of source files in a project. The developer has the possibility to store several modules together in one package. This might be because otherwise the overall system would contain too many small files, or because there is a hard link between the bytecode modules (see below).

Irrespective of the fact that modules are stored together or not, a bytecode module keeps a list of the other (compiled) modules that were required for its compilation. When the VM loads a bytecode package it keeps track of the modules in that package. The VM will check for each module what other module is required and in which package the module resides. Either it is a module that was already loaded or it was not yet loaded, and then the VM will search for the missing file and load that file as well. Also for this file it will check if all modules are present.

There are two types of dependencies between modules : there is a hard dependency and a soft dependency.

A soft dependency exists between modules if one or both modules create cells based on designs in the other module. The VM will only check that the design is still present in the module. Additionally a check can be made on the messaging interface - do the messages sent to the design still correspond to the message handlers of the design. However this last check is not 100% full proof because messages are not always sent directly to a cell but arrive at the cell after having been sent to the parent cell(s). So this can give false positives and false negatives, but it is a good help.

In any case what is not checked is the version of the module. A soft dependency will gladly work even if the version of the modules is different from what it was during development. The reason for this is off course that in this way it is easy to update a system, by simply changing the module where design has been updated. As long as the message profile remains the same, the other modules do not need to be changed or even rebuilt.

A hard dependency between modules exist when they share definitions (type definitions, function definitions). If the layout of a type changes then it is important that all modules that are using that type are being rebuild. The bytecode modules contain information whether they have a hard or a soft dependency on another module.

A reasonable approach when making packages from modules in an application, is to put modules that have a hard dependency together in the same package.

15.5 Where are my files

How does the VM locate the files it needs to run an application ?

The file below gives a typical example of a project file for a CellSpeak project. The files in the array *Source* are the source files of the project. The project is called *CubeOfCubes* and the resulting bytecode will be saved in a file called *cubeofcubes.cellbyc*.

```
--------------------------------------------------
--   Project file for the cube of cubes application
--------------------------------------------------

group Project.CubeOfCubes

-- The source files for the project

const string.ansi Source[] = [
     "..\\Applications\\CubeOfCubes.cellsrc",
     "..\\Groups\\Camera.cellsrc",
     "..\\Groups\\Light.cellsrc",
     "..\\Groups\\Shape.cellsrc",
]

-- the bytecodes for this project

const string.ansi ByteCode[] = [
     "Math.cellbyc"
]
```

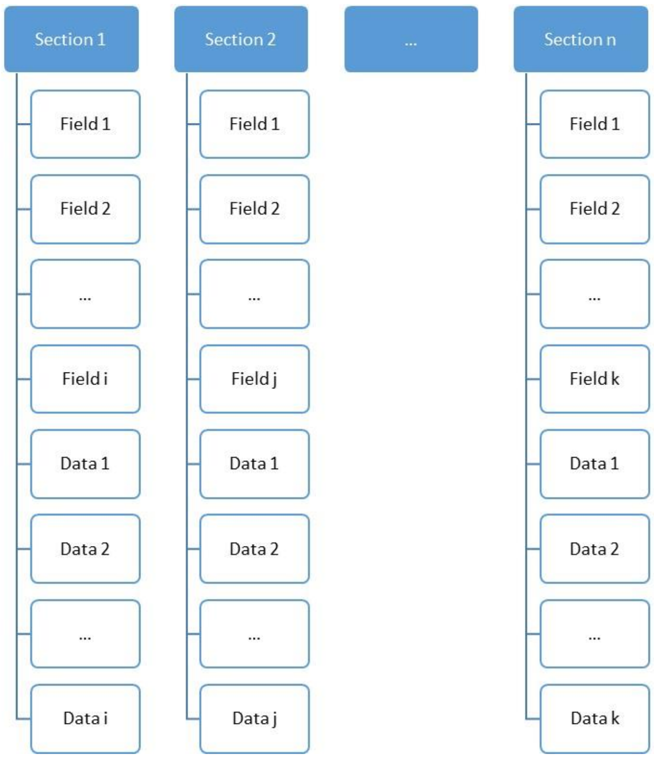
The files listed in the array *ByteCode* are external to the file *cubeofcubes.cellbyc* - unless the developer has used the compiler option to combine both into the same package - but let us assume that this is not the case here. The VM will use then the same path as given in the ByteCode array to locate the package file. This path however will now be relative to the path where the VM (cellrun) was started.

15.6 File formats

A byte code file can actually be saved in two distinct file formats. The first format is the binary *cellbyc* format and the second is the portable XML format.

The details of the *cellbyc* format are explained in the document *Design of the CellSpeak Compiler and Virtual Machine*, so here we limit ourselves to a short overview.

The bytecode package can contain several bytecode modules and for each module the bytecode file contains of a number of sections, and each section consists of the list of fields in the section followed by the actual data of the field. The fields have a fixed size, but the actual data can off course vary in size.



Each field has the same format and contains three elements : an identifier for the type of field, the offset in the file where the data for the field starts and the length of the data of the field.

The following sections are part of a module in a bytecode file.

Section	Content
Byte Code Section	A short section with the name, version and timestamp.
Byte code modules section	The section with all the bytecode modules used in the package
Exported Designs Section	This section contains an entry for each exported design.
Unresolved Designs Section	The list of designs used in the package but that are not part of the package and will have to found elsewhere.
Code Unit Section	This section contains an entry for each code unit in the module. There is a code unit for each function, method
Link table Section	information needed to write native code addresses where required
X Library Section	The list of external libraries needed in this module
End of Byte Code Section	End of module indicator

Except for the first and the last section, if there is nothing to save for a section then it is skipped.

The bytecode for a module is normally followed by the symbol table information for that module. Because symbol table information is stored in the bytecode module, one does not have to have access to the source of a module in order to be able to use the module for further development and use the types and designs in the module to work with. The developer however has always the possibility to leave out the symbol table information for those situations where it is not required or wanted.

The symbol table information for the module consists of the following sections

Section	Content
Symbol Table Section	Indicating the start of symbol table section
Name Space Section 1	There is a namespace section for each namespace in the module
Symbol Section 1..i	A variable number of fields per symbol section
Name Space Section 2	
Symbol Section 1..j	
…	
Name Space Section n	
Symbol Section 1..k	

End of Symbol Table Section	Indicating the end of a symbol table section

Each namespace section contains a section for each symbol in the symbol table, and these sections contain a field for each detail (name, type, etc.) that has to be saved for that symbol.

The bytecode file format is a binary format and when reading the binary file one must take into account the byte order used in the file.

The XML format is a text format, it is more verbose, but it has the advantage of being readable. To give a flavor of how the XML format looks, we have listed below a short example of an XML file output.

The file starts with some version information and then with the list of the exported designs.

```
<?xml version="1.0" encoding="utf-8"?>
<!--Created on : Mon Aug 03 13:51:36 2015-->
<ExportedDesigns Records="16">
    Cube.SharedMesh                    fP                  128
    CubeOfCubes                        iP                  173
    CubeWithAction                     fP                  167
    Cylinder.SharedMesh                ffP                 131
    LightsForTheScene                                      164
    Shape.Cone.SharedMesh              fP                  134
    Sphere.SharedMesh                  fP                  137
    camera.Basic                                           140
    camera.FromTo                      Rfff#Rfff#          143
    light.Behaviour                                        113
    light.Directional                  Rffff#Rffff#Rffff#Rfff# 116
    light.Point                        Rffff#Rffff#Rffff#Rfff#ff   118
    light.Spot                         Rffff#Rffff#Rffff#Rfff#fRfff#ff 120
    shape.SharedMesh                   P                   122
    world                                                  175
</ExportedDesigns>
```

The string of characters after the name of the design is the parameter signature for the creation of the design, followed by the bytecode identifier. The next entry in the file is the code unit table (codemap is the old name) with an entry for each code unit. The code unit contains the bytecode in a readable format

```
<CodeMapTable Records="180" TableBits="8" IndexBits="10" LastEntry="179">
    <CodeMap Name="sin" Map="1" Size="48">
        0000 FCT_DESC    0x24 0x0
        000C FLO_SIN     0x00000020 0x0000001C
        0018 FCT_RETURN  0x00000020 0x4
        0024 FCT_END     sin
    </CodeMap>
    <CodeMap Name="cos" Map="2" Size="48">
        0000 FCT_DESC    0x24 0x0
        000C FLO_COS     0x00000020 0x0000001C
        0018 FCT_RETURN  0x00000020 0x4
        0024 FCT_END     cos
    </CodeMap>
```

```
...
```

Also the symbol table information is stored in the XML file, a short extract of an example:

```
<namespace name="$" Symbols="6">
    <ArrayType>
        <name String="array[] of byte"/>
        <token Number="135"/>
        <Type String="byte"/>
        <indirect Hex="0x0"/>
        <Dimension Number="1"/>
        <Index Number="0"/>
    </ArrayType>
    <ArrayType>
        <name String="array[] of word16"/>
        <token Number="137"/>
        <Type String="word16"/>
        <indirect Hex="0x0"/>
        <Dimension Number="1"/>
        <Index Number="0"/>
    </ArrayType>
    <Design Name="world" Signature="" Map="175" Constructor="176">
        <MessageTable>
            <Message Name="FrameTick" Signature="i" Map="179"/>
        </MessageTable>
    </Design>
    <Design Name="Lights" Signature="" Map="164" Constructor="165">
    </Design>
    <Design Name="CubeWithAction" Signature="fP" Map="167" Constructor="0">
        <Ancestor String="Cube.SharedMesh"/>
        <MessageTable>
            <Message Name="Draw" Signature="P" Map="123"/>
            <Message Name="FrameTick" Signature="i" Map="172"/>
            <Message Name="Move" Signature="x" Map="125"/>
            <Message Name="MoveTo" Signature="x" Map="124"/>
            <Message Name="SetLog" Signature="P" Map="168"/>
            <Message Name="SetMaterial" Signature="z" Map="127"/>
            <Message Name="StartRotating" Signature="f" Map="169"/>
        </MessageTable>
    </Design>
```

The XML file is not meant to be tampered with, so for distribution the preferred format is the binary format.

15.7 Interaction with the VM

In a CellSpeak program it is possible to have direct interaction with the cells that make up a CellSpeak program, simply by sending messages to the cells.

Appendix 1 Backus-Naur format of CellSpeak

```
//----------------------------------------------------------------------------
// File     : CellSpeakGrammar.txt
// Copyright: 2000-2015 by Sam Verstraete
// Website  : CellSpeak.org
// License  : See website or License.txt
//
// This file contains the Backus-Naur form of the CellSpeak language.
// It consists of two parts $TERMINALS and $GRAMMAR.
//
// By convention terminals are written in capital letters and non-terminals in lower case.
// The $TERMINAL part contains a line per terminal specifying its string and priority.
// The priority is used to determine execution order in expressions and type-casts.
//
// The $GRAMMAR part contains the productions of the grammar in the following format:
//
// non-terminal :  rule_1 | rule_2 | .. | rule_n ;
//
// where rule1, rule2, ... rule_n are alternative productions for the non-terminal.
// A rule is made of a sequence of terminals and non-terminals.
// When at a certain point in a rule it is required to do something,
// an action is inserted in the rule between angled brackets : <action>
// Optional parts of a rule are written between curly brackets { and }
//
//----------------------------------------------------------------------------

$TERMINALS

PERIOD                     string = ".";
SEMI_COLON                 string = ";";
OPSQBR                     string = "[";
CLSQBR                     string = "]";
OPBR                       string = "(";
CLBR                       string = ")";
COMMA                      string = ",";
COLON                      string = ":";

// Type terminals start with TYPE_
// A type cast is treated like a unary operator with highest priority
// Not all terminals have a string image but they are needed in the compiler

TYPE                       string = "type";
TYPE_INT                   string = "int"        priority=8;
TYPE_INT16                 string = "int16"      priority=8;
TYPE_INT32                 string = "int32"      priority=8;
TYPE_INT64                 string = "int64"      priority=8;
TYPE_WORD                  string = "word"       priority=8;
TYPE_WORD16                string = "word16"     priority=8;
TYPE_WORD32                string = "word32"     priority=8;
TYPE_WORD64                string = "word64"     priority=8;
TYPE_BYTE                  string = "byte"       priority=8;
TYPE_CLASS                                       priority=8;
TYPE_PTRBOX                string = "ptr"        priority=8;
TYPE_FLOAT                 string = "float"      priority=8;
TYPE_DOUBLE                string = "double"     priority=8;
TYPE_RECORD                                      priority=8;
TYPE_ARRAY_MIA                                   priority=8;
TYPE_ARRAY_SIA                                   priority=8;
TYPE_ARRAY_BAR                                   priority=8;
TYPE_SLICE                                       priority=8;
TYPE_RENAMED                                     priority=8;
TYPE_LIST                  string = "list"       priority=8;
TYPE_EXPR_LIST                                   priority=8;
TYPE_VOID                  string = "void"       priority=8;
TYPE_FUNCTION                                    priority=8;
TYPE_ENUM                                        priority=8;
TYPE_BOOLEAN               string = "bool"       priority=8;
TYPE_CELL                  string = "cell"       priority=8;
TYPE_MESSAGE                                     priority=8;
```

```
TYPE_VEC_4X_BYTE           string = "vec4b"      priority=8;
TYPE_VEC_2X_INT            string = "vec2i"      priority=8;
TYPE_VEC_3X_INT            string = "vec3i"      priority=8;
TYPE_VEC_4X_INT            string = "vec4i"      priority=8;
TYPE_VEC_2X_FLOAT          string = "vec2f"      priority=8;
TYPE_VEC_3X_FLOAT          string = "vec3f"      priority=8;
TYPE_VEC_4X_FLOAT          string = "vec4f"      priority=8;
TYPE_VEC_2X_DOUBLE         string = "vec2d"      priority=8;
TYPE_VEC_3X_DOUBLE         string = "vec3d"      priority=8;
TYPE_VEC_4X_DOUBLE         string = "vec4d"      priority=8;
TYPE_MTX_2X2_FLOAT         string = "mtx2f"      priority=8;
TYPE_MTX_3X3_FLOAT         string = "mtx3f"      priority=8;
TYPE_MTX_4X4_FLOAT         string = "mtx4f"      priority=8;
TYPE_MTX_2X2_DOUBLE        string = "mtx2d"      priority=8;
TYPE_MTX_3X3_DOUBLE        string = "mtx3d"      priority=8;
TYPE_MTX_4X4_DOUBLE        string = "mtx4d"      priority=8;

// Other reserved words - in a few cases a trailing underscore is added to a terminal
// because there was a name clash with other definitions in c++ include files

_RECORD                    string = "record";
ELLIPSIS                   string = "...";
LIKE                       string = "like";
IS                         string = "is";
FROM                       string = "from";
CONSTANT                   string = "const";
_NULL                      string = "null";
ON                         string = "on";
DESIGN                     string = "design";
GROUP_                     string = "group";
CONSTRUCTOR                string = "constructor";
DESTRUCTOR                 string = "destructor";
LIB                        string = "lib";
NEW_                       string = "new";
DELETE_                    string = "delete";
CREATE                     string = "create";
DESTROY                    string = "destroy";
SIZEOF                     string = "sizeof";
SEL                        string = "sel";
NEL                        string = "nel";
LEL                        string = "lel";
VALID                      string = "valid";
YIELD                      string = "yield";
FLUSH                      string = "flush";
//FLOW                     string = "flow";
VAR                        string = "var";
ATTACH                     string = "attach";
DETACH                     string = "detach";
FUNCTION                   string = "function";
OPERATOR                   string = "operator";
RETURN                     string = "return";
SENDER                     string = "sender";
SELF                       string = "self";
THIS                       string = "this";
SAME                       string = "same";
ALL_CHILDREN               string = "all";
TRUE_CONSTANT              string = "true";
FALSE_CONSTANT             string = "false";
TEMPLATE                   string = "template";
USE                        string = "use";
RENAME                     string = "rename";
KEEP                       string = "keep";
PRIVATE                    string = "private";
PUBLIC                     string = "public";
ASSERT                     string = "assert";
SYSTEM                     string = "system";
INTERFACE                  string = "interface";
FORMAT                     string = "format";
BODY                       string = "body";
TASK                       string = "task";
```

```
// Reserved words used for statements

END                         string = "end";
IF                          string = "if";
THEN                        string = "then";
ELSE                        string = "else";
ELIF                        string = "elif";
CASE                        string = "case";
SWITCH                      string = "switch";
DEFAULT                     string = "default";
REPEAT                      string = "repeat";
UNTIL                       string = "until";
WHILE                       string = "while";
WITH                        string = "with";
DO                          string = "do";
FOR                         string = "for";
LOOP                        string = "loop";
TO                          string = "to";
CONTINUE                    string = "continue";
EACH                        string = "each";
IN_                         string = "in";
OUT_                        string = "out";
LEAVE                       string = "leave";
LAST                        string = "last";
CATCH                       string = "catch";
RAISE                       string = "raise";
SIGNAL                      string = "signal";
RETRY                       string = "retry";
CLASS                       string = "class";
EXTENDS                     string = "extends";
YOUR                        string = "your";
LAMBDA_ARROW                string = "=>";

// Operators

EQUALS                      string = "="      priority = 1;
DELAYED_EQUALS              string = "~"      priority = 1;
COPY_REF                    string = ":="     priority = 1;
DELAYED_COPY_REF            string = ":~"     priority = 1;
QMARK                       string = "?"      priority = 1;
XMARK                       string = "!"      priority = 1;
AND                         string = "and"    priority = 2;
OR                          string = "or"     priority = 2;
XOR                         string = "xor"    priority = 2;
IS_EQUAL                    string = "=="     priority = 3;
IS_NOT_EQUAL                string = "!="     priority = 3;
IS_BIGGER                   string = ">"      priority = 3;
IS_BIGGER_OR_EQUAL          string = ">="     priority = 3;
IS_LESS                     string = "<"      priority = 3;
IS_LESS_OR_EQUAL            string = "<="     priority = 3;
SHIFT_LEFT                  string = "<<"     priority = 4;
SHIFT_RIGHT                 string = ">>"     priority = 4;
PLUS                        string = "+"      priority = 5;
MINUS                       string = "-"      priority = 5;
STAR                        string = "*"      priority = 6;
SLASH                       string = "/"      priority = 6;
PERCENT                     string = "%"      priority = 6;
CROSS                       string = "#"      priority = 6;
AMPERSAND                   string = "&"      priority = 6;
ATSIGN                      string = "@"      priority = 6;
POWER                       string = "^"      priority = 7;
PIPE                        string = "|"      priority = 8;
UNARY_MINUS                 string = " -"     priority = 8; // special case
NOT                         string = "not"    priority = 8;
DOLLAR_SIGN                 string = "$";

// Tokens used by the compiler, but not used in the rules below
consumed;
EQUIVALENT_TYPE;
FIELD_NAME;
LIB_METHOD_NAME;
```

```
//-------------------------------------------------------------------------------
// The second part details the grammar of the language
//-------------------------------------------------------------------------------

$GRAMMAR

//-Compilation unit, groups, libraries------------------------------------------
//
// A compilation unit is a file that starts with a use list for that file and
// contains one or more groups of designs
//-------------------------------------------------------------------------------

compilation_unit
     : {use_list <CheckUseAndLoadList>} {group_content} END_OF_COMPILATION <EndOfCompilation>
     | UNKNOWN_NAME <IsThisASourceFile>
     ;

use_list
     : USE <ResetGroupPath> group_list {use_list}
     ;

group_list
     : group_path <AddGroupPathToUseList> { COMMA <ResetGroupPath> group_list }
     ;

group_path
     : PRIORITY_NAME <AddGroupNameToGroupPath> { PERIOD <SetPriorityContext> group_path }
     ;

group
     : GROUP_ <StartOfGroup> group_path <GroupName>
     | GROUP_ <StartOfGroup> group_path <GroupName> { IS LIB <SetPriorityContext>
library_file_name }
     ;

library_file_name
     : STRING_LITERAL <ExternalLibraryFile>
     | PRIORITY_NAME <ExternalLibraryFile>
     ;

group_content
     : group                           {group_content}
     | type_definition                 {group_content}
     | function_definition             {group_content}
     | template_use                    {group_content}
     | external_equivalence            {group_content}
     | constant_declaration            {group_content}
     | design_definition               {group_content}
     | operator_definition             {group_content}
     | full_constructor_definition     {group_content}
     | full_destructor_definition      {group_content}
     | full_message_definition         {group_content}
     | template_definition             {group_content}
     ;

group_prefix
     : GROUP_NAME_ PERIOD { group_prefix }
     ;

code_block
     : type_definition          {code_block}
     | function_definition      {code_block}
     | template_use             {code_block}
     | declarations             {code_block}
     | statement                {code_block}
     | exception_table
     ;


//-Templates------------------------------------------------------------------------------------------
```

```
template_definition
     : TEMPLATE <SetNewNameContext> NEW_NAME <NewTemplateName> { FOR <SetPriorityContext>
originals_list } IS <StartOfTemplate> TEMPLATE END <EndOfTemplate>
     ;

template_use
     : USE TEMPLATE full_template_name { FOR <SetLocalContext> substitute_list }  { WITH
<SetPriorityContext> template_flags } anything <TemplateWithSubstitutes>
     ;

originals_list
     : PRIORITY_NAME <TemplateOriginal> { COMMA <SetPriorityContext> originals_list }
     ;

substitute_list
     : substitute { COMMA <SetLocalContext> substitute_list }
     ;

substitute
     : LOCAL_CONTEXT_NAME <TemplateSubstitute> { PERIOD <TemplateAppendPeriod> substitute }
     | LOCAL_CONTEXT_NAME <TemplateSubstitute> OPSQBR CLSQBR <SubstituteBrackets>
     | NUMBER <TemplateSubstitute>
     | STRING_LITERAL <TemplateStringSubstitute>
     ;

full_template_name
     : {group_prefix} TEMPLATE_NAME <TemplateToUse>
     ;

template_flags
     : PRIORITY_NAME <SaveTemplateFlag> { COMMA <SetPriorityContext> template_flags }
     ;

//-Designs----------------------------------------------------------------------------------

design_definition
     : DESIGN <SetNewNameContext> design_header <DesignHeader> TO DO <DesignHeaderOnly>
     | DESIGN <SetNewNameContext> design_header <DesignHeader> {LIKE design_list} IS
<StartOfDesignContent> {design_content} END <EndOfDesignContent>
     ;

design_header
     : design_name
     | design_name OPBR {formal_design_parameter_list} CLBR
     ;

design_name
     : NEW_NAME <NameOfDesignToDefine>
     | { group_prefix } DESIGN_NAME <NameOfDesignToDefine>
     ;

design_list
     : design_to_inherit_from { COMMA design_to_inherit_from }
     ;

design_to_inherit_from
     : {group_prefix} DESIGN_NAME <DesignToInheritFrom> OPBR { inheritance_parameter_list }
CLBR <EndOfDesignInheritance>
     | {group_prefix} DESIGN_NAME <DesignToInheritFrom> anything <EndOfDesignInheritance>
     ;

inheritance_parameter_list
     : expression <CheckInheritanceParameter> { COMMA inheritance_parameter_list }
     ;

formal_design_parameter_list
     : type_spec <SetPriorityContext> PRIORITY_NAME <AddFormalDesignParameter> { COMMA
formal_design_parameter_list }
     ;
```

```
design_content
     : type_definition          {design_content}
     | keep_parameters          {design_content}
     | declarations             {design_content}
     | statement                {design_content}
     | constructor_definition   {design_content}
     | destructor_definition    {design_content}
     | interface                {design_content}
     | message_definition       {design_content}
     | function_definition      {design_content}
     | template_use             {design_content}
     | exception_table          {design_content}
     ;

constructor_definition
     : CONSTRUCTOR  IS <StartOfConstructorCode> {code_block} END <EndOfConstructorCode>
     | CONSTRUCTOR  TO DO <ConstructorAnnounced>
     ;

full_constructor_definition
     : full_structor_path PERIOD CONSTRUCTOR <StartOfOutOfDesignConstructorCode> IS
{code_block} END <EndOfOutOfDesignConstructorCode>
     ;

full_structor_path
     : { group_prefix } DESIGN_NAME <DesignInStructorName>
     ;

destructor_definition
     : DESTRUCTOR  IS <StartOfDestructorCode> {code_block} END <EndOfDestructorCode>
     | DESTRUCTOR  TO DO <DestructorAnnounced>
     ;

full_destructor_definition
     : full_structor_path PERIOD DESTRUCTOR <StartOfOutOfDesignDestructorCode> IS {code_block}
END <EndOfOutOfDesignDestructorCode>
     ;

keep_parameters
     : KEEP keep_parameter_list
     ;

keep_parameter_list
     : VARIABLE_NAME <KeepDesignParameter> { COMMA keep_parameter_list }
     ;

//-Type specification------------------------------------------------------------------------

type_definition
     : TYPE <SetPriorityContext> PRIORITY_NAME <NewUserName> IS type_spec
<EndOfTypeDefinition> {with}
     ;

type_spec
     : type_core {indirection} {index_list <CheckTypeIndexList>}
     ;

type_core
     : BASIC_TYPE <BasicType>
     | { group_prefix } named_type
     | _RECORD <StartOfRecord> {fields} END <EndOfRecordDefinition>
     | OPSQBR <StartOfEnumeration> enum_list CLSQBR <EndOfEnumeration>
     ;

named_type
     : TYPE_RENAMED <UserTypeName>
     | TYPE_RECORD <UserTypeName>
     | TYPE_ARRAY_MIA <UserTypeName>
     | TYPE_ARRAY_SIA <UserTypeName>
     | TYPE_ARRAY_BAR <UserTypeName>
     | TYPE_FUNCTION <UserTypeName>
     | TYPE_CLASS <ClassType>
     | TYPE_ENUM <UserTypeName>
     ;

index_list
     : OPSQBR <StartOfIndices> {indices} CLSQBR <EndOfIndices> {index_list}
     ;

indices
     : index { COMMA <CountIndex> indices }
     | COMMA <CountIndex> { indices }
     ;

index
     : expression <Index>
     | {group_prefix} named_type <EnumeratedIndex>
     ;

indirection
     : PERIOD TYPE_PTRBOX <SetPointerToRecord>
     ;

with
     : WITH <StartOfWith> {fields} END <EndOfWith>
     ;

fields
     : field_spec  {fields}
     ;

field_spec
     : type_spec <SetLocalContext> LOCAL_CONTEXT_NAME <FieldName> {index_list
<CheckFieldIndexLists>} {field_initialisation}
     | UNKNOWN_NAME <UntypedFieldName> field_initialisation
     | VARIABLE_NAME <UntypedFieldName> field_initialisation
     | RENAME <SetLocalContext> LOCAL_CONTEXT_NAME <CheckFieldName> TO <SetLocalContext>
LOCAL_CONTEXT_NAME <ChangeFieldName>
     | KEEP keep_field_list
     | { VAR <SetVar> } function_definition
     | constant_declaration
     | type_definition
     ;

keep_field_list
     : VARIABLE_NAME <KeepVariableAsField> { COMMA keep_field_list }
     ;

field_initialisation
     : EQUALS <StartOfFieldInitialisation> expression <EndOfFieldInitialisation>
     | COPY_REF <StartOfFieldInitialisation> expression <EndOfFieldInitialisation>
     ;

enum_list
     : PRIORITY_NAME <Enumerator> { COMMA enum_list }
     ;

//-Declarations------------------------------------------------------------------------------

declarations
     : constant_declaration
     | variable_declaration
     ;

constant_declaration
     : CONSTANT <SetConstant> type_spec <SetNewNameContext> new_constant_list
<ClearDeclaration>
     | CONSTANT <SetConstant> new_constant_list <ClearDeclaration>
     ;

new_constant_list
```

```
     : NEW_NAME <AddConstantToSymbolTable> {index_list <CheckConstantIndexLists>}
new_constant_value { COMMA new_constant_list }
     ;

new_constant_value
     : EQUALS <EqualsForConstant> expression <ConstantAssignment>
     | IS <EqualsForConstant> _RECORD <StartOfRecordDeclaration> fields END
<EndOfRecordDeclaration>
     ;

variable_declaration
     : type_spec <SetNewNameContext> new_variable_list <ClearDeclaration>
     | VAR <SetVar> new_variable_list <ClearDeclaration>
     | VAR <SetVar> function_definition
     ;

new_variable_list
     : new_variable COMMA <AddToLHSList> {type_spec <SetNewNameContext>} new_variable_list
     | new_variable anything <CheckVarDeclarations>
     ;

new_variable
     : NEW_NAME <AddVariableToSymbolTable> {index_list <CheckVariableIndexLists> }
{variable_initialisation}
     | NEW_NAME <AddVariableToSymbolTable> LIKE variable_example <CopyVariableType>
     | VARIABLE_NAME <VariableRedefinition> {index_list <CheckVariableIndexLists> }
{variable_initialisation}
     ;

variable_initialisation
     : EQUALS <StartOfInitialisation> expression <EndOfInitialisation>
     | COPY_REF <StartOfInitialisation> expression <EndOfInitialisation>
     | IS <StartOfInitialisation> _RECORD <StartOfRecordDeclaration> fields END
<EndOfRecordDeclaration>
     ;

variable_example
     : VARIABLE_NAME
     ;

//-External Type Equivalence-----------------------------------------------------------------

external_equivalence
     : LIB TYPE <SetPriorityContext> type_equivalence_list
     | LIB CLASS <SetPriorityContext> class_equivalence_list
     | LIB FUNCTION <SetPriorityContext> function_equivalence_list
     ;

type_equivalence_list
     : type_equivalence { COMMA <SetPriorityContext> type_equivalence_list }
     ;

type_equivalence
     : library_type_name IS type_spec <EndOfTypeEquivalence>
     | library_type_name IS TYPE_VOID <EndOfTypeEquivalence>
     | library_type_name STAR IS <SetNewNameContext> NEW_NAME <EndOfPointerEquivalence>
     | library_type_name anything <EndOfSameNameTypeEquivalence>
     ;

library_type_name
     : PRIORITY_NAME <LibraryTypeName>
     | BASIC_TYPE <LibraryTypeName>
     | TYPE_VOID <LibraryTypeName>
     | STRING_LITERAL <LibraryTypeName>
     ;

class_equivalence_list
     : class_equivalence { COMMA <SetPriorityContext> class_equivalence_list }
     ;

class_equivalence
```

```
     : library_type_name <CopyTypeToExtend> {class_extension} IS <SetNewNameContext> NEW_NAME
<EndOfClassEquivalence>
     | library_type_name <CopyTypeToExtend> {class_extension} anything
<EndOfSameNameClassEquivalence>
     ;

class_extension
     : EXTENDS <SetPriorityContext> library_type_name <CppClassInheritance>
     ;

function_equivalence_list
     : function_equivalence { COMMA <SetPriorityContext> function_equivalence_list}
     ;

function_equivalence
     : library_type_name IS <SetNewNameContext> NEW_NAME <EndOfFunctionEquivalence>
     | library_type_name anything <EndOfSameNameFunctionEquivalence>
     ;

//-Messages---------------------------------------------------------------------------------

interface
     : INTERFACE <NewInterface> { interface_name } IS <CheckInterfacePrefix>
     | INTERFACE <NewInterface> END <CancelInterfacePrefix>
     ;

interface_name
     : PRIORITY_NAME <InterfaceNameSegment> { PERIOD <SetPriorityContext> interface_name }
     ;

message_definition
     : ON <SetPriorityContext> new_message_name <NewMessageName> { message_header } DO
<StartOfMessageCode> {keep_message_parameters} {code_block} END <EndOfMessageCode>
     | ON <SetPriorityContext> new_message_name <NewMessageName> { message_header }
LAMBDA_ARROW <StartOfMessageLambda> expression <EndOfMessageLambda>
     | ON <SetPriorityContext> new_message_name <NewMessageName> { message_header }
LAMBDA_ARROW <StartOfMessageLambda> keep_message_parameters <EndOfMessageLambda>
     | ON <SetPriorityContext> new_message_name <NewMessageName> { message_header } TO DO
<MessageHeaderOnly>
     ;

new_message_name
     : PRIORITY_NAME <AddToMessageName>
     | STRING_LITERAL <AddToMessageName>
     | PRIORITY_NAME <AddToMessageName> PERIOD new_message_name
     | STRING_LITERAL <AddToMessageName> PERIOD new_message_name
     | QMARK <DefaultHandler>
     ;

message_header
     : OPBR <StartOfFormalMessageParameters> formal_message_parameters CLBR
     | OPBR <StartOfFormalMessageParameters> CLBR
     | LIKE message_to_get_header_from <InheritMessageHeader>
     ;

formal_message_parameters
     : type_spec <SetPriorityContext> PRIORITY_NAME <AddFormalMessageParameter>  { COMMA
formal_message_parameters }
     | VAR <SetPriorityContext> PRIORITY_NAME <FormalMessageParameterNoType> LIKE
parameter_example <CopyMessageParameterType>
     ;

full_message_definition
     : ON full_message_name message_header <CheckIfDefinedAndNoCode> DO <StartOfMessageCode>
{code_block} END <EndOfFullMessageCode>
     ;

full_message_name
     : GROUP_NAME_ PERIOD full_message_name
     | DESIGN_NAME <DesignInMessageName> PERIOD <SetPriorityContext> new_message_name
<KnownMessageName>
```

```
     ;

keep_message_parameters
     : KEEP message_parameters_to_keep
     ;

message_parameters_to_keep
     : VARIABLE_NAME <KeepMessageParameter> COMMA message_parameters_to_keep
     | VARIABLE_NAME <KeepMessageParameter>
     ;

//-Statements--------------------------------------------------------------------------------

statement
     : for_statement
     | loop_statement
     | while_statement
     | repeat_statement
     | if_statement
     | switch_statement
     | continue_statement
     | leave_statement
     | return_statement
     | raise_statement
     | compound_statement
     | bytecode_instruction
     | expression anything <DropExpression>
     | multiple_lhs_expression anything
     | task_header
     | body_specification
     ;

for_statement
     : FOR <StartOfForStatement> loop_ranges <LoopRange> lambda_or_code_block
<EndOfDoForRanges>
     | FOR <StartOfForStatement> EACH loop_variable IN_ variable <ArrayListable>
lambda_or_code_block <EndOfDoForArray>
     | FOR <StartOfForStatement> EACH loop_variable IN_ named_type <EnumerationListable>
lambda_or_code_block <EndOfDoForEnumeration>
     | FOR <StartOfForStatement> EACH loop_variable IN_ OPSQBR <StartOfExpressionList>
expression_list CLSQBR <ExpressionListable> lambda_or_code_block <EndOfDoForList>
     | FOR <StartOfForStatement> EACH OPSQBR <StartOfIndexVariables> index_variables CLSQBR
<EndOfIndexVariables> IN_ variable <ArrayIndexed> lambda_or_code_block
<EndOfDoForArrayIndexed>
     | FOR <StartOfForStatement> OPBR {loop_start} COMMA <LoopExpressionFollows> expression
COMMA <UpdatesFollow> {updates} <StartOfLoop> CLBR lambda_or_code_block <EndOfLoop>
     ;

lambda_or_code_block
     : LAMBDA_ARROW expression <DropExpression>
     | DO {code_block} END
     ;

loop_ranges
     : loop_variable EQUALS <CheckCountableLoopVariable> expression TO <StartOfTo> expression
{ COMMA <NextLoopRange> loop_ranges }
     ;

loop_variable
     : VARIABLE_NAME <LoopVariable>
     | NEW_NAME <LoopVariable>
     ;

index_variables
     : VARIABLE_NAME <IndexVariable>
     | NEW_NAME <IndexVariable>
     | VARIABLE_NAME <IndexVariable> COMMA index_variables
     | NEW_NAME <IndexVariable> COMMA index_variables
     ;

loop_start
```

```
     : expression <DropExpression> {loop_start}
     | NEW_NAME <LoopVariable> EQUALS expression {loop_start}
     ;

updates
     : expression {updates}
     ;

while_statement
     : WHILE <StartOfWhileStatement> expression <StartOfDoInWhileDo> lambda_or_code_block
<EndOfDoInWhileDo>
     ;

repeat_statement
     : REPEAT <StartOfRepeatStatement> {code_block} UNTIL expression <EndOfRepeatUntil>
     ;

if_statement
     : IF <StartOfIfStatement> expression THEN <StartOfThen> {code_block} {elif_part} ELSE
<StartOfElse> {code_block} END <EndOfElse>
     | IF <StartOfIfStatement> expression THEN <StartOfThen> {code_block} {elif_part} END
<NoElse>
     | IF <StartOfIfStatement> expression LAMBDA_ARROW <StartOfThen> expression
<DropExpression>
     ;

elif_part
     : ELIF <StartOfElif> expression THEN <StartOfThen> {code_block} {elif_part}
     ;

switch_statement
     : SWITCH <StartOfSwitchStatement> expression <StartOfCaseList> {case_list} END
<EndOfSwitchStatement>
     ;

case_list
     : CASE simple_constant_list <SimpleConstantList> COLON {code_block} <EndOfCase> {
case_list }
     | DEFAULT <DefaultCase> COLON {code_block} <EndOfDefaultCase>
     ;

simple_constant_list
     : simple_constant { COMMA simple_constant_list }
     ;

continue_statement
     : CONTINUE TASK_NAME <ContinueStatement>
     | CONTINUE anything <ContinueStatement>
     ;

leave_statement
     : LEAVE  TASK_NAME <LeaveStatement>
     | LEAVE anything <LeaveStatement>
     ;

return_statement
     : RETURN <CheckForReturnNothing> return_values <CheckReturnStatement>
     | RETURN <CheckForReturnNothing> anything <CheckFunctionReturnsNothing>
     ;

return_values
     : expression <CheckReturnValue> {COMMA return_values}
     ;

compound_statement
     : DO <StartOfCompoundStatement> {code_block} END <EndOfCompoundStatement>
     ;

task_header
     : TASK <SetNewNameContext> NEW_NAME <TaskName>
     ;
```

```
body_specification
     : {NEW_ <PrepareAllocation>} BODY variable <BodyVariable> {capture_list} IS
<StartOfBodyCode> {code_block} END <EndOfBodyCode>
     | {NEW_ <PrepareAllocation>} BODY variable <BodyVariable> {capture_list} LAMBDA_ARROW
<StartOfVariableLambda> expression <EndOfVariableLambda>
     ;

//-Expressions-------------------------------------------------------------------------

expression
     : variable ref_assign_sign <StartOfReferenceAssignment> expression <ReferenceAssignment>
     | variable assign_sign expression <Assignment>
     | simple_part send_operator message_list <SendMessagesToCell>
     | simple_part QMARK <QMarkInExpression> qmark_selection
     | simple_part infix_operator assign_sign expression <OperatorAssignment>
     | simple_part infix_operator expression
     | prefix_operator expression
     | cast expression
     | simple_part
     ;

simple_part
     : variable
     | simple_constant
     | directive
     | OPBR <OpeningBracketInExpression> expression CLBR <ClosingBracketInExpression>
{variable_continuation}
     | OPSQBR <StartOfExpressionList> expression_list CLSQBR <EndOfExpressionList>
     | UNKNOWN_NAME <ReportUnknownIdentifier>
     ;

expression_list
     : expression <ElementOfExpressionList> { COMMA  expression_list }
     ;

cast
     : IS_LESS <SaveCurrentType> type_spec <TypeCast> { STAR <SetCppPtr> } IS_BIGGER
     | IS_LESS <SaveCurrentType> TYPE_VOID <TypeCast> { STAR <SetCppPtr> } IS_BIGGER
     ;

infix_operator
     : PLUS                <OperatorInExpression>
     | MINUS               <OperatorInExpression>
     | STAR                <OperatorInExpression>
     | SLASH               <OperatorInExpression>
     | PERCENT             <OperatorInExpression>
     | POWER               <OperatorInExpression>
     | CROSS               <OperatorInExpression>
     | AND                 <OperatorInExpression>
     | OR                  <OperatorInExpression>
     | XOR                 <OperatorInExpression>
     | IS_EQUAL            <OperatorInExpression>
     | IS                  <IsInExpression>
     | IS_NOT_EQUAL        <OperatorInExpression>
     | IS_BIGGER           <OperatorInExpression>
     | IS_BIGGER_OR_EQUAL  <OperatorInExpression>
     | IS_LESS             <OperatorInExpression>
     | IS_LESS_OR_EQUAL    <OperatorInExpression>
     | SHIFT_LEFT          <OperatorInExpression>
     | SHIFT_RIGHT         <OperatorInExpression>
     ;

ref_assign_sign
     : COPY_REF    <RefAssignmentSign>
     | DELAYED_COPY_REF <RefAssignmentSign>
     ;

assign_sign
     : EQUALS <AssignmentSign>
     | DELAYED_EQUALS <AssignmentSign>
     ;

send_operator
     : IS_LESS <LeftArrow> { send_modifier } MINUS <SendOperator>
     ;

send_modifier
     : STAR  <SendModifier> { send_modifier }
     | PLUS  <SendModifier> { send_modifier }
     | XMARK <SendModifier> { send_modifier }
     ;

prefix_operator
     : PLUS                 <UnaryPlus>
     | MINUS                <UnaryMinus>
     | NOT                  <Not>
     | AMPERSAND            <Ampersand>
     ;

variable
     : VARIABLE_NAME <VariableName> { variable_continuation }
     | const_function { variable_continuation }
     | THIS <This> { variable_continuation }
     | SELF <Myself>
     | SELF <Myself> PERIOD VARIABLE_NAME <MyVariableName>
     | SENDER <Sender>
     | SYSTEM <System>
     | SAME <SameMessage>
     | ALL_CHILDREN <AllChildren>
     | CREATE <SetPriorityContext> { PRIVATE <PrivateCell> } design_instance
     | SEL <SetAllocationContext> variable <SetArraySize>
     | NEW_ <PrepareAllocation> { OUT_ <AllocateForOut> } { allocation_cast } variable
<AllocateVariable>
     ;

allocation_cast
     : IS_LESS type_spec <AllocationCast> IS_BIGGER
     ;

multiple_lhs_expression
     : lhs_list EQUALS <StartOfExpressionList> expression_list <MultiAssignment>
     | lhs_list COPY_REF <StartOfMultiReferenceAssignment> expression_list
<MultiReferenceAssignment>
     | lhs_list send_operator message_list <SendMessagesToCellList>
     ;

lhs_list
     : variable COMMA <AddFirstVariableToList> tail_of_lhs_list
     ;

tail_of_lhs_list
     : variable <AddNextVariableToList> { COMMA tail_of_lhs_list }
     ;

variable_continuation
     : field_selection { variable_continuation }
     | array_selection { variable_continuation }
     | array_slice
     | function_parameters { variable_continuation }
     ;

field_selection
     : PERIOD <SetLocalContext> LOCAL_CONTEXT_NAME <FieldOrMethodSelection>
     ;

array_selection
     : OPSQBR <StartOfIndexSelectorOrRange> index_selector CLSQBR <EndOfIndexSelector>
     ;

array_slice
     : OPSQBR <StartOfIndexSelectorOrRange> index_range CLSQBR <EndOfIndexRanges>
     ;

function_parameters
     : OPBR <StartOfActualFunctionParameters> {actual_function_parameters} CLBR
<EndOfActualFunctionParameters>
     | OPBR <StartOfActualFunctionParameters> formal_function_parameters CLBR
<FunctionReference>
     ;

const_function
     : const_function_name function_parameters
     | const_function_name anything <FunctionReference>
     ;

const_function_name
     : {group_prefix} CONST_FUNCTION_NAME <CallConstantFunctionName>
     | {group_prefix} DESIGN_NAME <PredecessorInFunctionCall> PERIOD CONST_FUNCTION_NAME
<PredecessorFunctionName>
     | {group_prefix} LIB_FUNCTION_NAME <CallLibFunctionName>
     ;

actual_function_parameters
     : expression <ActualFunctionParameter> { COMMA actual_function_parameters }
     ;

index_selector
     : expression <ElementOfIndexSelector> { COMMA index_selector }
     ;

index_range
     : {expression <ElementOfIndexSelector>} COLON <FromInIndexRange> {expression} anything
<ToInIndexRange>
     | {expression <ElementOfIndexSelector>} COLON <FromInIndexRange> {expression} COMMA
<ToInIndexRange> index_range
     ;

simple_constant
     : NUMBER              <Number>
    | STRING_LITERAL       <StringLiteral>
     | TRUE_CONSTANT       <BooleanConstant>
     | FALSE_CONSTANT      <BooleanConstant>
     | named_constant
     | _NULL               <NullConstant>
    ;

named_constant
     : {group_prefix} CONSTANT_NAME <ConstantInExpression> { variable_continuation }
     | TYPE_ENUM <EnumTypeInConstant> PERIOD <SetPriorityContext> PRIORITY_NAME
<EnumeratedConstant>
     ;

qmark_selection
     : expression_sequence COLON <ColonInExpression> expression_sequence <SelectExpression>
     | expression_sequence anything <OnlyTrueExpression>
     ;

expression_sequence
     : expression_item
     | expression_item { AMPERSAND expression_sequence }
     ;

expression_item
     : expression
     | return_statement <PushBackReturn>
     | leave_statement
     | continue_statement
     | raise_statement
     ;

directive
     : FLUSH <FlushMessages>
     | YIELD <YieldMessage>
     | ATTACH variable <AttachCell>
     | DETACH variable <DetachCell>
     | DESTROY variable <DestroyCell>
     | SIZEOF variable <DetermineVariableSize>
     | SIZEOF type_core <DetermineTypeSize>
     | NEL named_type <NumberOfElementsInType>
     | NEL variable <NumberOfElementsInVariable>
     | NEL OPSQBR expression <IndexNumber> CLSQBR variable <NumberOfElementsForIndex>
     | LEL variable <LastElementInVariable>
     | LEL OPSQBR expression <IndexNumber> CLSQBR variable <LastElementForIndex>
     | DELETE_ variable <DeleteVariable>
     | ASSERT assertion
     | FORMAT STRING_LITERAL <FormatString> OPSQBR <StartOfExpressionList>
format_expression_list CLSQBR <EndOfFormatDirective>
     | PRIVATE <SetPermission>
     | PUBLIC <SetPermission>
     ;

assertion
     : expression { COMMA STRING_LITERAL }
     ;

format_expression_list
     : expression <ElementOfFormatExpressionList> { COMMA  format_expression_list }
     ;

design_instance
     : design_path <DesignPathToInstantiate> anything <EndOfActualDesignParameters>
     | design_path <DesignPathToInstantiate> OPBR { actual_design_parameters } CLBR
<EndOfActualDesignParameters>
     ;

design_path
     : PRIORITY_NAME <AddToDesignPath> { PERIOD <CheckDesignPath> design_path }
     ;

actual_design_parameters
     : expression <ActualDesignParameter> { COMMA actual_design_parameters }
     ;

message_list
     : message <MessageInList> { COMMA message_list }
     ;

message
     : message_name <NameOfMessageToSend>
     | message_name <NameOfMessageToSend>  OPBR <StartOfActualMessageParameters> {
actual_message_parameters} CLBR <EndOfActualMessageParameters>
     ;

message_name
     : message_name_segment { PERIOD message_name }
     | OPBR <ClearPriorityContext> expression CLBR <ExpressionInMessageName>
     ;

message_name_segment
     : PRIORITY_NAME <AddToMessageNameToSend>
     | STRING_LITERAL <AddToMessageNameToSend>
     | FORMAT <ClearPriorityContext> STRING_LITERAL <FormatString> OPSQBR
<StartOfExpressionList> format_expression_list CLSQBR <EndOfFormatDirective>
     ;

actual_message_parameters
     : expression <ActualMessageParameter> { COMMA  actual_message_parameters }
     ;

//-Functions --------------------------------------------------------------------------

function_definition
```

```
     : FUNCTION <SetFunctionContext> function_name {function_signature}
function_implement_or_delay
     ;

function_implement_or_delay
     : {capture_list} IS <StartOfFunctionCode> {code_block} END <EndOfFunctionCode>
     | {capture_list} LAMBDA_ARROW <StartOfLambda> expression <EndOfLambda>
     | TO DO <FunctionSignatureOnly>
     ;

function_name
     : GROUP_NAME_ PERIOD function_name
     | NEW_NAME    <NewFunctionName>
     | CONST_FUNCTION_NAME <NewFunctionName>
     | VARIABLE_NAME <NewFunctionName>
     | LOCAL_CONTEXT_NAME <NewFunctionNameForField>
     | DESIGN_NAME <DesignName> PERIOD CONST_FUNCTION_NAME <KnownFunctionName>
     | DESIGN_NAME <DesignName> PERIOD type_with_functions PERIOD <SetLocalContext>
LOCAL_CONTEXT_NAME <KnownFunctionNameForField>
     | type_with_functions PERIOD <SetLocalContext> LOCAL_CONTEXT_NAME
<KnownFunctionNameForField>
     ;

type_with_functions
     : TYPE_RENAMED    <TypeWithFunctions>
     | TYPE_RECORD     <TypeWithFunctions>
     | TYPE_ARRAY_MIA  <TypeWithFunctions>
     | TYPE_ARRAY_SIA  <TypeWithFunctions>
     | TYPE_ARRAY_BAR  <TypeWithFunctions>
     | TYPE_ENUM       <TypeWithFunctions>
     | BASIC_TYPE      <TypeWithFunctions>
     ;

capture_list
     : KEEP <StartOfCapture> capture_variable_list
     | WITH <StartOfCapture> {fields} END
     ;

capture_variable_list
     : VARIABLE_NAME <KeepVariableAsField> { COMMA capture_variable_list }
     ;

function_signature
     : OPBR { formal_function_parameters } CLBR {return_spec}
     | {return_spec}
     | LIKE function_example <CopyFunctionType>
     ;

function_example
     : CONST_FUNCTION_NAME
     | VARIABLE_NAME
     | FIELD_NAME
     ;

formal_function_parameters
     : formal_function_parameter <AddFormalFunctionParameter> {COMMA
formal_function_parameters}
     ;

formal_function_parameter
     : {parameter_qualifier} type_spec <SetPriorityContext> { PRIORITY_NAME <NewUserName> }
{index_list <CheckParameterIndexLists>}
     | {parameter_qualifier} FUNCTION <SetPriorityContext> PRIORITY_NAME
<NameOfFunctionAsParameter> {function_signature} <EndOfFunctionAsParameter>
     | {parameter_qualifier} VAR <SetPriorityContext> { PRIORITY_NAME <NewUserName> } LIKE
parameter_example <CopyParameterType>
     ;

parameter_qualifier
     : IN_    <SetInParameter>
     | OUT_   <SetOutParameter>
     ;

return_spec
     : OUT_ return_type_list
     | OUT_ OPBR { formal_return_list } CLBR <EndOfFormalReturnList>
     ;

return_type_list
     : type_spec <AddTypeToReturnList> {COMMA return_type_list}
     ;

formal_return_list
     : type_spec <AddTypeToReturnListAndSetContext> { PRIORITY_NAME <AddNameToReturnList>}  {
COMMA <ClearDeclaration> formal_return_list }
     | VAR <SetPriorityContext> PRIORITY_NAME LIKE parameter_example { COMMA
<ClearDeclaration> formal_return_list }
     ;

parameter_example
     : VARIABLE_NAME
     ;

// the following rules allow to inherit headers -------------------------------------------

message_to_get_header_from
     : {group_prefix} DESIGN_NAME <DesignName> PERIOD MESSAGE_NAME <KnownMessageName>
     | MESSAGE_NAME <KnownMessageName>
     ;

design_to_get_header_from
     : {group_prefix} DESIGN_NAME <DesignName>
     ;

// Operators ----------------------------------------------------------------------------

operator_definition
     : OPERATOR user_operator <UserOperator> operator_signature IS <StartOfUserOperatorCode>
{code_block} END <EndOfFunctionCode>
     | OPERATOR user_operator <UserOperator> operator_signature anything
<UserOperatorSignatureOnly>
     ;

user_operator
     : infix_operator
     | EQUALS
     | IS_LESS type_spec <TypeCastOperator> IS_BIGGER
     ;

operator_signature
     : OPBR formal_function_parameters CLBR {return_spec}
     ;

//-Exception Handler-------------------------------------------------------------------------

exception_table
     : CATCH <StartOfExceptionTable> {exception_list} END <EndOfExceptionTable>
     ;

exception_list
     : PRIORITY_NAME <NewExceptionName> { formal_exception_parameters } exception_handler  {
exception_list }
     | DEFAULT <NewExceptionName> { formal_exception_parameters } exception_handler
     ;

exception_handler
     : DO <StartOfExceptionCode> {code_block} END <EndOfExceptionCode>
     | LAMBDA_ARROW <StartOfExceptionCode> expression <EndOfExceptionCode> { AMPERSAND
<ClearPriorityContext> return_statement <PushBackReturnAndSetContext> }
     ;

formal_exception_parameters
```

```
     : OPBR <StartOfFormalExceptionParameters> { formal_exception_parameter_list } CLBR
<EndOfFormalExceptionParameters>
     ;

formal_exception_parameter_list
     : type_spec <SetNewNameContext> NEW_NAME <FormalExceptionParameter> { COMMA
formal_exception_parameter_list }
     ;

raise_statement
     : RAISE <SetPriorityContext> PRIORITY_NAME <ExceptionNameToRaise> OPBR
<StartOfActualExceptionParameters> { actual_exception_parameters } CLBR
<EndOfActualExceptionParameters>
     | RAISE <SetPriorityContext> PRIORITY_NAME <ExceptionNameToRaise> anything
<EndOfActualExceptionParameters>
     ;

actual_exception_parameters
     : expression <ActualExceptionParameter> { COMMA  actual_exception_parameters }
     ;

//-ByteCode assembly---------------------------------------------------------------------
--------------------

bytecode_instruction
     : DOLLAR_SIGN <SetInstructionExpected> LOCAL_CONTEXT_NAME <CheckInstruction>
{operand_list} <AddInstructionToCode> anything <EndOfByteCodeInstruction>
     ;

operand_list
     : variable <CheckOperand> {operand_list}
     | special_variable {operand_list}
     | simple_constant <CheckOperand> {operand_list}
     | SIZEOF type_core <SizeOperand> {operand_list}
     ;

special_variable
     : PERCENT <SetLocalContext> LOCAL_CONTEXT_NAME <SpecialOperand> { displacement }
     ;

displacement
     : PLUS  <OperandOperator> simple_constant <AdjustOperand>
     | MINUS <OperandOperator> simple_constant <AdjustOperand>
     ;
```