Desi: Erasmus for 2010

Peter Grogono and Brian Shearing

November 2011

Contents

Revision History

Date	Revised by	Description
22 June 2010	P.G.	First complete draft.
30 July 2010	P.G.	Revisions based on compiler implementation.
29 April 2011	P.G.	Added shift operators and a few comments.
24 August 2011	P.G.	Modified to reflect recent discussions.
02 November 2011	P.G.	Simplified protocols and removed **such that**.

Desi: Erasmus for 2010

1. Introduction

During the last few months (i.e., early 2010), we have produced several discussion papers on the future of Erasmus—notably, KIWIs 090923, 100116, 100125, and 100131. This report describes a new version of Erasmus that is built on earlier versions, these discussions, and a few other recent ideas.

This project is intended to be significant, hopefully leading to a new implementation, and it deserves a name. Names of the form "Erasmus 2" or "Erasmus II" are probably premature, since only a dozen or so people are familiar with the first version. We propose the name Desi (as in "Desiderius Erasmus") as the working title this project and the language described in this document, but the ultimate name of the language that it produces—if any—will still be Erasmus.

In this report, mE denotes the current implementation, a.k.a. miniErasmus, and Desi denotes the proposed new language. (Actually, Desi refers to both the project and the language, but the intended meaning will usually be clear from the context.)

Section 3 describes the overall goals of the project. Subsequent sections consider one aspect of Desi, introducing goals, reviewing alternatives, and proposing solutions (or, perhaps, compromises). These sections are self-contained and, as far as possible, independent. That is, making a different choice from alternatives in one section should not invalidate choices in another section.

Each section with the title "Alternatives" reviews different possibilities in the design space. Read these sections if you are interested in discovering why certain design choices were made and others avoided. If you just want to find out how Desi works, these sections are likely to be confusing and you should skip them.

► Significant changes made since the previous version are flagged with a triangle in the left margin.

2. Conventions

2.1. Syntax

A syntax rule has the general form

Nonterminal = *Expression* .

Entities are distinguished by fonts, as follows:

Nonterminal **keyword** `terminal` information-carrying terminal

An *information-carrying terminal* is a terminal symbol as far as the grammar is concerned, but carries information that is useful to the compiler. Identifiers and literals are information-carrying terminals.

In addition to terminals and nonterminals, syntax expressions may contain the following metasymbols:

- Spaces denote concatenation. $A\ B$ means "A followed by B".
- `"|"` denotes alternation: $A \mid B$ means "either A or B".
- Brackets denote an option: $[A]$ means "either A or empty".
- Braces denote repetition: $\{A\}$ means "zero or more As".

- Braces followed by a subscript denote a list in which items are separated by the subscripted character. For example, "{ *Lvalue* }," expands to a possibly empty list of expressions separated by commas.

2.2. Compilation and Execution

The current (May 2011) plan is to construct the Desi system in two parts. A Universal Desi Compiler (UDC) translates Desi source code to Desi Intermediate Language (DIL) that is either interpreted or compiled. The DIL compiler is referred to as the JIT.[1]

Conventional distinctions (static/dynamic, compile/run time, etc.) may become ambiguous when implementation is based on byte-code interpreters and/or JIT compilers. In this document, we use these three terms:

Compile time is the time at which UDC translates Desi code to DIL;

Code-generation time is the time at which JIT translates DIL to machine code;

Run-time is the time at which machine code is actually executed.

3. General Goals

- Desi will be simple enough for a small team to implement.
- Desi will be rich enough to allow interesting and useful programs to be written in it.
- In order to meet the two preceding criteria, Desi will be designed as a *simple kernel* with hooks that facilitate *extension*.
- Desi will be a "load-and-go" language, similar to Smalltalk and Java, but unlike the edit/-compile/link/execute cycle of FORTRAN and C++.
- Desi will be more dynamic than mE. mE provides for dynamic creation of channels and processes, but Desi will go further. For example, it should be possible to log the behaviour of a running process intermittently.
- The basic building block of Desi programs is the *process*. The primary purpose of building programs from processes is to minimize coupling. A secondary, but important, benefit is to maximize concurrency.
- The *meaning* of a Desi program does not depend on how it is mapped onto the hardware. For example, loading all processes into a single memory space should give the same results as linking processes through a network. Of course, *performance* may be significantly affected by different deployment.
- Desi avoids sharing memory, for two reasons. The first reason is the "standard" reason: if memory is shared, race conditions must be avoided. The need to avoid races significantly complicates the development of shared-memory programs. The second reason is more subtle and applies mainly to very recent processors. Traditionally, sharing has been considered more efficient than copying, for obvious reasons. But with the use of multicore processors and multiple caching levels, there is a lot of copying going on anyway, and allowing several processors to share a variable may be more expensive than giving them each their own copy.

  At the lowest level, of course, memory will be shared. This is unavoidable, but should be entire;y invisible to Desi programmers.

[1] We use "JIT" (sans-serif) to refer to this program and "JIT" (Roman) as a general abbreviation for "just in time".

Table 1: Various rules for declaration and scope

DBU	Every name must be declared before it can be used ("declaration before use").
SIB	The scope of a name is the entire block containing its declaration ("scope is block").
DFU	Explicit declarations are not required; the first use of a name is assumed to be its declaration ("declaration at first use").
HIS	When the same name is used in two nested scopes, the inner name hides the outer name ("holes in the scope").
NDN	Using the same name at different nesting levels within a scope is an error ("no duplicate names").
GLO	When a name from an outer scope is imported into an inner scope, it must be declared **global**.

4. Scope Rules

4.1. Goals

Clearly, scope rules should be: intuitive; easy to remember; free of anomalies; and have no long-distance effects. As we show in the following section, few languages meet all of these requirements.

4.2. Alternatives

Table 1 lists some of the rules used to govern declarations and scopes. Each of these rules is used in at least one programming languages, but it is clear that they cannot *all* be used. The left column shows three-letter mnemonics that we will use in the discussion that follows.

As a first example, consider this snippet from a language that requires names to be declared; braces indicate scopes:

```
{
  n: Integer := 3;
  ....
  {
    write(n);
    n: Integer := 4;
    write(n);
  }
}
```

With DBU, the output from this snippet is "34". With SIB, however, the output would be "44". The same would be true for GLO, except that an extra declaration would be required. DFU is not applicable. HIS applies to the second use of n but not the first. With NDN, the snippet is illegal because n is declared twice.

Each rule has minor disadvantages, anomalies, or quirks:

- DBU causes the anomaly in the example above, where the two uses of n in the inner scope refer to different variables.
- With SIB, adding a declaration at the end of a block may change the meaning of statements that appear earlier in the block.
- HIS allows accidental hiding of names.

- Programmers may be surprised when NDN informs them that they have used a name from an outer scope of which they were unaware.
- GLO imposes the requirement to declare use of a global name in an inner scope.

Part of the problem with scopes is that different situations apply at different levels of scale. The outermost scope in which a component is compiled typically contains tens of thousands of names, almost any of which may legitimately be used anywhere in the program. Requiring, for example, explicit import of global names into a local scope would make programming tedious and prone to error. Moreover, disallowing redefinition of global names would place an intolerable burden on programmers.

On a small scale, such as the body of a process, however, things are quite different. The body of a process should not occupy more than a screenfull or two, and it is reasonable to require that names local to the process be unique.

4.3. Design

4.3.1. Module Scope

A *module* is a collection of definitions and invocations. Typically, a module is stored as a single file of source code.

Modules may contain **import** directives that refer to other modules. Two modules, A and B, are considered to be distinct even if A **import**s B or *vice versa*.

Every name used at the top-level of a module (i.e., not within a component) must have an explicit definition that is accessible to the compiler in one of the ways described below.

Names introduced by a top-level definition in a module may be used anywhere in the module.

When a module is imported, names introduced by **public** definitions are accessible to the importing module.

Importing is normally intransitive (that is, if module A imports module B, module A cannot access entities imported by B). However, **public import** is transitive: if an **import** definition is **public**, the imported entities are accessible to importing modules.

Suppose that a program contains the following modules:

Module `a.des`:
public *pubA* = **constant** "A1";
priA = **constant** "A2";

Module `b.des`:
public *pubB* = **constant** "B1";
priB = **constant** "B2";

Module `c.des`:
public import "a";
import "b";
public *pubC* = **constant** "C1";
priC = **constant** "C2";

Module `d.des`:
public import "c";

Table 2 summarizes the accessibility of the variables. 'Y' means that the name is accessible in the module named at the top of the column; 'N' means that the name is not accessible; and a blank means that name is not in scope. Here are explanations of each entry:

Table 2: Application of scope rules

Name	`a.des`	`b.des`	`c.des`	`d.des`
pubA	Y		Y	Y
priA	Y		N	N
pubB		Y	Y	
priB		Y	N	
pubC			Y	Y
priC			Y	N

- *priA*, *priB*, and *priC* are accessible only in the modules in which they are declared.
- *pubA*, *pubB*, and *pubC* are accessible in the modules in which they are declared and may be imported by other modules.
- *pubA* is accessible by public import in module `c.des` and by transitive public import in module `d.des`.
- *pubB* is accessible by import in module `c.des` but not in module `d.des` because `c.des` did not import module *b* publically.
- *pubC* is accessible by **public import** in module `d.des`.

► The current version of UDC ignores **public** and **private** declarations, treating everything as **public**.

4.3.2. Local Scope

In this section, we describe only scope rules that apply to sequences within routines and processes.

As mentioned in Section 4.3.1, names defined globally in a module may be used anywhere in the module. "Anywhere" includes inner scopes at any nesting level. We refer to these names as *global names*. A name that is not global is *local*.

In the following rules, temporal expressions such as "first" and "before" refer to places in the text and have nothing to do with time of execution.

- All names (types, constants, and variables) must be declared explicitly.
- The scope of a declaration consists of the entire block in which the declaration appears. In the following snippet, the two occurrences of *y* refer to the same variable, which is local to this scope.

```
{
    x: Word := 1;
    y := 2;
    z := 3;
    y: Real;
}
```

- If the same name is declared in both an outer scope and a nested inner scope, the declaration in the inner scope creates a "hole" in the outer scope. In other words, the name is not accessible within the inner scope.

  The code below is legal. The outer *x* is assigned 1 and the inner *x* is assigned 2.

```
{
    x := 1;
    loop
    {
        x := 2;
        x: Real;
    }
    x: Real;
}
```

- In particular, local names hide global names.

5. Program Structure

5.1. Goals

The goals for program structuring facilities include:

- Programs should have a clear and simple structure (whatever that means).
- Typical programs should not require a large number of nested scope levels.
- Programs should be scalable: recall the early goal of "fractal software".
- Programs may be composed from multiple files.
- The syntax should be consistent so that, once learned, a syntax rule applies throughout the language.

5.2. Alternatives

In mE, statement groups were delimited by '|' and **end**. Desi uses somewhat more "modern" syntax, with '{' and '}' as delimiters.

mE-style declarations, however, have been retained. They have the form *name* = *body* and allow, at least in principle, the *body* to be used as an unnamed entity, e.g., an anonymous function or "lambda".

With respect to separators, the syntax has been designed to meet the following goals as far as reasonably possible:

- There should be enough redundancy that the parser can recover quickly from most syntax errors.
- Programmers should have some flexibility to use a style that they like or are familiar with.
- Source code should not be cluttered with redundant separators.

The following possibilities were considered and rejected for the reason given:

- Algol60/Pascal style: the semicolon is a separator. Studies have reported that programmers do not like the inconsistency of this approach. It requires, for example, that removing the second statement from

```
{
    a := 4;
    b := 5
}
```

Table 3: The role of semicolons

	1	2	3	4	5	6
Separator	×	×	•	×	•	×
Terminator	•	•	•	•	×	•
Natural	×	×	•	×	×	×

requires removing the semicolon as well.

- C/Java style: the semicolon is part of a declaration or statement. The problem with this convention is that many redundant semicolons are needed. In fact, C and Java both allow "`for (i = 0; i < M; ++i)`".
- No separators at all. Even if the grammar can be made unambiguous, syntax errors may give rise to misleading diagnostics.
- Python/Haskell style: indentation determines structure. This would work well if the world could agree on a consistent definition of blanks and tabs.

Various conventions were considered, and none seemed entirely satisfactory. Either the code ended up being cluttered with too many required semicolons, or there were inconsistencies in semicolon placement. In the following snippet, semicolons have been numbered.

example = **process** *p*: +*prot* ; (1)
{
  loop *i* := 1 ; (2)
  {
    | *i* < 10|
      write(*out*, *i*) ; (3)
      i += 1 ; (4)
  } ; (5)
  write(*out*, "finished") ; (6)
}

Table 3 shows which semicolons would be required according to various conventions.

- If semicolons are separators, as in Pascal, only the semicolons marked with a • in the first row are required. In particular, note that the semicolon 5 is required, even though the closing brace makes it seem unnecessary. (The rule about when to put semicolon after **end** used to confuse Pascal programmers.)
- If semicolons consistently terminate statements, those marked in the second row are required. Some of these seem unnecessary, either because they are immediately followed by other punctuation (1, 2, 4, 6), or because there is another terminator (5). Although it is widely believed that semicomons terminate statements in C, semicolon 5, which is not required in C, show that this is false.
- The only semicolon that actually separates statements in this short example is 3. Although even this one is arguably unnecessary, since the call is clearly terminated by a parenthesis, most people would feel it natural as a separator.

The Desi grammar tries to effect a compromise between styles.

5.3. Comments

Programs are written as LaTeX documents in which Desi code appears between `\begin{code}` and `\end{code}`. Code between `\begin{dummy}` and `\end{dummy}` is formatted in the same way but is not read by the compiler.

Source files have extension `.tex` and may be processed by LaTeX, provided that the package `cloe` is included and text outside the `code` and `dummy` environments conforms to LaTeX conventions. Code segments will be typeset using the conventions for the `listing` style defined in `cloe.sty`.

Within a code block, the symbol "`--`" (two adjacent hyphens) tell the compiler to ignore the rest of the current line.

Here is a short source file that illustrates these conventions:

```
This is commentary.
\begin{code}
answer := 42; -- source code with comment
\end{code}
More commentary.
\begin{dummy}
answer := 42; -- formatted as code but ignored by the compiler.
\end{dummy}
More commentary.
```

5.4. Design

Table 4 summarizes the top-level syntax of Desi.

Table 4: Program structure

Module	=	{ [**public**] (*Import* | *Definition* | *Invocation*) } .
Import	=	**import** { TextLiteral }‘,’ [‘;’] .
Definition	=	*ConstantDefinition*
	|	*TypeDefinition*
	|	*EnumDefinition*
	|	*ProtocolDefinition*
	|	*RoutineDefinition*
	|	*ProcessDefinition*
	|	*CellDefinition* .

The expects a Desi source file to contain a *Module* containing imports, definitions, and invocations.[2] If there are no invocations, the module is a *library*, otherwise it is a *program*.

The syntax allows a program to contain several invocations. However, since variable and channel declarations are not allowed at the top level, the entities invoked have no way of communicating. Consequently, a typical program contains one invocation of a parameterless cell that instantiates a collection of cells and processes.

An *Import* directive instructs the compiler to read source code from other modules.

[2] We use the pompous term "invocation" rather than the simpler "call" because the latter suggests functions, but we may actually be invoking a cell, process, or routine.

Table 5 shows the forms of the various definitions. The names of entities are identifiers, except for routines, which may have names like "**op+**". Detailed explanations of the components are given in subsequent sections: types (Section 6); literals (Section 7.3.2); routines (Section 9); processes (Section 11); cells (Section 12); protocols (Section 13).

Table 5: Varieties of definition

RoutineDefinition	=	(iden | **op** (*UnaryOp* | *BinaryOp*)) '=' [**implicit**] **routine** *Parameters* (*Sequence* | '=' Intrinsic ';' | **external** TextLiteral ';') .
ConstantDefinition	=	iden '=' **constant** *Rvalue* ';' .
TypeDefinition	=	iden '=' **type** [*Type*] ';' .
ProtocolDefinition	=	iden '=' **protocol** '{' { ['^'] iden [':' iden] }',' '}' .
ProcessDefinition	=	iden '=' **process** *Parameters* (*Sequence* | **external** TextLiteral ';') .
CellDefinition	=	iden '=' **cell** *Parameters* (*Sequence* | **external** TextLiteral ';') .

The compiler attempts to infer the type of the constant from the form of the expression. If it cannot do so, it reports an error. If the programmer prefers the constant to have a different type, the expression can be cast. For example, the compiler would infer the type of $year - 2000$ as *Word*, but the constant *Adjusted* has type *Text*:

Adjusted = **constant** *Text*(*year* − 2000)

The *Rvalue* is evaluated and its value is bound to the identifier. Subsequent assignments to the identifier are not allowed.

Most types will be defined at global scope; defining a type for a single process is unlikely to be useful. However, it is reasonable to expect an occasional type definition within a process, for an enumeration perhaps, and so type definitions will eventually be allowed at the statement level. Types are described in Section 6.3.

► Eventually, we should allow all definitions to appear in local or global contexts, except that variables and channels should not be global.

5.4.1. The Standard Prelude

The Standard Prelude is a module that contains declarations for types and routines that are commonly used. Unlike other modules, the Standard Prelude cannot be altered arbitrarily. For example, many of the routines in it are defined with an intrinsic name (written `$abc`) or an empty body (written {}), which means that the compiler knows how to generate appropriate in-line code for them. Other routines are written with a body of the form "= *name*", in which *name* refers to an external function that must be accessible to the compiler.

Consequently, the Standard Prelude should be viewed as documentation, not as code to be hacked.

► Following changes to the DIL specification, the number of distinct intrinsic names is rather small. For example, all conversion routines now have body `$c`. Previous versions of this document referred to external functions as "C functions". This version refers to "external functions" even though C is the language most likely to be used for implementation.

6. Types

6.1. Goals

Desi will provide a set of *basic*[3] and *standard* types and an unlimited number of *user-defined types*. As far as possible, programmers should not have to know whether they are using a basic type or a user-defined type. For example, if values of type *Integer* (assumed to be basic) can be added with infix '+', then users should be able to define another numeric type, such as *Complex*, and add instances with infix '+'.

In addition to types, there are also *type constructors* (a.k.a. *type generators*, *compound types*, etc.) that enable the programmer to build complex types from simple types.

In object-oriented programming (OOP), there is a close relation between classes and types, with some languages actually identifying these concepts. Desi will have a similar relation between cells and types. The difficult part is to provide the nice features of sequential OOP while avoiding the pitfalls of multithreaded OOP. If an object is multithreaded, it must avoid race conditions on its instance variables. This can be accomplished by making it a monitor that can execute only one method at a time, but this might make the object a bottleneck. (See also Section 12.)

6.2. Alternatives

Any programming language requires integral types. There are various ways of providing such types, ranging from a single type of all storable integers to a plethora of types defining signed and unsigned integers of various sizes. The following types seem to be fundamental:

- *Bytes* have been adopted as the smallest unit of addressable storage on almost all contemporary architectures. Bytes are most commonly used for organizing data on a small scale (e.g., packing machine instructions) and are rarely used for arithmetic. Consequently, the Desi basic type *Byte* represents unsigned integers in [0, 256).
- Most architectures provide a *word* that is larger than a byte (typically with 32 or 64 bits) and arithmetic operations on words. Calculations may involve positive and negative quantities. Providing signed and unsigned variants does not add much except, perhaps, slightly increased security. Consequently, the Desi basic type *Word* represents signed integers that are as large as the architecture supports efficiently.
- Finally, there may be situations in which programmers want to work with quantities larger than a *Word*. For these purposes, Desi provides a standard type *Integer* that represents integers of arbitrary size.

Most languages have a type constructor for arrays. The elements of an array are all of the same type and are stored contiguously in memory, provided efficient access. One point of view is that arrays are low-level, machine-oriented constructs that are inappropriate for modern, high-level languages. The problem with this view is that there are a few applications for which arrays are essential—e.g., code generation, image processing, etc.

Maps are a high-level generalization of arrays. A map is a set of values of one type that are indexed by values of another type.

Arrays and maps overlap but both are needed: arrays for low-level applications such as storing compiled code, and maps for higher level applications. The syntax for access is the same for each (e.g., $a[i]$) but the compiler uses type information to generate appropriate code.

[3] These were previously called "intrinisc" types, but the term "basic" types has been used more often and avoids confusion with intrinsic routines.

Linked structures, such as lists, trees, and graphs, pose a problem for Desi.

- Pointers provide a natural way of implementing linked structures: each node contains stores the addresses of nodes related to it. This technique works well in a single memory space but is not easily extended to distributed spaces.
- Recent languages eschew pointers in favour of references. However, references are little more than thin clothing for pointers, hiding the private parts and making things slightly simpler for programmers. Under the hood, the addresses are still there.
- If values are instances of cells then, by analogy with OOP, cells could reference one another. This would work against our desire to freely move cells between memory spaces.
- Cells could reference one another using channels instead of references. This is consistent with the general goals of Desi, and allows a linked structure to use as many processors as it has nodes. The main problem would be the time taken to exchange messages.

See Section 18.1.3 for further discussion.

The following definition for array types was considered and abandoned, since a map with a modular domain serves the same purpose:

> A type followed by an expression in square brackets is an array type. The *Rvalue* between the brackets is not necessarily a constant, but its value must be integral and known at compile time. If its values is N, elements of the array are indexed by $0, 1, \ldots, N-1$. Arrays are regarded as low-level, primitive values: indexing is fixed, and they cannot be resized.

► The type system presented here does not allow for any form of genericity (e.g., parameterized types) or specialization (e.g., inheritance). These are subjects of on-going discussion.

6.3. Design

Table 6: Syntax for types

Type	=	iden
	|	*Type* **indexes** *Type*
	|	**mod** NumericLiteral .

When a type is defined simply by an identifier (the first alternative in the rule for *Type* in Table 6), it belongs to one of three categories: *basic*, *standard*, or *user-defined*.

A *basic type* is a type for which the compiler has complete information; most operations on basic types are translated to a single opcode in DIL. Modifying the properties of basic types usually involves modifying the compiler. Table 7 describes them.[4]

A *standard type* is a type about which the compiler has partial information, most of which is provided by the Standard Prelude. Standard types are implemented by calling external functions rather than by emitting DIL opcodes. Consequently, standard types can be added and modified without changing UDC. Table 8 suggests some of the standard types that an implementation might provide.

► So far, most of our effort has focused on basic types, in the expectation that standard types will be implemented by libraries. UDC will have to recognize them, but will do no more than emit calls to external functions. This, design of standard types awaits decisions about the implementation of libraries.

[4] Many years ago, PG was chastised by Willem van der Poel for using *Real* as a type name. Computers, he was told, provide rationals, not reals. But *Real* is a more descriptive name than *Double* and less implementation-oriented than *Float*.

Table 7: Basic Types

Type name	Description
Boolean	The type with values **true** and **false**, represented by any convenient number of bits. May be abbreviated as "*Bool*".
Byte	The smallest addressable unit, usually 8 bits with values $0, 1, \ldots, 255$.
Word	The most efficient addressable unit for the architecture, at least 32 bits, but possibly more.
Real	A floating-point representation that is supported efficiently by the architecture and has sufficient precision for most applications. Typically, this will be the IEEE 754 *double* (or *binary64*) format.
Character	A Unicode character, probably represented with 32 bits. May be abbreviated as "*Char*".
Text	A sequence of *Character*s, probably represented as a contiguous array. *Text*s should be implemented efficiently, but implementation details, such as a length field or a terminating character should be invisible to programmers.
Void	A type with a unique value, used as a place-holder. For example, the type of a signal is *Void*. The implementor must allocate at least $\log_2 1$ bits of memory for each value of type *Void*.

Table 8: Possible Standard Types

Type name	Description
Integer	Integers of arbitrary size ("big integers").
Fraction	Rational numbers of arbitrary precision, represented as the ratio of two big integers.
Complex	Complex numbers represented as pairs of *Real*s, $x + iy$.
Stream	A sequence of *Character*s stored on an external device, such as a disk. This type is distinct from *Text* because different operations are provided: see Section 10.3.

6.4. User-Defined Types

A user-defined type is a type that is described to the compiler by an explicit definition. There are only a few forms of definition, and each is described in this section.

6.4.1. Enumerations

A type expression that starts with **enum** introduces an *enumerated type*. The values of the type are denoted by the identifiers in the list following the keyword **enum**.

Enumerated types have limited properties:

- variables of an enumerated type may be declared and assigned values of the enumeration;
- two values of an enumeration may be compared for equality or inequality;
- a value of an enumeration is not equal to any other value of the enumeration;
- a value of an enumeration can be cast to *Word*;
- a **for** or **any** statement can be used to iterate over the values of an enumeration.

6.4.2. Modulo Types

A type expression that starts with **mod** introduces a *modular type*. The *Rvalue* must evaluate to a positive value, M, which may be a *Word* or a *Real*. The values of the type satisfy $0 \leq x < M$. For example, values of the type "**mod** 10" belong to the set $\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$. Modulo types are intended to be useful for applications such as circular buffers.

► The current version of UDC accepts only a numeric literal after **mod**. Variables of the type are stored as *Word*s or *Real*s. When the value is updated, its range is restricted to $0 \leq x < M$.

When a modulo type is used as a subscript, out-of-range errors cannot occur, because every value of the modulo type is a valid subscript. It is not clear that this is desirable. Although it adds a slight convenience to programmers, it also hides an important class of error.

6.4.3. Maps

A type with the keyword **indexes**, such as "*Word* **indexes** *Text*", is a *map type*. The type on the left is the *domain type* (also called *index type*) and the type on the right is the *range type*.

The domain type of a map may be sparsely populated. For example, a map of type *Integer* → *Text* could be indexed by 2, 3, 5, 7, ... There is one exception to this rule: if the domain type is **mod** N, in which N is a positive integer, the compiler may assume that indexes are 0, 1, 2, ..., $N-1$ and allocate contiguous space for the N values.

► We do not currently have any way of indicating a dense domain in DIL.

Example These are valid type definitions. Note that the keyword **type** is required in a top-level definition (where the parser does not know what to expect after '=') but not in a context in which a type is expected.

Number = **type** *Word*;
Colour = **enum** { *violet*, *indigo*, *purple* }
Angle = **type mod** 2 ∗ *PI*;
Subscript = **type mod** 500;
Array = **type** *Subscript* **indexes** *Real*;
Dictionary = **type** *Text* **indexes** *Text*;

► A few minor changes have been made to the syntax for type definitions:

1. Consider this routine definition:

   exp = **routine** *x*: *Real* → *y*: *Real* { ... }

   The parser naturally tries to interpret *Real* → ... as a map type. To avoid this ambiguity, we have reverted to the mE form '*D* **indexes** *R*' for map types.

2. A type definition can be empty. That is, the following declaration is allowed:

   Word = **type**;

   This convention allows us to inform the compiler that *Word* is a type in a standard imported source file without requiring any magic. Empty definitions should be used only in the Standard Prelude.

6.5. Notation for Types

To save space later on, we introduce some abbreviations for documentation concerning types.

- $\mathcal{T}(E)$ stands for "the type of expression E"
- $T_1 \leq T_2$ stands for "T_1 can be coerced to T_2"

For example, $\mathcal{T}(555) = \mathit{Word}$ and $\mathit{Byte} \leq \mathit{Word}$.

7. Expressions

► Several of the proposals in this section are derived from KIWI 100131 *Some Design Experiments*. In particular, Desi provides a rather C++-ish form of the symbolic tags suggested in §1.5.

As in most languages, an *expression* is a syntactic unit that is *evaluated* and yields a *value*. Desi expressions do not have side-effects.

7.1. Goals

The meaning of an expression should be obvious to a reader, even a reader who is not a Desi expert. This suggests that a familiar notation is more likely to gain acceptance than an oddball notation such as operator postfix.

Nevertheless, the notation should be flexible and should allow for extensions, such as overloaded operators.

7.2. Alternatives

The syntax of literals is kept simple. The compiler infers the type of a literal and a programmer who does not like the inferred type can cast it. Thus, for example, there is nothing corresponding to the 'l' suffix for *long* in Java and C++.

Various ways of structuring expressions, and the choices proposed for Desi, are listed here.

- Several different notations are available: prefix or functional, infix, or postfix. We propose the most conventional: prefix for unary operators; infix for binary operators.
- Evaluation order can be fixed (e.g., left to right); determined by precedence levels; or determined by parentheses. Again, we propose conventional evaluation, governed by precedence levels with parentheses to override when necessary.
- Operators may be overloaded. Overloading of the most common operators (e.g., '=' to compare values of various types, '+' used to add integers and floats) is more or less universal. Desire for consistency suggests that all operators may be overloaded.
- Operators may be user-defined. If operators are overloaded anyway, allowing users to overload them further does not greatly complicate the implementation. Again, desire for consistency suggests that the same syntax should be used for adding integers as for adding some exotic algebraic type (polynomial, quaternion, etc.) introduced by the programmer.

- Two issues arise for user-defined operators. Can the user introduce new symbols? And can the user define precedence levels? Both options have potential complications. There must be a way of resolving conflicts when a program uses two modules that define the same operator symbol for different purposes and, perhaps, with different precedences. Even worse, the modules might define the same operator symbol at different levels of precedence.

  For Desi, we propose essentially the system used by C++: there is a fixed, but fairly generous, collection of operator symbols, each with a fixed precedence level. These operators may be overridden for operands of different types.

7.3. Design

Table 9 defines the syntax for expressions. Binary operators are shown separately in Table 10. The syntax in these tables includes only operators that are required by the language defined herein. A parser may recognize other operators for user-defined functions.

Operator precedences are not specified by the grammar but are shown in Table 12.

Table 9: Syntax for expressions

Lvalue	=	iden { '[' *Rvalue* ']' } [('.' | '?') *Rvalue*] .
Rvalue	=	*Lvalue*
	|	*Literal*
	|	*UnaryOp Rvalue*
	|	*Rvalue BinaryOp Rvalue*
	|	*Rvalue* **if** *Rvalue* **else** *Rvalue*
	|	*Rvalue* '[' *Rvalue* '..' *Rvalue* ']'
	|	*Invocation*
	|	'(' *Rvalue* ')' .
Invocation	=	iden '(' { *Rvalue* }‹,› ['->' { *Lvalue* }‹,›] ')' .
UnaryOp	=	'+'
	|	'-'
	|	'~'
	|	'#'
	|	**not** .
BinaryOp	=	*ArithmeticOp*
	|	*CompOp*
	|	*ShiftOp*
	|	*BoolOp*
	|	*BitOp*
	|	*TextOp* .

Semantically, lvalues and rvalues are distinct, because an lvalue denotes a container for a value and an rvalue denotes a value. Syntactically, however, lvalues are a subset of rvalues, and this is expressed in the syntax for *Lvalue* and *Rvalue*. The parser can distinguish lvalues and rvalues from the context. Specifically, an expression is an lvalue if it is used:

- on the left side of an assignment statement (Section 8.3.3); or
- as an argument corresponding to an output parameter (Section 9.3).

Table 10: Binary operators

ArithmeticOp	=	'+'
	|	'-'
	|	'*'
	|	'/'
	|	'%' .
CompOp	=	'='
	|	'<>'
	|	'<'
	|	'<='
	|	'>'
	|	'>=' .
ShiftOp	=	'<<'
	|	'>>'
	|	'>>>' .
BoolOp	=	**and**
	|	**nand**
	|	**or**
	|	**nor**
	|	**xor**
	|	'==>'
	|	'<=='
	|	'<=>' .
BitOp	=	**andb**
	|	**orb**
	|	**impliesb**
	|	**xorb** .
TextOp	=	'//' .

7.3.1. Conversion

A value may be converted from one type to another either implicitly, by *coercion*, or explicitly, by *casting*. Coercion requires no special syntax; casting uses routine call syntax, with the target type as the routine name. Thus $T(e)$ invokes code that converts the expression e to type T, assuming that the appropriate conversion has been declared.

Table 11 shows how conversions between basic types work. The cell entries have the following meanings:

$-$: the types are the same and no conversion is necessary.

I: UDC generates conversion code implicitly. Since the source is smaller than the destination, checking is unnecessary.

C: UDC generates conversion code implicitly. Since the source may be larger than the destination, UDC will perform a check, at compile time if possible, otherwise at run time.

T: the conversion function must be provided explicitly by a routine whose name is the destination type.

F: the conversion function must be provided explicitly to obtain the correct semantics. The functions available are *floor*, *round*, and *ceil*.

$\square$: an empty cell indicates that the conversion cannot be performed in a single operation.

Table 11: Conversions between basic types

↗	*Boolean*	*Byte*	*Word*	*Real*	*Char*	*Text*
Boolean	–		*T*			*T*
Byte		–	*I*	*I*		*I*
Word		*C*	–	*I*	*T*	*I*
Real			*F*	–		*I*
Char			*T*		–	*I*
Text					*C*	–

Remarks:

- There are no conversions to *Boolean*.

  These is no universal translation from non-*Boolean* values to *Boolean* values. However, common conventions are easy to implement. For example, the convention that a *Word* w is "true" if and only if it is not zero can be implemented as $w <> 0$.

- Explicit conversion is required between *Word* and *Char*, even though it is likely that no operation is needed, to prevent accidental conversion and sloppy programming.

► • Every basic type except *Boolean* can be implicitly converted to *Text*. Coercion of *Boolean*s would allow expressions such as $2 < 3 < 4$ to be compiled (by implicitly converting **true** and 4 to *Text*, and then comparing them).

7.3.2. Literals

Literal	=	NumericLiteral
	|	TextLiteral
	|	*BooleanLiteral*
	|	*ArrayLiteral*
	|	*MapLiteral* .
BooleanLiteral	=	**true**
	|	**false** .
ArrayLiteral	=	'{' { *Rvalue* }',' '}' .
MapLiteral	=	'{' { *Rvalue* '->' *Rvalue* }',' '}' .

Numeric literals are either decimal or hexadecimal. Decimals literals have the form

```
digit *digit [ .*digit ] [ (e|E) [+|-] digit *digit ]
```

If either or both of the optional fields are present, the literal is interpreted as a floating-point value (basic type *Real*), otherwise it is assumed to be an integer (basic type *Word*).

Underscores within a numeric literal are ignored, thereby allowing long numbers to be written in groups of three digits. (It would be more conventional to use comma as a separator, but this risks confusion with expression lists.)

A hexadecimal literal starts with the characters '`0x`' (that is, 'zero ex') and continues with hexadecimal digits. The hexadecimal digits consist of decimal digits, `"0"`, `"1"`, ..., `"9"`, and the letters

`"A"`, `"B"`, ..., `"F"`. Both upper and lower case letters are allowed for the hexadecimal digits, but the '`x`' must be lower case.

A text literal is enclosed between a matching pair of quotes, which may be either single, as in `'single'`, or double, as in "double". A single-quoted text may contain double-quoted characters and *vice versa*. The usual escape sequences are allowed, including `\uxxxx` for Unicode symbols.

A quoted literal that contains exactly one character is considered by the compiler to be a *Character* literal but, in a context in which a *Text* is expected, it will be coerced to *Text* (see Section 7.3.1).

The literals of type *Boolean* are keywords: **true** and **false**.

An array literal is just a list of expressions. The expressions do not have to be constant, provided their values can be computed when the literal is needed.

A map literal is similar to an array literal, but each component is an index/value pair.

► UDC does not yet recognize array or map literals.

Examples These are literals:

Numeric:	99
	2_432_902_008_176_640_000
	0xffff
	Decimal(3.14159265358979323846264338327950288)
Text:	"I'm a double−quoted text"
	'And this is a "single−quoted" text'
	"Here are some escapes: \n \t \\"
	"\u2297 is Unicode for ⊗ "
Array:	{ 2, 3, 5, 7, 11, 13 }
Map:	{ 'e' → 2.71828, 'pi' → 3.14159, 'gamma'→ 0.57722 }

7.3.3. Conditional Expressions

Conditional expressions have the form "E_1 **if** E_2 **else** E_3" and are evaluated as follows:

- Evaluate E_2 and let B be its Boolean value.
- If $B =$ **true**, evaluate E_1 and return its value.
- Otherwise, evaluate E_3 and return its value.

The **else** part of a conditional expression can be conditional, as in this example:

```
size: Text := "small" if x < 10 else
              "medium" if x < 30 else
              "large";
```

Table 12: Operator Precedence

Precedence	Operators	Bitwise Operators
10	`+` `-` `#` **not**	`~`
9	`*` `/` `%`	
8	`+` `-`	
7		`<<` `>>` `>>>`
6	`<` `<=` `=` `<>` `>=` `>`	
5	**and** **nand**	**andb**
4	**or** **nor** **xor**	**orb** **xorb**
3	`==>` `<==`	**impliesb**
2	`<=>`	
1	`//`	
0	... **if** ... **else** ...	

7.3.4. Text

The operator `//` concatenates expressions of type *Text*. It goes slightly beyond the usual rules for implicit conversion, and will attempt to coerce its operands to *Text*s whenever possible. Statements such as this one are fine:

msg: *Text* := "Pi = " // 3 // "+" // 0.14159;

If t is a *Text*, then $t[i]$ is the ith character of t. The first character of t is $t[0]$ and the last character is $t[\#t-1]$, in which $\#t$ is the number of characters in t. The value of i should satisfy $0 \le i < \#t$; the run-time system reports a range error otherwise.

Furthermore, $t[i..j]$ is the *Text* consisting of characters i through $j-1$ of t. The values of i and j should satisfy $0 \le i < \#t$ and $0 \le j < \#t$, respectively. If $j - i > 0$, the resulting *Text* has $j - i$ characters; otherwise it is empty.

In the following examples, the expression on the left has the value shown on the right.

t: *Text* := "abcdefg";	
t[0]	Char('a')
t[6]	Char('g')
t[1..4]	'bcd'
t[3..3]	"

7.3.5. Operators

Table 12 shows the precedence of binary operators.

The operators are conventional but a few points may be worthy of note:

- Unary operators have the highest precedence and appear in the first row of the table. The operators in the remaining rows are binary operators.
- The unary operator `~`, applied to an operand of type *Word* or *Byte*, returns its operand with all bits inverted.
- The unary operator `#` may be placed in front of any expression and returns a non-negative integer that has some relation to the size of the expression. See Table 13.

► Currently, `#` may be used only to find the number of characters in a *Text* value.

- Operator `%` (modulus) can be used with operands of type *Word* or *Real*.
- Shift operators have conventional precedence (see Table 12) and include: logical left (`<<`); logical right (`>>`); and arithmetic right, i.e., with sign bit propagated (`>>>`). The left operand contains the bits to be shifted and may be a *Byte* or a *Word*. The right operand specifies the number of bits to shift and is a *Word*.
- Operators **andb**, **orb**, **xorb**, and **impliesb** are bitwise operators that act on *Byte*s or *Word*s.
- The type *Boolean* is ordered with **false** < **true**. Consequently, the comparison operators `<`, `<=`, `=`, `<>`, `>=`, and `>` can be used with *Boolean* operands and yield a *Boolean* result.
- Most comparisons (operators `<`, `<=`, `>`, `>=`, `=`, and `<>`) are defined as routines in the prelude. Additoinally, ports may be compared with for equality (`=`) and inequality (`<>`).
- In addition there are eight *Boolean* operators that represent logical operations: **and**, **nand**, **or**, **nor**, **xor**, `==>`, `<==`, and `<=>`. With **and** and **or** defined conventionally, we have the following equivalences:

$$\begin{aligned} P \textbf{ nand } Q &\equiv \textbf{not } (P \textbf{ and } Q) \\ P \textbf{ nor } Q &\equiv \textbf{not } (P \textbf{ or } Q) \\ P \textbf{ xor } Q &\equiv P \neq Q \\ P \texttt{ ==> } Q &\equiv \textbf{not } P \textbf{ or } Q \\ P \texttt{ <== } Q &\equiv P \textbf{ or not } Q \\ P \texttt{ <=> } Q &\equiv P = Q \end{aligned}$$

- Some of the *Boolean* operators are implemented as conditional expressions ("lazy evaluation"):

$$\begin{aligned} P \textbf{ and } Q &\equiv Q \textbf{ if } P \textbf{ else false} \\ P \textbf{ nand } Q &\equiv \textbf{not} Q \textbf{ if } P \textbf{ else true} \\ P \textbf{ or } Q &\equiv \textbf{true if } P \textbf{ else } Q \\ P \texttt{ ==> } Q &\equiv \textbf{true if not } P \textbf{ else } Q \\ P \texttt{ <== } Q &\equiv \textbf{true if } P \textbf{ else not } Q \end{aligned}$$

- The operator `//` implicitly casts its operands to *Text*s and yields their catenation.

Table 13: Effect of `#`

If E is:	return:
Char, *Integer*	number of bytes used to store E
Text	number of characters
Array	number of elements that can be stored (i.e., capacity)
Map	number of elements actually stored

Desi treats operators as routines. Let '$\oplus$' to stand for an arbitrary binary operator symbol. For each symbol $\oplus$, there is a corresponding routine name, **op**$\oplus$. The expressions $x \oplus y$ and **op**$\oplus(x, y)$ are equivalent, and the implementation of $\oplus$ is specified by a declaration of the form

$$\textbf{op}\oplus = \textbf{routine } x : T,\ y : T \rightarrow z : T\ \{\ldots\}$$

For further details of routine declarations, see Section 9.

Desi allows routine names to be overloaded and it follows that operators can also be overloaded. Section 9.3 explains how invocations of overloaded routines are processed.

8. Statements

Many of the proposals in this section are derived from KIWI 090923 *A Latvian Perspective*, which suggested an even more radical rationalization of the statement syntax of mE than we have dared to advance here.

As in most languages, a statement is executed for its effect. A *sequence* consists of several statements executed one after the other. Unlike Occam, a statement is not considered to be a process.

8.1. Goals

mE grew incrementally, and the final syntax was somewhat messy. Nevertheless, lessons were learned, and the successes and failures of mE syntax can be used to guide the design of Desi syntax. For example, the *guarded sequence*, $|G_1|\ S_1\ |G_2|\ S_2\ \ldots$, forms the basis of several kinds of statement.

8.2. Alternatives

The potential danger of guarded sequence syntax is that it may give code a homogeneous appearance in which different control structures do not stand out clearly enough. We will find out whether this is a serious problem in practice only by using the language.

8.2.1. Skip

The **skip** statement of mE has been dropped, but could be reintroduced if it turns out that there is a need for it.

8.2.2. Assignment

Assignments of the form "$a[\,i\,]\ +\!\!=\ e$" are allowed. The effect of this statement i to evaluate e and add this value to $a[i]$. Assignments of the form "$p.f\ +\!\!=\ e$" are *not* allowed. The natural expansion of this statement "$p.f := p.f\ +\ e$", which would require f to be a bidirectional field.

8.2.3. Selection

In mE, the policies for selection (**fair**, **ordered**, and **random**) were keywords, which implied that the policy was fixed at compile time. In Desi, the policy is a constant or variable of the enumerated type *Policy*, which allows the choice of policy to be made at run time. This complicates the implementation somewhat, but is compatible with JIT compilation.

Flexible Ordering It is tempting to write a simple summer using a one-line **select** statement:

```
Num = protocol { w: Word }
summer = process in1: +Num; in2: +Num; out: Num
{
  loop select
  {
    || out.w := in1.w + in2.w
  }
}
```

The problem with this code is that it does nothing until it has received data from *in1* and then strictly alternates between *in1* and *in2*. We can do better with a state machine, but only at the expense of several extra lines of code and duplication:

```
summer = process in1: +Num; in2: +Num; out: Num
{
  loop select
    w1: Word;
    w2: Word;
    s: Word := 0
  {
    |s = 0| w1 := in1.w; s := 1
    |s = 0| w2 := in2.w; s := 2
    |s = 1| w2 := in2.w; out.w := w1 + w2; s := 0
    |s = 2| w1 := in1.w; out.w := w1 + w2; s := 0
  }
}
```

It would be nice to allow more elegant ways to code applications of this kind.

Dynamic Branches Another limitation of mE was that **select** statements had a fixed number of branches. It was not possible to implement a process with a specification like this:

> Write a server that initially has no clients. As the server runs, it can accept new clients (passed to it as channels), and respond to them.

The following snippet suggests a possible way of implementing such a server.

```
Faq = protocol { qry: Text; ↑ans: Text }
Chan = protocol { f: Faq }
Server = process c: +Chan; q[]: +Faq
{
  loop select
  {
    || q += c.f
    || t: Text := q[i].qry; q[i].ans := reply(t, i)
  }
}
```

The protocols are conventional. The proposed new features are as follows:

- The parameter $q[]$ indicates that q is an array of ports. The array is initially empty.
- The effect f the **select** branch $p \mathrel{+}= q.f$ is to add a new port to the array q. The compiler checks that the port has the correct type (*Faq* in this case).

- The variable i occurs three times in the next line. The first occurrence is binding: one of the q ports is ready, and i indexes this port.

  The second and third occurrences of i in the branch are uses. Port $q[\,i\,]$ is used to reply on the same channel. The call $reply(\,t,\ \ i\,)$ illustrates the fact that i can be used to discriminate between ports.

A slight variation would make the acceptance of new ports explicit:

```
....
Server = process c: +Chan; q[]: +Faq
{
  np := 0;
  loop select
  {
    || q[np] := c.f; np += 1
    || ....
```

The notation is tentative and could be improved. In particular, it would be better to have a way of distinguishing binding and use occurrences of the port index.

8.2.4. Failure

In this document, we sometimes use the term "failure". For example, we say that the statement "**assert**(c)" fails if the condition c yields **false**.

But what does it mean to say that a large, possibly distributed, program "fails"? Where should the message be sent to? Hopefully, we can come up with some reasonable definition.

8.3. Design

Actions are performed within a *UnguardedSequence* or *GuardedSequence*. This convention improves the uniformity of the language: **if** and **case** statements are not required and, in most situations, the **exit** statement is not required either. Each of the other forms in Table 14 is described in one of the sections that follow.

8.3.1. Sequences and Guards

A *Sequence* is just a list of *Statement*s. Semicolons are considered to be parts of statements, not separators. They are included explicitly in the syntax of statements when required.

A *Sequence* defines a scope. Variables declared within a sequence may be used anywhere within the sequence and in nested sequences (unless their declarations are overridden by nested declarations: see Section 4.3.2.

A *GuardedSequence* is a list of statement sequences, each preceded by a *Guard*.

A *Guard* is a *Boolean* expression (with one exception for **case** statements, described in Section 8.3.4) between vertical bars. The *empty guard* is written $||$, and is equivalent to $|\mathbf{true}|$.

Execution of a guarded sequence consists of executing *exactly one* of the sequences with a **true** guard.

The order in which guards are evaluated depends on the form of the statement enclosing the guarded sequence, and is defined in the corresponding section below.

Table 14: Syntax for statements

UnguardedSequence	=	‘{’ { *Statement* } ‘}’ .
GuardedSequence	=	‘{’ { *Guard* { *Statement* } } ‘}’ .
Sequence	=	*UnguardedSequence*
	|	*GuardedSequence* .
Guard	=	‘|’ [*Rvalue*] ‘|’ .
Statement	=	*Sequence*
	|	*Assert* [‘;’]
	|	*Assignment* ‘;’
	|	*Case*
	|	*Declaration* ‘;’
	|	*Exit*
	|	*ForAny*
	|	*Invocation*
	|	*Loop*
	|	*Select*
	|	*Signal* .

8.3.2. Assert

Assert = **assert** ‘(’ *Rvalue* [‘,’ *Rvalue*] ‘)’ .

When an **assert** statement is executed, the first *Rvalue*, which should be a Boolean expression, is evaluated. If it is **true**, the **assert** statement has no effect; otherwise, the program fails with the message “Assertion failure” and the coordinates of the **assert** statement.

If the second *Rvalue* is present, it should be a *Text*, and its value will be appended to the error message.

8.3.3. Assignment

Table 15: Syntax for assignments

Assignment	=	*Lvalue AssignOp Rvalue* .
AssignOp	=	‘:=’
	|	‘+=’
	|	‘-=’
	|	‘*=’
	|	‘/=’
	|	‘%=’
	|	‘//=’
	|	‘<<=’
	|	‘>>=’
	|	‘>>>=’ .

Assignments are conventional, as shown in Table 15, with $x \oplus{=} y$ abbreviating $x := x \oplus y$ for various operators $\oplus$ (Section 7.3.5).

The syntax is only an approximation of the way in which assignments are actually treated:

- If the *Lvalue* preceding the *AssignOp* is a simple identifier or a subscript expression (e.g., $a[i]$), then any assignment operator is allowed.
- If the *Lvalue* preceding the *AssignOp* is a field selector (e.g., $p.f$), then the only assignment operator allowed is `:=`.

The statement $L := R$ is type correct if $\mathcal{T}(R) \leq \mathcal{T}(L)$.

The statement $L \mathrel{+}= R$ is type correct if $L + R$ is type correct and $\mathcal{T}(L + R) \leq \mathcal{T}(L)$. Similarly for other operators.

8.3.4. Case

Case = **case** [*Rvalue*] *GuardedSequence* .

There are two forms of **case** statement, depending on whether the *Rvalue* is present.

If there is no *Rvalue*, the guards must be *Boolean* expressions. They are evaluated in turn until a true guard is found. The corresponding sequence is executed and the **case** statement terminates. If none of the guards is **true**, the statement has no effect.

One way to ensure that a **case** statement always has an effect is to use an empty guard as the final case.

The other form of **case** statement has an *Rvalue* immediately after the keyword **case**. This *Rvalue* is evaluated once, and its value is compared with each of guards. Let V be the value following **case** and G be a guard. The **case** statement is type correct if $V = G$ is type correct or if G is empty (and therefore considered **true**).

Examples Here are examples of both kinds of **case** statement.

```
case
{
  |n < 0|  s := 'negative';
  |n ≤ 100|  s "= 'small';
  ||  s := 'large';
}
```

```
case colour
{
  |red|  p.stop
  |amber|  p.slowdown
  |green|  p.go
  ||  p.ignore
}
```

8.3.5. Declaration

Declaration = { iden }‘,’ [‘:’ [‘+’ | ‘-’ | ‘@’] *Type*] [‘:=’ *Rvalue*] .

A declaration may introduce a variable, channel, or port.

A declaration may introduce several identifiers, separated by commas. Some contexts allow only one variable to appear in a declaration.

► Earlier versions allowed the keyword **alias** to appear in a declaration, indicating that the identifier was a reference rather than a value. This feature has been removed.

A declaration that does not contain ‘+’ or ‘−’ introduces a variable or channel. A declaration with ‘+’ introduces a server port, and a declaration with ‘−’ introduces a client port.

Declarations with ‘@’ are explained in Section 12.

If ':=' is present, the *Rvalue* is evaluated and the result becomes the initial value of the variable, channel, or port. A declaration containing ':=' is called an *initialized declaration* and the final *Rvalue* is called the *initializer*.

The initialized declaration v: $T := e$ is type correct if $\mathcal{T}(e) \leq T$.

► If there is no initializer, the declaration has no effect at run time. Using a variable before assigning a value to it is an error. The current (May 2011) version of UDC does not detect this error.

8.3.6. Exit

Exit = **exit** .

Executing an **exit** statement within a *Sequence* causes immediate termination of the sequence.

Although **exit** statements are allowed in any *Sequence*, they are intended mainly to execute **select** statements (see Section 8.3.10). Using an **exit** statement in a top-level *Sequence* (i.e., the *Sequence* that is the body of a routine or process) is an error.

8.3.7. For and Any

ForAny = (**for** | **any**) { *Comprehension* }$_{\textbf{also}}$ [**such that** *Rvalue*]
Statement [**else** *Statement*] .

Comprehension = '(' *Assignment* ';' *Rvalue* ';' *Assignment* ')'
| iden **in** (**domain** | **range**) *Rvalue*
| iden **in** *Type* .

A *Comprehension* declares a *controlled variable* and defines a set of values for it. The scope of a controlled variable starts with its declaration and ends at the end of the **for**/**any** statement. Assignment to controlled variables is not allowed.

A **for** loop executes the statements in the *Sequence* for each value of the variables declared in the comprehensions. A **for** loop must not have an **else** clause.

An **any** loop executes the statements in the *Sequence* at most once; the body is executed for the first values of the variables that satisfy the criteria. If the criteria are such that the *Sequence* is never executed, the *Sequence* following **else** is executed.

Many variants of the first form of *Comprehension* were considered and rejected. The proposed form is familiar from C, C++, Java, and other languages, and is also clear, general, unambiguous, and fairly concise. It consists of three parts:

- An *Assignment* that declares and initializes a controlled variable. A type declaration is allowed but not required, because the compiler can normally infer the type of the controlled variable from the assigned value.
- An *Rvalue*, which is a Boolean expression giving the condition for executing the loop body.
- An *Assignment* to the controlled variable that prepares it for the next iteration.

Comprehensions of the second kind, with **domain** and **range**, may be used with *Text*s, arrays, and maps. In each case, **domain** binds the controlled variable to the indexes of the structure, and **range** binds the controlled variable to the values of the structure.

Comprehensions of the third kind are defined by a type that must be an enumeration. The controlled variable assumes each value of the enumeration in turn.

Several comprehensions may be combined with **also**. The result is not a nested loop, but just several controlled variables that are stepped together. The loop terminates as soon as any of the continuation conditions becomes **false**.

If there is a **such that** clause, the body of the loop is executed only for values of the controlled variables for which the *Rvalue* is **true**.

It would be possible to eliminate the **such that** clause an use guards instead. Instead of writing

for C **such that** B { S }

we could write

for C { $|B|$ S }

The **such that** clause, however, was introduced primarily for **any** statements. The statement

any C **such that** B { S } **else** E

executes as follows:

- Values v of the comprehension C are generated in turn.
- If $B(v)$ is **true**, S is executed, and the statement terminates.
- If all values of C have been generated and S has not been executed, then E is executed.

If we were to write this statement as

any C { $|B|$ S } **else** E

it would either behave differently, because "$|B|$ S" would be executed as "**if** B **then** S", or we would have to give "$|B|$ S" a different semantics in this context.

If there is an **else** clause, it is executed only if the comprehension is exhausted without the body of the loop being executed.

Examples The most common form:

```
for (i := 0; i < 5; i += 1)
{
  -- values of i are 0, 1, 2, 3, 4
}
```

Going both ways:

```
txt := "amanaplanpanama";
len := size(pal);
pal := true;
for (i := 0; i ≤ len/2; i += 1) also (j := len; j ≥ len/2; j -= 1)
{
  | txt[i] <> txt[j]|  pal := false;
}
```

Using **domain** and **range** for linear search. The program writes `"11 13"`. If **for** was replaced by **any**, it would write `"11"`.

```
a := { 2, 3, 5, 7, 11, 13 };
for i in domain a also v in range a
{
  |v > 7| write(i);
}
```

Using **exit** to find an integer square root:

```
r: Word := 0;
for (n := 1; true; n += 1)
{
  |n * n > 100| r:= n; exit
}
```

Looping through an enumeration:

```
Vehicle = enum { bicycle, car, bus, boat, plane }
for v in Vehicle
{
   s += Word(v)
}
```

8.3.8. Invocation

Invocation = iden '(' { *Rvalue* }',' ['->' { *Lvalue* }','] ')' .

The *Invocation* syntax is used to invoke routines (i.e., functions and procedures), processes, and cells. Although the syntax is formally the same, different rules apply to each case.

Invoking routines Routines are described in Section 9. In the calling statement, arguments before the arrow are inputs (i.e., values sent to the routines) and arguments after the arrow are outputs (i.e., results returned by the routine). Inputs must be rvalues and outputs must be lvalues.

The arrow is required if there are no inputs but may be omitted if there are no outputs.

If a routine has exactly one output, the call may be either a statement or an expression, as shown by the following two snippets, which are equivalent.

```
x: Real := exp(10)
```

```
x: Real;
exp(10 → x)
```

Invoking processes Processes are described in Section 11. Since processes do not have outputs, their invocations may not contain an arrow.

The arguments that may be passed to a process are constants, variables, and ports.

If the process parameter is a port, the corresponding argument must be a channel. Unlike mE, Desi does not place any constraints on the number or direction of ports that can be attached to a channel.

Invoking cells Cells are described in Section 12. Since cells do not have outputs, their invocations may not contain an arrow.

The arguments that may be passed to a cell are constants and channels.

An argument corresponding to a constant must be an rvalue. An argument corresponding to a channel must be a channel or port.

8.3.9. Loop

Loop = **loop** { *Declaration* }‘,’ *Statement* .

Declarations are executed before entering the *Sequence* for the first time. The scope of the variables declared is the *Sequence*; these variables are not accessible after the statement has terminated.

After the declarations have been executed, the *Sequence* or *GuardedSequence* is executed repeatedly.

A **loop** statement terminates when an **exit** statement has been executed or when all of the guards of the *GuardedSequence* are **false**. These are not mutually exclusive: a **loop** statement may have a *GuardedSequence* as the loop body, and branches may contain **exit** statements.

Example Three loops that perform the same computation are shown below. The version on the left exits when its single guard becomes **false**; the centre version uses an explicit **exit**; and the version on the right uses a *Sequence* with a **case** statement for termination. For this particular problem, the version on the left is the simplest. However, the other forms may be more appropriate for loops of different kinds. For all three versions, *sum* is in scope after the loop terminates but *i* is not.

```
sum := 0;
loop i := 0;
{
  | i < MAX|
      sum += i;
      i += 1;
}
```

```
sum := 0;
loop i := 0;
{
  | i = MAX|
      exit;
  ||
      sum += i;
      i += 1;
}
```

```
sum := 0;
loop i := 0;
{
  case
  {
    | i = MAX| exit
  }
  sum += i;
  i += 1;
}
```

8.3.10. Select

Select = [**loop**] **select** [**policy** iden ‘;’] { *Declaration* }‘,’
GuardedSequence .

A **select** statement makes a non-deterministic choice of communication. In this context, “Non-deterministic” means that the effect of the statement cannot be determined statically, because the action taken depends on the run time state of other processes.

Writing “**loop**” before **select** indicates that the **select** statement should be executed repeatedly, as described below, until an **exit** statement has been executed.

The identifier following **policy** must be one of the values of the enumeration *Policy* defined by

Policy = **enum** { *Fair*, *Ordered*, *Random* }

If there is no policy clause, the compiler assumes **policy** *Ordered*;.

If declarations are present, they are executed once only before the *Sequence* is executed for the first time. The scope of the declared variables is the *Sequence*.

The compiler recognizes two kinds of **select** statement: in the first kind, every guard checks the same port; in the second kind, guards may check different ports. The difference is transparent semantically (i.e, the programmer does not have to worry about it), but the first kind of **select** is likely to be more efficient to execute than the second.

► The current (May 2011) version of UDC does not make any distinction between the two kinds of **select** statement.

Each branch of a **select** statement must contain at least one communication (see also Section 14). A **select** branch is *enabled* if its guard is **true** and the channel for its *first communication* is ready to transfer data.

Implicitly, each guard includes a query that checks the status of the first communication. For example, if $p.a$ is the first communication in

select
{
  $|\,b|$ ... $p.a$ ...
}

the compiler generates code corresponding to

select
{
  $|\,p?a$ **and** $b|$ ... $p.a$ ...
}

The following rules define an ordering "earliest". The first communication is the earliest communication in a **select** branch.

- If there is only one communication in the branch, it is the earliest.
- In the sequence $S_0;\ S_1;\ \ldots;\ S_n$, S_i is earlier than S_{i+1}.
- In the assignment $L := R$, R is earlier than L.
- In the expression $L \oplus R$, in which $\oplus$ stands for any binary operator, L is earlier than R.

In each branch of the following (meaningless) **select** statement, $p.a$ is the first communication:

select
{
  || $p.a := 77$
  || $p.b := p.a$
  || $w := 45;\ p.a := 77;\ p.c := 88$
  || $w := p.a + 45$
  || $w := p.a + p.b;\ p.c := 77$
}

Examples The following snippet shows the implementation of a simple server, **select**ing on a single port:

```
SP = protocol { c: Char; w: Word; t: Text ) }
 ....
p: +SP;
loop select
{
   || write('Character ' // p.c)
   || write('Word ' // p.w)
   || write('Text ' // p.t)
}
```

In this snippet, process *buffer* is a "filter" that stores up to 10 *Word*s. The **select** statement handles both ports.

```
WP = protocol { w: Word }
 ....
buffer = process inp: +WP; out: −WP
{
   store: Word[10];
   ix: mod 10 := 0;
   ox: mod 10 := 0;
   usedCount: Word := 0;
   loop select
   {
      |usedCount < 10| store[ix] := inp.w; ix += 1; usedCount += 1;
      |usedCount > 0| out.w := store[ox]; ox += 1; usedCount −= 1;
   }
}
```

8.3.11. Signal

Signal = iden { '[' *Rvalue* ']' } '.' *Rvalue* .

A signal is a message that transmits no information other than its own existence. Signals may be used to synchronize two processes by ensuring that they both recognize the signal at the same time or to allow one process to request another process to perform a particular action.

Example The following extract shows how a *stop* signal could be used to terminate a loop.

```
prot = protocol { ...; stop }

server = process p: +prot
{
   loop select
   {
    ....
   || stop; exit
   }
}
```

9. Routines

In conventional usage, *procedures* are invoked for their side-effects and do not return values and *functions* are invoked to compute a value that is returned to the caller. In practice, the distinction is less sharp: depending on the language, functions may have side-effects and procedures may return values via reference parameters.

9.1. Goals

Desi does not distinguish procedures and functions. Instead, it provides only *routines*, which can play either role. A routine definition has a parameter list of the form $i_1, i_2, \ldots \rightarrow o_1, o_2, \ldots$ in which the i_k are inputs and the o_k are outputs. A routine call has an argument list of the form $e_1, e_2, \ldots \rightarrow v_1, v_2$ in which the e_k are expressions that are evaluated and passed to the routine and the v_k are variables to which results should be assigned. The arrow may be omitted if there is nothing to its right. If there is exactly one output parameter, the call may be used in an expression; the output variable is omitted.

9.2. Alternatives

We briefly considered omitting routines altogether. Apart from the fact that programmers expect simple abstraction mechanisms for statements and expressions, some kind of routine is needed for linking to code written in other languages. Therefore, Desi includes routines.

An alternative possibility would be to merge the concept of process and routine, which would certainly simplify the syntax of the language. However, the gains would be largely illusory because the implementation would have to treat processes and routines differently. Programmers concerned with efficiency would then have to reverse-engineer the implementation, and the advantages of "simplifying" would be lost.

9.3. Design

RoutineDefinition	=	(iden | **op** (*UnaryOp* | *BinaryOp*)) '=' [**implicit**] **routine** *Parameters* (*Sequence* | '=' Intrinsic ';' | **external** TextLiteral ';') .
Parameters	=	{ *Declaration* }‹,› ['->' { *Declaration* }‹,›] .
Declaration	=	{ iden }‹,› [':' ['+' | '-' | '@'] *Type*] [':=' *Rvalue*] .

The syntax for routine definitions provides for a number of different forms. The alternatives are summarized below, and examples follow.

- The name of a routine is either an identifier, used in the conventional way, or the keyword **op** followed by an operator symbol. The second kind of routine is invoked by using the operator in the usual way, either as prefix or infix. When the operator might be unary or binary (e.g., $-$), the number of parameters disambiguates.
- All routines have a list of input parameters followed by an arrow and a list of output parameters. Both lists may be empty; if the output parameter list is empty, the arrow is omitted.
- The name of a routine may be a type name, T. In this case, the routine acts as a constructor (in the OOP sense) for type T. If there is a single input parameters, the routine is effectively a conversion. This kind of routine must have a single output value of type T.

- A routine may be declared as **implicit**, which means that calls to it may be inserted into the code by UDC during code generation.

  The qualifier **implicit** is allowed only if the name of the routine is a type and there is exactly one input parameter and one output parameter. Implicit routines are used to perform type conversions transparently to the programmer.

- Four kinds of body are recognized for routines:

  1. The conventional body is a sequence of statements enclosed in braces.
  2. An empty body (i.e., "{}") is used for routines that will never actually be called because the compiler generates in-line code.
  3. An *Intrinsic* consists of `$` followed by letters: e.g., `$abcd`. The letters correspond to a DIL mnemonic.

  ▶ Following recent (April 2011) changes to the DIL specification, intrinsics that perform conversions have body `$c`. A conversion must have signature $T_1 \rightarrow T_2$, with an input parameter of type T_1 and an output parameter of type T_2.

  4. The form "**external** TextLiteral" declares that the routine has been implemented as an external function with the given name. The *Text* literal specifies the language in which the external routine is implemented.

  ▶ See KIWI 110426 and Section 16 for further discussion.

Desi supports overloading of routine names. Different functions may have the same name, provided that their input parameters differ in number or type. Overloading is implemented as follows:

- A routine declaration *matches* an invocation if:
  1. the names of the declared and invoked routines are the same; and
  2. the number of parameters of the declared routine and the number of arguments of the invoked routine are the same; and
  3. each input argument type can be implicitly converted to the corresponding parameter type.
- Match M_1 is *closer* than match M_2 if M_1 requires fewer conversions than M_2.
- The set of routines that match the invocation is called the *match set*, M. Let $|M|$ be the number of elements in M.
  - If $|M| = 0$, the compiler reports an error.
  - If $|M| > 0$, the compiler uses the closest match. In the event of a tie, the choice is arbitrary.[5]
  - If $|M| > 1$ and the compiler flag "report warnings" is set, the compiler reports all routines in the match set.

[5] The choice actually depends on the order in which the routines are declared, but programmers should never make assumptions based on this.

Examples

- A conventional routine for squaring an instance of the basic type *Word*:

  sqr = **routine** *x*: *Word* → *xs*: *Word* { *xs* := *x* ∗ *x* }

- A routine for multiplying values of the basic type *Word*. Since the compiler knows how to do this, the body is an empty sequence.

  op∗ = **routine** *x*: *Word*; *y*: *Word* → *z*: *Word* {}

- A routine for rounding a *Real* value to produce a *Word* value, implemented by the DIL opcode `rnd`:

  round = **routine** *r*: *Real* → *w*: *Word* = *$rnd*;

- A routine for constructing a value of the user-defined type *Complex*, implemented as an external function called *make_complex*:

  Complex = **routine** *x*: *Real*; *y*: *Real* → *z*: *Complex* = "make_complex";

- Implicit conversion from *Real* to *Complex* (presumably, $z = x + 0i$). With this function in scope, any *Real* value will be converted in a context that expects a *Complex* value. This function overloads the previous one.

  Complex = **implicit routine** *x*: *Real* → *z*: *Complex* = "real_to_complex";

- Subtraction for a user-defined type *Vector*, implemented by an external function:

  op− = **routine** *u*: *Vector*; *v*: *Vector* → *w*: *Vector* = "vector_subtract";

- Negation for *Vector*s. Since there is only on input parameter, the operator "−" is unary. This function overloads the previous one.

  op− = **routine** *v*: *Vector* → *mv*: *Vector* "vector_negate";

10. Input and Output

This section describes input and output (I/O) in the traditional sense: reading from the keyboard, writing to the screen, and performing both operations on files. Section 14 discusses communication using ports and channels.

10.1. Goals

There should be a simple syntax for accessing external media, such as files. The syntax and semantics should be consistent with the rest of the language.

I/O features should be minimal. For example, rather than providing elaborate parsing and formatting functions, we provide only the ability to read and write *Text* values. Parsing and formatting capabilities should be provided by the *Text* type itself, which thereby serves as a "memory stream".

10.2. Alternatives

mE used somewhat baroque syntax for input and output (I/O). In Desi, all I/O is performed by routines that belong to an I/O library. Thus the syntax is standard and the compiler does not have to know anything about I/O.

I/O is usually classified as *character* or *binary*. The differences are:

- Character data includes only recognized characters. Most characters correspond to a particular symbol, but a few are interpreted in special ways when data are displayed: e.g., newline and tab characters. Binary data consists of arbitrary bit strings, usually packaged into units of a fixed size, such as bytes.
- Some operating systems process the two kinds of external data differently. For example, early Microsoft OS converted "new line" to "carriage return, line feed", with disastrous consequences for binary data. Fortunately, most of these archaisms seem to have quietly died off.

Desi performs I/O operations under the assumption that the data are *Text*s.

Some languages provides *text streams* (or *string streams*). Text streams are used because they provide a convenient way of using the parsing and formatting capabilities of the I/O library routines on internal data. As mentioned above, the type *Text* will provide a rich collection of parsing and formatting routines, making text streams unnecessary.

10.3. Design

Data external to the program takes the form of a *stream*. Types and operations for streams are summarized in Table 16 and described briefly below.

Table 16: I/O operations

type *Stream*;	The type of streams
StreamMode = **enum** { *In*, *Out*, *Append* }	Modes in which a stream may be opened
StreamState = **enum** { *OK*, *OpenError*, *ReadError*, *WriteError* }	Possible states of a stream
open(*p*: *Text* → *s*: *Stream*)	Link stream *s* to path *p* for reading
open(*p*: *Text*, *m*: *StreamMode* → *s*: *Stream*)	Link stream *s* to path *p* with mode *m*
close(*s*: *Stream*)	Close stream *s*
state(*s*: *Stream* → *h*: *StreamState*)	Returns the state of stream *s*
eof(*s*: *Stream* → *b*: *Boolean*)	Set *b* according to whether *s* has more data
read(*s*: *Stream* → *t*: *Text*)	Read from stream *s* and store the result in text *t*
write(*s*: *Stream*, *t*: *Text*)	Write the text *t* to the stream *s*

Desi has a basic type *Stream*. A *Stream* variable (or "stream" for short) is a structure associated with external media such as a disk file.

The type *Stream* is isolated in the type system: there are no conversions to or from *Stream*. The only way to assign a value to a *Stream* variable is to invoke *open*. (However, *Stream* variables can be copied.)

Internally, a *Stream* is represented as a *Word*. Thus *Stream*s do not appear in DIL code: instead, the JIT maintains a mapping between *Word* values (which it chooses itself) and file objects.

The routine *open* establishes the connection between a stream and a file. The arguments passed to *open* are the stream, the path to the file, and an optional mode. If the mode is present, it must be a value of the enumeration *StreamMode*.

The routine *close* breaks the association between the stream and the file after performing any necessary finalizing actions on the file.

The call *state*($s \rightarrow h$) checks whether s is in a healthy state and sets h to a value of *StreamState*. This call should always be used to check that a file was opened successfully, and may also be used after reading or writing.

The call *eof*($s \rightarrow b$) checks if there is more data available in the input stream s, assigning $b :=$ **true** if there is none.

If *state*(s) $=$ *OK*, then *eof*(s) may be **true** or **false**. If *state*(s) $<>$ *OK*, then *eof*(s) is undefined, as are the effects of calling other I/O routines. However, *close*(s) will close s unless things have gone badly wrong.

The call *read*($s \rightarrow t$) reads one line of *Text* from stream s and stores the result in t. If there is no more data in s (that is, *eof(s)* is **true**), the effect is $t :=$ ''.

The call *write(s, t)* writes the *Text* t to the stream s.

► We might want a richer set of routines for reading and writing. E.g., *readline*, *readfile*, etc.

Example The following snippet copies `in.txt` to `out.txt` and writes the number of lines copied to standard output.

```
loop
  i: Stream; open('in.txt' → i);
  o: Stream; open('out.txt', Out → o);
  lines: Word := 0;
{
  |not eof(i)| write(o, read(i)); lines += 1;
}
write('Copied ' // lines // ' lines.\n')
```

The first two lines of the loop could alternatively be written like this:

```
i: Stream := open('in.txt');
o: Stream := open('out.txt', Out);
```

11. Processes

11.1. Goals

A process should be a self-contained entity. This goal has several ramifications.

- A process is responsible for the resources that it owns. When it dies, it must release any resources that it is holding. One resource that all processes own is a region of memory: thus each process is responsible for its own garbage collection.

- mE did not allow both ends of a channel to be processed with a **select** statement, a restriction inherited from Joyce (Brinch Hansen 1987). This restriction is unacceptable for large-scale software development, because it implies that know things about a process that are not part of its interface. (Of course, we could require that a port processed with **select** advertise this fact, making selection part of the process's interface. However, this solution does not seem particularly attractive.)

11.2. Alternatives

Process structure has worked quite well in mE, and there is no pressing need to consider alternatives.

The main novelty in Desi is greater activity at run time. A process may be started before all of its ports have been connected to channels. No structural change is needed, but we will need additional syntax for checking the status of ports. In a **select** statement, unconnected ports will never yield **true** guards.

11.3. Design

ProcessDefinition	=	iden `'='` **process** *Parameters* (*Sequence* | **external** TextLiteral `';'`) .
Parameters	=	{ *Declaration* }`','` [`'->'` { *Declaration* }`','`] .
Declaration	=	{ iden }`','` [`':'` [`'+'` | `'-'` | `'@'`] *Type*] [`':='` *Rvalue*] .

As soon as a process has been constructed, it starts executing its *Sequence*. When the sequence terminates, the process releases any resources that it has allocated, including the memory that it is using, and dies.

Most processes are organized around a loop that performs computations and communicates via the processes's ports. Even these processes may eventually die if they find there is nothing left to communicate with.

The parameters of a process have standard syntax; however, a process cannot return results and so the parameter list cannot contain an arrow.

The parameters of a process are constants, variables, and channels. The following snippet illustrates.

```
doit = process
  max: Word;
  alias table: Word indexes Text;
  p: +SP;
  q: -CP;
{ .... }
```

Since *max* is not qualified with **alias**, its value is passed in when the process is created but cannot be altered thereafter. On the other hand, *table* is a variable declared in the enclosing cell; it can be altered by this process and perhaps also by other processes in the cell. This process acts as a server with protocol *SP* for port *p* and as a client with protocol *CP* for port *q*.

► The body of a process is either a *Sequence* or the keyword **external** followed by a *Text* literal *L*. This indicates that the process is implemented by external code written in language *L*.

► This example is no longer legal, because **alias** has been abolished.

12. Cells

Cells provide the structuring mechanism for Desi programs. Since a cell may contain anything from a byte to a multi-continental system, cells provide something of a fractal quality to Desi programs.

12.1. Goals

The purpose of cells is to encapsulate structures of arbitrary complexity and to provide simple interfaces to them (in the form of ports and protocols).

12.2. Alternatives

The big decision is whether cells and processes should be merged into a single construct. The strongest motivation for doing that is that cells and processes are already quite similar and could fairly easily be forced into the same mould. However:

- Distinguishing cells and processes provides separation of concern: processes define actions; cells define structure.
- The rule that a processes has a single thread of control is easy to understand and is an aid to reasoning about programs. If cells and processes were the same, this rule would be lost, because processes would be allowed to contain sub-processes and shared variables.
- At present, cells precisely define the domain of shared data (only the top-level processes within a cell can access the cell's data, and they must explicitly declare their intention of doing so). If cells and processes were the same, different, and probably more complex, rules would be required to determine the scope of shared data.

We have sometimes discussed the possibility that shared variables could be viewed as cells. The following, rather tentative proposal, develops the discussion of §1.11 of KIWI 100131.

- The declaration 'v: T' in a process introduces a variable of type T in the usual way. The implementation allocates space for an instance of T, probably on the current process's stack, and v is a reference to (i.e., address of) this instance. Access to the variable is limited to the process.
- The declaration 'v: T' in a cell also introduces a variable of type T in the usual way. The implementation allocates space for an instance of T in the cell's memory space, and v is a reference to (i.e., address of) this instance. The address of this variable may be passed as an **alias** to processes within the cell, and these processes can access the variable, with locks for writing, etc.

► This model is no longer viable, since **alias** is no longer part of the language.

- This is the proposed new feature: the declaration 'c: $@T$' introduces a *cell* of type T and provides a channel, c, that can be used to access it. The channel can be passed around, even sent to another process, and anyone who possesses it can access this particular instance of T by linking a port to the channel.

Two questions arise. Why bother? And how can we minimize the differences between an "ordinary" variable and a cell?

The answer to the first question is that we need to distinguish access by shared memory and access by communication. The reasons for this are explained in other documents going back to the earliest days of the Erasmus project and will not be repeated here.

We can begin to answer the second question with a rather simplistic approach. The compiler distinguishes lvalues and rvalues, generating writes and reads, respectively. The compiler can generate code as follows:

- If v is declared as 'v: T', the compiler uses standard addressing techniques.
- If c is declared as 'c: @T'', the compiler translates lvalues to $c.set$ and rvalues to $c.get$.

The second item suggests that the compiler should automatically associate a protocol

protocol { *get*: *T*; ↑*set*: *T* }

with each type T that is used as a cell, and that this is the protocol used for the channel c in the declaration 'c: @T'.

The mechanism described above is suitable only for simple types, for which the *get*/*set* interface is all that is needed. One of the main points of having cells as types, however, is to provide structured types with many components and an interface richer than just *get* and *set*. There must therefore be a means by which a programmer can define a cell with several components and a correspondingly rich interface. This issue is revisited (briefly) in Section 18.1.2.

► UDC does not yet implement the @ construct.

12.3. Design

CellDefinition	=	iden '=' **cell** *Parameters* (*Sequence* | **external** TextLiteral ';') .
Parameters	=	{ *Declaration* }',' ['->' { *Declaration* }','] .
Declaration	=	{ iden }',' [':' ['+' | '-' | '@'] *Type*] [':=' *Rvalue*] .

A cell contains constants, variables, channels, processes, and cells. As soon as a cell is constructed, it constructs its components and links them appropriately.

The parameters of a cell are constants, variables, and channels.

► The body of a cell is either a *Sequence* or the keyword **external** followed by a *Text* literal L. This indicates that the cell is implemented by external code written in language L.

13. Protocols

Protocols are to ports as types are to variables. By specifying both the type and allowed orderings of messages, protocols provide greater expressiveness than is typically available in other languages. In particular, protocols address a weakness of OOPLs—namely, the implementor of a class cannot control the sequence in which its methods are invoked.

Listing 1: A program that deadlocks

```
Prot = protocol { w: Word }
Proc = process p: −Prot; q: +Prot;
{
    p.w := "Hi!"; reply := q.w;
}
Cell = cell
{
    a: Prot; b: Prot c: Prot;
    Proc(b, c); Proc(c, a); Proc(a, b);
}
Cell()
```

13.1. Goals

Ideally, it should be possible to check at code generation time that all messages will eventually be delivered. Unfortunately, this goal may not be fully realizable, and all we can hope to do is get as close to it as possible. It is very likely that run time checks will be required in addition to static checks.

Listing 1 and Figure 1 illustrate the difficulty. Checking each channel reveals no problems. For example, channel a links a server (the second process) to a client (the third process), and transfers one message: `"Hi!"`. However, the three processes together deadlock, because each wants to send before it receives. Thus the port declarations alone are not sufficient for safety checking: the checker must know that the first process sends on b before receiving on c, etc.

13.2. Alternatives

The protocol operators '?' (zero or one) and '+' (one or more) of mE were not used much and have been dropped from Desi. The operator '->' has been introduced: '$a \rightarrow b$' is synonymous with '$a;\ \uparrow b$'.

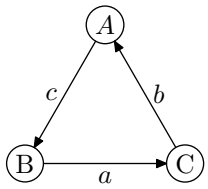

Figure 1: Processes that deadlock

A protocol may contain the name of another protocol. This idea was proposed in KIWI 100131 and was also used in NIL (Strom 1986). After defining

```
dessert = protocol { iceCream; fruit }
meal = protocol { meatAndPotatoes; dessert }
```

the protocol *meal* is equivalent to

```
meal = protocol { meatAndPotatoes; iceCream; fruit ) }
```

The formalism allows the definition of recursive protocols. For example, a message satisfies protocol *balance* if it consists of x enclosed in balanced brackets, indicated by the signals L and R:

$$balance = \textbf{protocol}\ \{\ x\ (\ L;\ balance;\ R\ )\ \}$$

However, Desi does *not* allow either direct or indirect recursion, for various reasons:

- Protocols that name other protocols can be textually expanded by the compiler, as we did with *meal* above. After expansion, the compiler can check protocol satisfaction as in MEC. Recursion gives rise to unbounded expansion, which cannot be checked in this way.
- More generally, recursion raises the level of the language. Without it, protocols correspond roughly to regular languages, in which most interesting properties (e.g., language subset, equality, etc.) are easily decided. With recursion, protocols form a context-free language, and many properties become undecidable.[6]
- A process that implements a simple protocol can do so by matching the protocol operators to coding structures: sequence, choice, repetition. A process that implements a recursive protocol must use a stack machine.
- ► The decision to remove dynamic specifications from protocols makes most of this discussion moot, anyway.

13.3. Design

ProtocolDefinition = iden '=' **protocol** '{' { ['^'] iden [':' iden] }',' '}' .

A protocol is a list of field declarations. A declaration may be:

- A simple name, s, denoting a signal.
- A name and a type identifier: $q : T$, denoting a query with the given type, sent from client to server.
- A name flagged with an arrow (caret in program source) and a type identifier: $\hat{}r : T$, denoting a reply with the given type, sent from server to client.

13.3.1. Parameterized Protocols

The syntax for a protocol definition, consistently with other forms of definition, allows a protocol to have parameters. The intention is that these parameters should be types, as in the following example.

$$PP = \textbf{protocol}\ T\ \{\ set\colon T;\ \uparrow get\colon T\ \}$$

A parameterized protocol is instantiated by providing an actual argument:

$$usePP = \textbf{process}\ p\colon +PP(Integer)\ \{\ ....\ \}$$

Parameterized protocols are envisaged as an abbreviation; the compiler treats them like macros. If and when protocols become first-class (so, e.g., a process can send a channel along with its protocol), we will have to review the meaning of parameterized protocols.

► The current version of UDC does not recognize parameterized protocols.

[6] All this seems vaguely plausible, but I haven't done any formal checks or proofs.

14. Communication

In general, Desi follows mE in communication. Processes have ports that are connected to channels. Channels are bidirectional and are associated with a protocol that defines the types and sequence of messages.

However, Desi is intended to be significantly more flexible than mE in its implementation of communication.

14.1. Goals

Communication is as important in Desi as method invocation is in OOP languages. Communication must be represented simply in source code and implemented efficiently at run time.

Communication is non-deterministic in that the order in which a process serves its clients is not known until run time. A process indicates its willingness to communicate with any of several fields of a port, or with several ports; this is called *selection*.

mE had the restriction, inherited from Brinch Hansen's language Joyce, that only one of the processes involved in communication could perform selection. This is a serious restriction, because static checking requires the compiler to have access to the source code of all processes (preventing independent compilation) and dynamic checking has non-trivial overhead. Desi removes this restriction, allowing any process to use selection. The cost is a loss of efficiency that we will try to minimize.

► mE also required a channel to be connected to exactly one input port and one output port. Desi retains this restriction because we have not yet found an appropriate semantics for a channel connected to more than two processes. KIWI 100131 introduced the idea of an *interaction*, which is a sequence of related simple communications. We could specify, for example, that once a process had established a link to another process sharing the channel, an interaction must be completed before the link is broken. However, work remains to be done in this area.

A process should not need to know anything about the location of the processes that it communicates with. A process communicates via a channel; the channel may be a single component in the current memory space, or it may be split over two memory spaces. Figure 2 shows processes P and Q using a channel, C, to communicate in a single memory space, M. Figure 3 shows P in memory space M_1 communicating with Q in memory space M_2. The channel is split into two parts, C_1 and C_2. Communication over the network is achieved by two links, L_1 and L_2, created to implement a virtual channel, $C_1L_1L_2C_2$, that appears to the processes to be identical to the simple channel, C, in Figure 2.

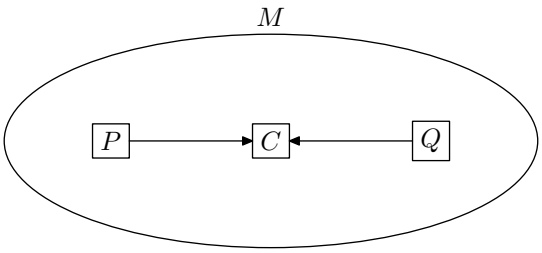

Figure 2: Communication in a single memory space, M

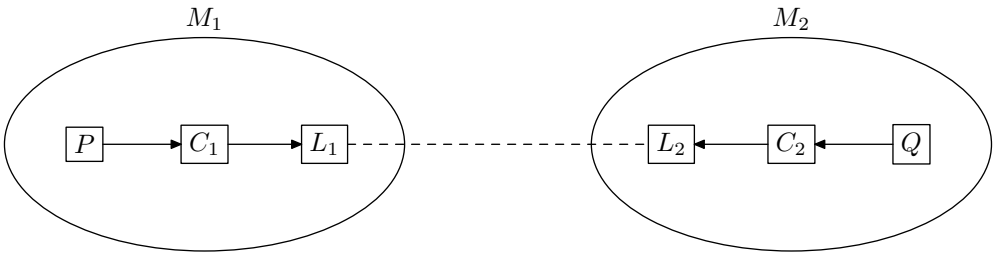

Figure 3: Communication between two memory spaces, M_1 and M_2

14.2. Alternatives

From mE, we adopt the convention that communication through field f of port p is denoted as $p.f$, used as an lvalue for sending and as an rvalue for receiving.

In CSP, $p!f$ denotes sending and $p?f$ denotes receiving. Hoare mentions the convention above as a possible alternative, but rejects it.

In mE, we found it necessary to include a field named, for example, *stop* in a protocol so that processes could be terminated by a construct of the form

$pr =$ **process** p: $+Prot$ |
  loopselect
  ....
  || $p.stop$; **exit**
end

In Desi, this mechanism is built in. The implementation implicitly prefixes each protocol with *start* and postfixes it with *finish*. Receiving a *finish* signal terminates a **loop select** statement. Further details are given below, in Section 14.3.

► Currently, UDC generates DIL on the basis of communications specified by the Desi programmer. In particular, it does not insert *start* or *finish* signals. We need further design effort to decide how this should be done and whether it is the responsibility of UDC or JIT.

14.3. Design

► At the time of this revision (June 2011), Desi communication is a hot discussion topic. Consequently, the contents of this section should be taken with a generous pinch of salt.

The requirements for a process, P_1, to communicate with another process, P_2, are as follows:

- P_1 must have a port, π_1.
- Process P_2 must have a port, π_2.
- One port must be a server (declared with '+') and the other a client (declared with '−').
- There must be a channel, χ. (In an implementation, π_1 and π_2 might be references to χ.)
- The ports and the channel must agree on a protocol. (The actual situation is slightly more complicated—see below.)
- Requests (unflagged fields in the protocol) are sent from the client to the server.
- Replies (fields flagged with ↑ in the protocol) are sent from the server to the client.

If the following have been declared

P = **protocol** { q: *Word*; $\uparrow r$: *Word* }
server = **process** s: $+P$ { ... }
client = **process** c: $-P$ { ... }

then in *server* it is legal to write

x: *Word* := $s.q$;
$s.r$:= ...

and in *client* it is legal to write

$s.q$:= ...
y: *Word* := $s.r$;

Suppose that the programmer writes a protocol, P. The implementation adds implicit signals *start* and *finish*, so that the actual protocol is

start; P; *finish*

(Internal symbols are used, rather than the actual names *start* and *finish*, to avoid conflicts with user-defined names.)

A client proceeds as follows:

1. Initialize the port, p, to be used for protocol P.
2. Send $p.start$.
3. Execute code corresponding to P.
4. Send $p.finish$.
5. Perform any necessary finalization of p.

A server proceeds as follows:

1. Initialize the port, p, to be used for protocol P. The state of p is READY.
2. Wait for $p.start$. When it has been received, change the state of p to ACTIVE.
3. Execute code corresponding to P.
4. When $p.finish$ is received, change the state of p to CLOSED.
5. Perform any necessary finalization of p.

When all of the ports of a server have been closed, the server can destroy itself, since nothing can access it.

Example Assume that the protocol *Prot* has been defined as

Prot = **protocol** { i: *Word* }

Assume also that the client and server below have been connected. Both processes would run, terminating after 10 message have been sent:

```
client = process p: -Prot
{
  for i := 1; i ≤ 10; i += 1;
    p.i;
}
```

```
server = process p: +Prot
{
  loop select
  {
    |p?i| write(p.i // '\n')
  }
}
```

Listing 2: Selective logging

```
PortStatus = type enum { Null, Disconnected, Connected }
LogData = protocol { msg: Text }
Orders = process log: −LogData; ...
{
    unlogged: Word := 0;
    loop
    {
        ...
        case status( log)
        {
            |Connected| log.msg := "Done another one";
            || unlogged += 1;
        }
    }
}
```

14.3.1. Dynamic Connections

If connections can be made and broken dynamically, processes must have some way of determining the status of a port. There is more design work to be done but, for now, we assume the existence of a built-in function *status* that, when passed a port, returns a member of an enumeration. In Listing 2 a process sends messages to a logger if the logger is connected, otherwise counts unlogged messages.

14.3.2. Communication Between Processors

Suppose that the protocol *FAQ* has been declared by

$$FAQ = \textbf{protocol}\ \{\ query\colon\ Text;\ \uparrow answer\colon\ Text\ \}$$

and that the declaration

$$p\colon\ FAQ;$$

appears in a cell. The effect of the declaration is to create and start a channel. The channel is a process and, in its initial state, it is waiting for something to happen. The cell then creates a server and passes the channel to it by reference:

$$server(p);$$

The server implementation starts like this:

```
server = process p: +FAQ
{
    q: Text := p.query;
    ....
```

The effect of the assignment is that the server sends a request-to-receive signal to p and then waits for a response.

The cell starts a client and passes p to it by executing

client (*p*);

The client implementation starts like this:

client = **process** *p*: −*FAQ*
{
  p.*query* := "What is your name?";
  ....

The effect of the assignment is that the client sends a request-to-send to p and waits for a response.

Channel, p, now has two requests that match. "Matching" means:

1. the requests are associated with the same protocol, *FAQ*;
2. the requests are associated with the same field, *query*;
3. one request is for sending and the other is for receiving.

Channel p asks the sender for the address of the message to be sent, and passes that address to the receiver. The receiver copies the message into its own space; the communication is complete; and both processes resume.

There are two complications:

- The message may be large and/or complex. In general, what we have called "copying" above will actually be serialization, except for data that occupies one word or less.
- The two processes may be running on different processors in different memory spaces (as in Figure 3 on page 43). In this case, the channel is split into two parts, one in each space. Each part communicates with the broker for its processor. The channel does not pass the address provided by the sender directly to the receiver, which could not use it, but instead copies the message to the broker. The broker transmits the message over the network, and the other broker receives it and passes it to the channel, which passes it to the receiver.

Figure 4 shows a simplified interaction diagram for the two-processor case. The column headings are:

C : Client running on processor 1
P_i : Channel part in processor i
B_i : Broker for processor i
S : Server running on processor 2

The messages sent are:

1: The client sends a request-to-send signal with the data address, α.
2, 3, 4: The signal is sent across the network to the other part of the channel, P_2.
5: The server sends a request-to-receive signal. The channel detects a match.
6, 7, 8: The match event is sent back across the network.
9: The data at address α, shown as $[\alpha]$, is copied to the channel.
10: The client is allowed to resume.
11, 12, 13: The data is transferred across the network. The server resumes after it has received the data.

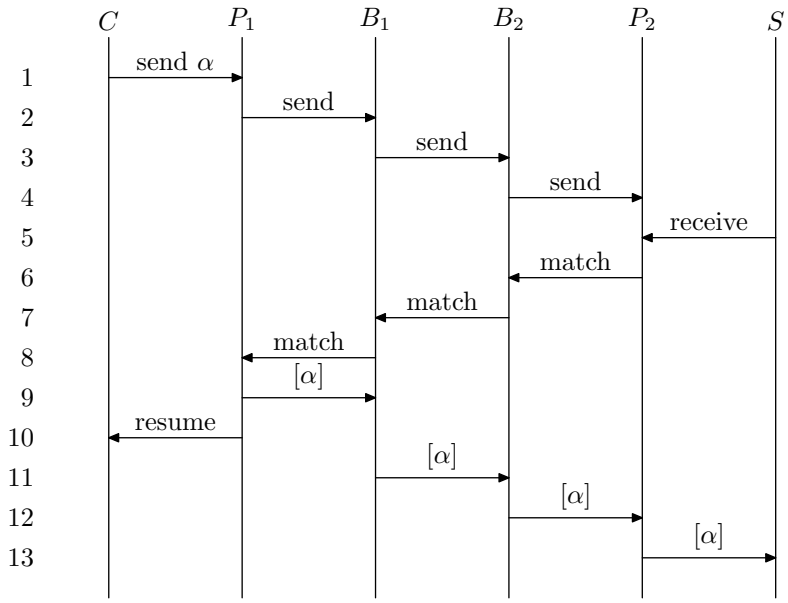

Figure 4: A communication scenario

14.3.3. Symmetrical Select

Suppose that two processes are communicating and that both are using **select** statements. Connection is established in the following way, as seen from one of the processes, P:

1. P sends a *ready-to-communicate* signal for each branch of the **select** statement. In general, these signals will go to several different channels.
2. Later, P may receive a *go-ahead* signal for one of its branches. It sends an *abort* message to the other branches and effects the transfer.
3. Alternatively, P may receive a *failed* message, in which case it starts the whole process over again.

This description is somewhat simplified: the complete algorithm can be found elsewhere (Grogono and Jafroodi 2010). The algorithm requires an ordering on processes, which is easily achieved by giving each process a UID.

Clearly, symmetrical **select** is going to be less efficient than simple communication. We anticipate that a clever combination of JITting and reflecting will enable Desi to generate the most efficient code possible for the particular situation.

15. Libraries

► Currently (May 2011) under development.

Desi will depend heavily on libraries, both existing libraries and libraries written especially for Desi. The compiler will implement essential, language-specific features of Desi but will otherwise be as simple as possible. This approach is intended to maximize the chances of achieving a useful system.

In this section, we consider only libraries written in Desi. Section 16 discesses libraries written in other languages.

15.1. Goals

The library system should provide easy access to existing libraries, such as the statndard C libraries. C has the advantage that it requires only a simple run time system (RTS). It would be more difficult to provide access to Java libraries, for example, because Java requires a more complex and specific run time environment.

It should be easy for users to write their own libraries. Ideally, users should be able to employ the language of their choice. In practice, they may have to use C or, of course, Desi, at least with the first implementations.

15.2. Alternatives

The library functions needed by an application can be linked statically or dynamically.

In a "classical" implementation, the compiler generates object code, the linker links the application object code with library object code, and the final result is an executable file. This approach has been discussed and rejected (KIWI 100125).

The alternative is a system that accepts source programs, interprets them or compiles them, and immediately executes them. The distinction between "interpreting" and "compiling" is not sharp. Few interpreters literally interpret source code; they usually do some analysis first, and may even construct an AST before executing anything. A JIT compiler resembles a compiler in that it generates machine code, but resembles an interpreter in that it does not do so until the code is needed for execution. Desi will be implemented with some form of interpreter or JIT compiler. We will refer to an implementation as "the interpreter".

This decision leaves two choices for libraries. Either the interpreter has all of the necessary libraries built into it, or libraries can be linked at run time. The first choice is undesirable for at least two reasons: first, the interpreter would be very large, not general enough for all purposes, or both; second and more importantly, the system would be hard to extend (the interpreter would have to be recompiled and relinked).

The above reasoning suggests that Desi should be an interpreter or JIT compiler that links libraries as needed during execution.

► During a discussion (May 2011), we noted that there are two approaches to interfacing with a large, existing library, such as the Windows API.

1. Desi programmers would be allowed to call library functions directly.

   This approach would probably be difficult to implement because of the differences in type systems between C and Desi. For example, the Windows API defines a large collection of types that would have to be duplicated in Desi — assuming that this was even possible.

2. Desi programmers would call wrapper functions that would, in turn, call library functions.

   This approach seems more practical. It allows for the possibility of exposing a simplified interface to the library.

15.3. Design

Library entities can be compiled on-the-fly, by JIT. They would be stored in source code form, introduced to the program with **import** directives, and compiled in the same was as any other entity, when needed.

It might be desirable to have pre-compiled libraries as well. (These will be needed in any case for non-Desi libraries: see Section 16.) The natural strategy would seem to be DLLs, but this may be seen as a rather out-of-date approach, as as being Windows-centric. If Desi goes with .NET, it should probably generate code for the CLR.

Assuming that DLLs constitute a viable approach, the implementation of pre-compiled libraries would look something like this:

1. Compile library functions to DLLs.
2. Write interfaces for libraries using Desi declaration syntax.
3. Include code in the interpreter that can open a DLL and link to the functions in it.

An **import** directive simply makes the source code in the **import**ed file accessible to the compiler. Section 16 discusses the implementation of modules that were not written in Desi.

16. Interoperability

> *Interoperability is the capability of a product or system—whose interfaces are fully disclosed—to interact and function with other products or systems, present or yet to come, without any access or implementation restrictions.*
>
> Wikipedia, 16.05.2010

The only aspect of interoperability that we deal with here is the need for Desi to invoke components written in other languages.

16.1. Goals

It is neither desirable nor feasible to require that all Desi software is written in Desi. A practical implementation of Desi will inevitably rely heavily on existing libraries for numerical computation, GUI tools, graphics, database access, and much more. The minimum requirement is that a Desi program should be able to make use of software written in another language. A secondary requirement, important for long-term acceptance, is that programs written in other languages should be able to make use of Desi software.

16.2. Alternatives

One fairly easy way to achieve interoperability is to target an existing platform. The JVM is an obvious choice, because it has been implemented on most platforms. .NET, on the other hand, is primarily a Windows system although implementations exist for other platforms.

Several Virtual machine builders are available. Despite the name, these systems are susually generic assemblers that define an abstract Instruction Set Architecture (ISA) (hence the name "VM") but generate code for specific architectures.

LLVM has attracted a lot of attention. It has been developed primarily for MacOS but runs—albeit with some difficulty—on Windows and other platforms. Its advantages include provision for concurrent programming (e.g., instructions for locking) and optimization.

Lightning is similar in spirit but much simpler. It does no optimization and does not seem to have any facilities for concurrent programming.

Windows provides dynamically-linked libraries (DLLs) for applications needing run time linkage. Since DLLs are subject to various problems ("DLL Hell"), Microsoft have put forward several improvements, including Windows File Protection, Side-by-side Component Sharing, and, finally, .NET.

16.3. Design

The following sequence of actions can be used to add a library of C functions to the Desi system.

1. Compile the C functions to a DLL.
2. Create a Desi interface for the C functions. The interface is a standard Desi module that contains only constants, types, and routines, and in which routine bodies are replaced by an identifier that specifies the source language (C in this example).
3. Include an **import** directive to the interface in the Desi program. We might need a pragma to indicate that the module has a corresponding DLL but no source code.

It would not be hard to develop a filter that reads C source and generates the Desi interface.

Similar approaches could be used for other languages, provided they can somehow be mapped to DLLs.

17. Implementation

17.1. Goals

Desi should be simple to use, with an "edit/run" style of development rather than "edit/compile/link/execute" style. Code is generated on the fly by JIT and libraries are dynamically linked as needed.

17.2. Alternatives

More "traditional" styles of implementation, involving separate compile, link, and load steps, with static libraries, were considered but rejected.

Although JIT tools exist, it is hard to know which one to choose. Lightning is simple and works well, but it performs no optimization, has no (apparent) facilities for concurrent programming, and is tightly coupled to C. LLVM is complex, performs serious optimization, and generates concurrent code, but is hard to understand and deploy (for Windows, at least).

17.3. Design

At the time of this revision (April 2011), we are building an implementation of Desi that will have two components:

- The Universal Desi Compiler (UDC) translates Desi source code to Desi Intermediate Language (DIL), writing its output to a text file.
- The Just In Time assembler (JIT) reads DIL and either interprets it or generates an executable.

► Interpreting and on-the-fly code generation represent two ends of a spectrum of possible implementations. The interpreter would define a "canonical" implementation against which other implementations would be tested. The code generator would be fast. Intermediate possibilities, such as generating and then compiling C code, were considered (May 2011) and rejected.

If this approach is successful, we should be able to streamline it by combining UDC and JIT into a single program, with DIL represented as an API.

18. Missing Features

The following features are *not yet implemented* but should be included very soon.

- Local constants and types within a sequence.
- External functions in languages other than C.
- Transfer of binary data (currently, all input and output is for *Text* only).
- The use of protocol names within protocols.
- Parameterized protocols.

The following sections discuss more significant weaknesses of Desi that do not appear to have an immediate resolution.

18.1. Data Structures

The Erasmus project has, narturally enough, tended to focus on flow of control, paying relatively less attention to data. At an early stage, the idea of cells as data receptacles emerged, and was revisited in KIWI 100131.

In this section, we suggest a classification of data into four main kinds, and discusses how Desi might handle each of them.

18.1.1. Simple Data

The category "simple data" includes all kinds of data that are usually considered as indivisible entities. As well as obvious candidates such is *Character*, *Word*, etc., we place *Text* into this category. A *text* can be split up into individual characters, but the ways of doing this are specific to *Text*s and do not generalize to other types.

We do not foresee any problems in Desi's handling of simple data: current mechanisms seem adequate.

18.1.2. Objects

There are many definitions of object. For the purposes of this discussion, an object has several components and associated methods for operating on those components. In many cases, the components are hidden, and access to them is provided *only* via the methods. There is an entire programming paradigm devoted to the management of this kind of data; it is called *object-oriented programming*, or OOP.

Desi accommodates OOP styles fairly easily. A simple object can be represented by a process. The process has one or more ports associated with protocols that correspond to the methods of a class.

More complex objects can be represented with cells. The interface of a cell is the same as that of a process: it consists of one or more ports, each with a protocol. However, a cell may contain several processes, and thereby provides concurrent access to shared data.

There is a potential danger when cells are used to represent objects. Several people have expressed concern about the difficulty of coding multithreaded objects (in OOPLs) correctly. It is not clear that coding cells in which concurrent processes access shared data would be any easier: in fact, the problems seem essentially equivalent. How can Desi do better than concurrent OOP?

18.1.3. Dynamic Data Structures

The characteristic of a dynamic data structure (DDS) that distinguishes it from the previous two categories is that its size may change at run time (hence the qualifier "dynamic"). The most general DDS, at least for the purposes of this discussion, is a graph consisting of nodes linked by edges (a.k.a. "arcs"); lists and trees are special cases.

In contemporary OOPLs, DDS are typically represented by objects at the nodes of the graph with references to objects acting as (directed) edges. This is a convenient abstraction from the old representation of a DDS as records linked by pointers.

For various reasons, Erasmus has eschewed pointers and (mostly) references. One problem is that, once an object is referenced, it is effectively locked in place. Another, more serious, problem is that references assume a single, uniformly-addressable memory space. This assumption does not necessarily hold for a distributed system in which processes may migrate between processors and memory spaces.

It is difficult to escape from addressable memory entirely, however. In mE, communicating processes both know the address of the channel that they share access to. This is clearly less efficient than a single pointer, but has the advantage that a channel may be split between two processors without the processes noticing the difference. See Figure 3 on page 43.

Desi could represent DDS with a process for each node and a channel for each edge. Performance will be lower than with a pointer-linked structure but, if our goal of efficient communication is achieved, should be acceptable.

18.1.4. Values

MacLennan (1982) introduced insights that were novel for the time and has influenced my (PG) thoughts about data ever since. The distinction that MacLennan makes is roughly this:

- A *value* is an abstraction such as a number or other kind of mathematical entity. Values do not have any sense of identity: the important thing is the value 2, say, not this 2 or that 2. Values do not change. The contents of a cell containing a value may change, but this does not mean that 2 becomes 3, it means only that the cell the used to contain 2 now contains 3. It is often unnecessary to store values, because we can recompute them when they are needed. Many kinds of value can be combined using *binary operations* that take two (sometimes more) values and produce a new value.
- An *object* typically corresponds to a real-world entity, perhaps a concrete object such as a person or invoice, but also possibly a concept such as a vehicle. Objects may have the same values but nevertheless correspond to different entities. Two objects may have the name attribute "John Smith" and the birthdate attribute "1 January 2000", but they do not necessarily represent the same entity. Thus objects have an *identity*, which further implies that we must carefully distinguish sharing and copying them.

Most PLs do not handle values well. In particular, OOPLs often introduce artifical asymmetries, as when $x + y$ is implemented as $x.plus(y)$. The problem with this expression is that x and y behave differently with respect to inheritance, since the class of *plus* is determined only by x.

C++ does a better job than most languages. Consider an expression such as `m*v`. in which `m` is a `Matrix` and `v` is a `Vector`. This expression relies on three features of C++:

1. Operators can be overloaded.
2. "Free" functions (as opposed to class functions, a.k.a. methods) allow symmetrical treatment of operands.
3. The `friend` attribute allows individual functions to have access to private fields of objects without exposing them to the caller.[7]

Despite valiant efforts in various KIWIs, most notable KIWI 100131, we do not yet have a fully satisfactory way of handling values in Desi.

18.2. Communication

In the current design of Desi, communication has a rather static nature, with most features frozen at compile-time. As noted in Section 14, we should try to make communication capabilities more dynamic than they are at present.

[7] The `friend` facility is often perceived as weakening encapsulation in C++. Used wisely, however, it actually *strengthens* encapsulation.

19. References

Brinch Hansen, P. (1987, January). Joyce—a programming language for distributed systems. *Software—Practice and Experience 17*(1), 29–50.

Grogono, P. and N. Jafroodi (2010, May). A fair protocol for non-deterministic message passing. In B. C. Desai, C. K. Leung, and S. Mudur (Eds.), *Proceedings of the Canadian Conference on Computer Science & Software Engineering (C^3S^2E'10)*, pp. 53–58. ACM Press.

MacLennan, B. J. (1982, December). Values and objects in programming languages. *ACM SIGPLAN Notices 17*(12), 70–79.

Strom, R. (1986, October). A comparison of the object-oriented and process paradigms. *ACM SIGPLAN Notices 21*(10), 88–97. SIGPLAN Object-Oriented Programming Workshop, June 9–13, 1986.

A. Grammar Summary

Since the grammar is generated automatically from `desi-grammar.grm`, the ordering of the rules does not correspond to the body of the report.

Terminal symbols

also and andb any assert case cell constant domain else enum exit external false for if implicit impliesb import in indexes loop mod nand nor not op or orb policy process protocol public range routine select such that true type xor xorb

Intrinsic NumericLiteral TextLiteral iden

‘#’ ‘%’ ‘%=’ ‘(’ ‘)’ ‘*’ ‘*=’ ‘+’ ‘+=’ ‘,’ ‘-’ ‘-=’
‘->’ ‘.’ ‘..’ ‘/’ ‘//’ ‘//=’ ‘/=’ ‘:’ ‘:=’ ‘;’ ‘<’ ‘<<’
‘<<=’ ‘<=’ ‘<==’ ‘<=>’ ‘<>’ ‘=’ ‘==>’ ‘>’ ‘>=’ ‘>>’
‘>>=’ ‘>>>’ ‘>>>=’ ‘?’ ‘@’ ‘[’ ‘]’ ‘^’ ‘{’ ‘|’ ‘}’ ‘~’

Non-terminal symbols

ArithmeticOp ArrayLiteral Assert AssignOp Assignment BinaryOp BitOp BoolOp BooleanLiteral Case CellDefinition CompOp Comprehension ConstantDefinition Declaration Definition EnumDefinition Exit ForAny Guard GuardedSequence Import Invocation Literal Loop Lvalue MapLiteral Module Parameters ProcessDefinition ProtocolDefinition RoutineDefinition Rvalue Select Sequence ShiftOp Signal Statement TextOp Type TypeDefinition UnaryOp UnguardedSequence

Module = { [**public**] (*Import* | *Definition* | *Invocation*) } .

Import = **import** { TextLiteral }‹‘,’› [‘;’] .

Definition = *ConstantDefinition*
| *TypeDefinition*
| *EnumDefinition*
| *ProtocolDefinition*
| *RoutineDefinition*
| *ProcessDefinition*
| *CellDefinition* .

ConstantDefinition = iden ‘=’ **constant** *Rvalue* ‘;’ .

TypeDefinition = iden ‘=’ **type** [*Type*] ‘;’ .

EnumDefinition = iden ‘=’ **enum** ‘{’ { iden }‹‘,’› ‘}’ .

ProtocolDefinition = iden ‘=’ **protocol** ‘{’ { [‘^’] iden [‘:’ iden] }‹‘;’› ‘}’ .

ProcessDefinition = iden ‘=’ **process** *Parameters* (*Sequence* | **external** TextLiteral ‘;’) .

CellDefinition = iden '=' **cell** *Parameters* (*Sequence* | **external** TextLiteral ';') .

RoutineDefinition = (iden | **op** (*UnaryOp* | *BinaryOp*)) '=' [**implicit**] **routine** *Parameters* (*Sequence* | '=' Intrinsic ';' | **external** TextLiteral ';') .

Type = iden
| *Type* **indexes** *Type*
| **mod** NumericLiteral .

Parameters = { *Declaration* }‹,› ['->' { *Declaration* }‹,›] .

UnguardedSequence = '{' { *Statement* } '}' .

GuardedSequence = '{' { *Guard* { *Statement* } } '}' .

Sequence = *UnguardedSequence*
| *GuardedSequence* .

Guard = '|' [*Rvalue*] '|' .

Statement = *Sequence*
| *Assert* [';']
| *Assignment* ';'
| *Case*
| *Declaration* ';'
| *Exit*
| *ForAny*
| *Invocation*
| *Loop*
| *Select*
| *Signal* .

Assert = **assert** '(' *Rvalue* [',' *Rvalue*] ')' .

Assignment = *Lvalue AssignOp Rvalue* .

Case = **case** [*Rvalue*] *GuardedSequence* .

Declaration = { iden }‹,› [':' ['+' | '-' | '@'] *Type*] [':=' *Rvalue*] .

Exit = **exit** .

ForAny = (**for** | **any**) { *Comprehension* }$_{\textbf{also}}$ [**such that** *Rvalue*] *Statement* [**else** *Statement*] .

Invocation = iden '(' { *Rvalue* }‹,› ['->' { *Lvalue* }‹,›] ')' .

Loop = **loop** { *Declaration* }‹,› *Statement* .

Select = [**loop**] **select** [**policy** iden ';'] { *Declaration* }‹,› *GuardedSequence* .

Signal = iden { '[' *Rvalue* ']' } '.' *Rvalue* .

Comprehension = '(' *Assignment* ';' *Rvalue* ';' *Assignment* ')'
| iden **in** (**domain** | **range**) *Rvalue*
| iden **in** *Type* .

Lvalue = iden { '[' *Rvalue* ']' } [('.' | '?') *Rvalue*] .

Rvalue = *Lvalue*
| *Literal*
| *UnaryOp Rvalue*
| *Rvalue BinaryOp Rvalue*
| *Rvalue* **if** *Rvalue* **else** *Rvalue*
| *Rvalue* '[' *Rvalue* '..' *Rvalue* ']'
| *Invocation*
| '(' *Rvalue* ')' .

Literal = NumericLiteral
| TextLiteral
| *BooleanLiteral*
| *ArrayLiteral*
| *MapLiteral* .

BooleanLiteral = **true**
| **false** .

ArrayLiteral = '{' { *Rvalue* }',' '}' .

MapLiteral = '{' { *Rvalue* '->' *Rvalue* }',' '}' .

AssignOp = ':='
| '+='
| '-='
| '*='
| '/='
| '%='
| '//='
| '<<='
| '>>='
| '>>>=' .

UnaryOp = '+'
| '-'
| '~'
| '#'

| | **not** .
BinaryOp = *ArithmeticOp*
| *CompOp*
| *ShiftOp*
| *BoolOp*
| *BitOp*
| *TextOp* .
ArithmeticOp = '+'
| '-'
| '*'
| '/'
| '%' .
CompOp = '='
| '<>'
| '<'
| '<='
| '>'
| '>=' .
ShiftOp = '<<'
| '>>'
| '>>>' .
BoolOp = **and**
| **nand**
| **or**
| **nor**
| **xor**
| '==>'
| '<=='
| '<=>' .
BitOp = **andb**
| **orb**
| **impliesb**
| **xorb** .
TextOp = '//' .