Ellie Language Definition Report

Birger Andersen

Department of Computer Science

University of Copenhagen

Universitetsparken 1

2100 Copenhagen, Denmark

E-mail: birger@diku.dk, Fax: +45 31 39 02 21, Phone: +45 31 39 64 66

Partial Ph.D. Dissertation [1]

October 1991

1. This is 2nd edition of a paper published in *ACM SIGPLAN Notices*, 25(11):45-64, November 1990.

Abstract

This report defines the parallel object-oriented language Ellie in details by means of informal definitions and examples. The syntax of Ellie is defined by a grammar. Ellie has been designed as a part of my Ph.D. thesis.

The goals of Ellie are to achieve machine independent parallel programming and great language flexibility. Machine independent parallel programming is achieved by allowing the programmer to have a huge number of small processes for a virtual distributed memory parallel computer. If an actual distributed parallel computer can not execute the huge number of processes efficiently then a compiler can combine small processes hereby reducing the number of processes.

Great language flexibility is achieved by allowing definitions of new types and control structures and by using a non-strict type system. Ellie is meant as an all-purpose programming language that has the flexibility of languages like Smalltalk and Lisp without losing efficiency in numerical calculations, etc.

Keywords

Language definition, Object-oriented languages, Concurrency, Fine-grained programming

Acknowledgements

The many helpful comments on this report and the language design from my supervisors Jens Clausen and Eric Jul are acknowledged with great pleasure. I also thank Bjarne Hansen for the discussions we have had on details of the Ellie language and Bjarne Steensgaard for comments on this 2nd edition of the report.

Table of Contents

Chapter 1
Introduction 7
1.1 Ellie Objects 8
1.2 Object Definition 9
1.3 Object Activation 10
1.4 Fine Granularity 11

Chapter 2
The Type System 13
2.1 Conformity Requirements 13
2.2 Process Conformity 15

Chapter 3
Name Declarations 17
3.1 Binding Kinds 17
3.2 Scope Rules 18

Chapter 4
Defining Objects 19
4.1 The Object Pattern 19
4.2 Input and Output Parameter 20
4.3 The Object Interface 21
4.4 Defining Abstract Objects 23

Chapter 5
Activating Objects 25
5.1 The Ellie Field Expression 25
5.2 The Ellie Operator Expression 26
5.3 Trusted Activations 27
5.4 Defining Operators 28
5.5 Defining Control Structures 29

Chapter 6
Parallelism 31
6.1 Forking Processes 31
6.2 Synchronization 31

Chapter 7
Writing Programs 33
7.1 Modules 33
7.2 Polymorphism 34
7.3 Dining Philosophers 34

Chapter 8
Notation and Vocabulary 37
8.1 Legal Names 37
8.2 Reserved Names 37
8.3 Built-in Names 38
8.4 Literal Objects 38
8.5 Separators 39
8.6 The Syntax of Ellie 39

Chapter 9
Built-in Objects 43
9.1 Number 43
9.2 Integer 43
9.3 Real 43
9.4 Boolean 43
9.5 Char 43
9.6 Vector 43
9.7 String 44
9.8 Option 44
9.9 Loop 45
9.10 Read 45
9.11 Write 45
9.12 Implicit Objects 50

References 55

Index 57

List of Figures

Figure 1.1: Object Relations 10
Figure 1.2: Fibonacci Function 12
Figure 4.1: Record Object 22
Figure 4.2: Record Interface Inclusion 23
Figure 5.1: Complex Object 28
Figure 5.2: For-Loop Control Object 29
Figure 6.1: Semaphore Object 32
Figure 7.1: Block Zero Skeleton Object 33
Figure 7.2: Polymorphic Square Function 34
Figure 7.3: Dining Philosophers Program 35
Figure 9.1: Number Object 46
Figure 9.2: Integer Object 46
Figure 9.3: Real Object 47
Figure 9.4: Boolean Object 48
Figure 9.5: Character Object 48
Figure 9.6: Vector Object 49
Figure 9.7: Selective Control Object 49
Figure 9.8: Loop Control Object 49
Figure 9.9: Label Name Object 51
Figure 9.10: Constant Name Object 51
Figure 9.11: Variable Name Object 52
Figure 9.12: Simple Objects 53

List of Tables

Table 2.1: Conformity Relations 15
Table 5.1: Operator Precedence Levels 27
Table 5.2: Trust Dependencies 28

Chapter 1
Introduction

This language report defines the object-oriented programming language Ellie. Ellie has been designed to support high level fine-grained programming in a flexible and efficient way on distributed memory parallel computers. The language has been designed as a part of my Ph.D. thesis [Andersen 91a, 91b].

The primary goal of Ellie is to combine a small number of highly flexible object-oriented programming language concepts with the capability to express *fine-grained objects* and *fine-grained parallelism* explicitly in a way that the granularity, i.e., the size of objects and processes, used in a program may be adjusted by a compiler to correspond to the underlying architecture.

A distributed computer with hundreds or thousands of processors requires programs with many small fine-grained objects to exploit the distributed memory and processing power, whereas a computer with few processors can handle a smaller number of less fine-grained objects more efficiently. The reason is that a high number of fine-grained independent objects located on a single processor produces a higher administration overhead than a smaller number of less fine-grained objects.

By *static grain-size adaption* an Ellie program can to some extent be compiled to fit the number of processors and their memory size, i.e., the grain-size of objects and processes may be increased to *medium-grained* or *coarse-grained* (bigger) and the number of objects and processes may be decreased. Grain-size adaption is intended as a step towards machine independent parallel programming.

Although Ellie is a new language, it is based on a number of concepts known from other object-oriented programming languages. For example, the nature of objects is similar to that in Smalltalk [Goldberg 89], i.e., all data items are objects on which methods may be applied in a uniform way. As opposed to Smalltalk, Ellie does not have any notion of *classes* but relies on the concept of *prototypes* as known from several other languages, e.g., Actors [Agha 87]. A new object in Ellie is created based on an existing object which thus serves as a prototype.

Ellie relies on block structures like Beta [Kristensen 87; Madsen 87] and Emerald [Black 87; Hutchinson 87a, 87b; Raj 91]. To execute efficiently, static type checking is supported. Ellie relies on the powerful concept of *conformity typing* known from Emerald, but Ellie has no type objects. Instead, any object may serve as a type. In Smalltalk, a class may include the properties of a super class. This concept is called *inheritance*. In Actors, requests that can not be served by an object are sent along to some other object. This is called *delegation*. Ellie combines both concepts in one mechanism, *interface inclusion*.

Parallelism is explicitly established in Ellie at all levels, including the fine-grained level, by the concept of process forking (activating child processes) known from Concurrent Pascal [Brinch Hansen 75, 77], Joyce [Brinch Hansen 87], ConcurrentSmalltalk [Yokote 87], and many other programming languages. In Ellie, master processes do not need to wait for their child processes to finish because the results from child processes are returned to the masters by means of *future result objects*. Future result objects are, for example, known from ABCL/1 [Shibayama 87] and Cantor [2] [Athas 87]. Ellie has a powerful synchronization mechanism called *dynamic interface* that allows objects to choose between incoming requests in any sequence and deal with them sequentially for mutual exclusion or in parallel (or a combination). The synchronization mechanism is similar in expressiveness to Path Expressions [Campbell 74].

To deal with side-effect problems, Ellie has *functions* that have no side-effects and *operations* that are allowed to have side-effects. Functions and operations are objects that may have processes which are forked by masters (other functions and operations with processes). By static grain-size adaption of parallelism, the number of forked processes of functions is decreased.

Grain-size adaption does not change the semantics of correct programs but programs with indeterminable elements that seem to be correct at one grain-size may behave incorrectly at another grain-size. This kind of indeterminacy is illegal. Therefore, the programmer is encouraged to compile and run programs at more than one adapted level to recover such programming errors. In order to write the most machine independent programs, the programmer is encouraged to fork processes of functions where possible.

1.1 Ellie Objects

The *object* is the single abstraction mechanism for notions of names, classes, types, blocks, methods (functions and operations), and regular objects (data). All entities in Ellie are objects; from characters and numbers to programs and databases, i.e., they are all *first class objects*. Any object may serve as a prototype for new objects.

An object can either be *uninitialized*, *initializing*, or *initialized*. An uninitialized object may only serve as a prototype for creation of new objects. This is called an *activation* in Ellie. Initializing and initialized objects may also serve as prototypes. An initializing object is *active*, i.e., it is initializing or changing its state; it is said to have a *process*.

An initialized object may have had a process and is *passive*. An object consists of the following:

- A *reference*, which uniquely identifies the object.
- A *representation*, which consists of references to other objects.

---

2. Cantor is another fine-grained object-oriented language.

- An optional *interface*, which allows other objects in possession of the reference to access the part of the representation defined by the interface.
- An optional process, which is initiated when the object is activated.

The interface of an active object is *closed*, i.e., requests to access the representation are suspended. The interface will be *opened* when the object becomes inactive and any suspended request will be resumed. An activated object goes through the following phases:

1) It has just been activated. Optionally, it has a process that is changing the state of the object, i.e., the representation, and may change states of other objects known by references (the object is active and initializing itself).
2) The process has terminated (or never existed) and the interface, if any, is opened (the object is passive and initialized). This means that the state now is only accessed and changed by the process of other objects.
3) The last reference to the object has ceased to exist and the object also ceases to exist.

If the representation may change over time then the object is said to be *mutable*. If the interface does not allow other objects to change the state (only to read it), then the object becomes *immutable* in the second phase. Immutability simplifies sharing of objects in a distributed environment because immutable objects can be shared by making copies.

Parallelism exists both between objects and within an object. If an object with a *future process* is activated then parallelism is generated because the process will execute in parallel with the process activating the object (equivalent to *unbounded remote procedure call, URPC*). If an object with a *non-future process* is activated then no parallelism is generated because the process activating the object is *suspended*, i.e., it will not execute again before the process of the activated object has terminated (equivalent to *remote procedure call, RPC*). Because objects through an interface may activate objects within another object and these objects can have future processes, the number of processes within an object has no limit.

Processes synchronize by opening and closing object interfaces. By using this powerful mechanism, other synchronization primitives may be defined, e.g., condition and semaphore objects.

1.2 Object Definition

An Ellie object is defined by means of an *object constructor*. Data accessible through the interface, private encapsulated data, input and output parameter, and the process for reading and modifying the state are defined by the object constructor.

The value of an object constructor is a reference to an uninitialized object with the properties defined by the constructor. For example,

```
[ function <: i :- integer :> i * i ]
```

defines an object that when passed an integer number as actual parameter in an activation returns the square of the number.

Four kinds of objects can be defined: *Objects*, *future objects*, *express objects*, and *future express objects*. The process of a normal object can not execute in parallel with the process of the object activating it, whereas the process of a future object can. The process of an express object is given higher execution priority by the processor than the process of a normal object.

Furthermore, an object can be defined to be a *function* or an *operation*. An object defined to be a function may have a *functional* process, whereas an object defined to be an operation may have an *operational* process. If a process is operational, it is allowed to have side-effects when active. If a process is functional, it must not have any side-effect. By side-effects we mean modification of states outside the object itself. The creation of objects (allocation of memory) is not considered a side-effect.

1.3 Object Activation

The only method for communication in Ellie is the object activation. New objects are activated by using other objects as prototypes. As an example, let *X*, *R*, and *S* be existing objects where *S* is accessible by *X* through the interface of *R* (*S* has been defined within *R*). The object *X* may then activate a new object *S'* using *S* as a prototype. The process of *S'* may access (and possibly modify) the internal state of *R*. The object *S'* (and *S*) is called a *local object* of *R*. The object *R* is called the *origin object* of *S'* (and *S*). An object can only have one origin object but any number of local objects. The relations in this example are shown in Figure 1.1.

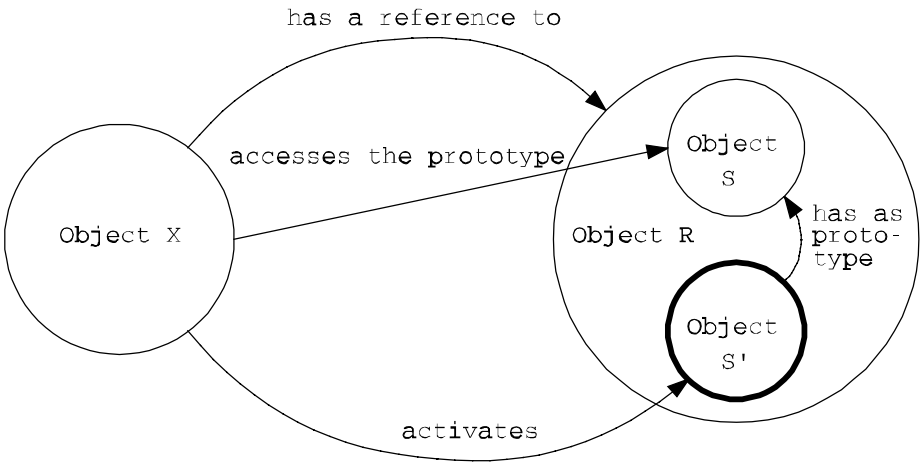

Figure 1.1: Object Relations

An object may activate an object within its origin object, origin-origin object, origin-origin-origin object, etc., or within one of its local objects by evaluating an *Ellie expression*, for example:

```
target.prototype parameter
```

The origin object (or local object) is referred to by the name `target` and an actual parameter by the name `parameter`. The name `prototype` refers to the object within `target` that is used as prototype for the activation.

An object may also activate an object within a local object, referred to by the name `local`, of its origin object (or local object), referred to by the name `target`, by evaluating an expression of the following form:

```
target.local.prototype parameter
```

This is possible to any depth of local objects as long as the local objects can be accessed through the interface of their origin objects.

If the process of a new object is non-future then the activation is synchronous, i.e., the process of the object evaluating the Ellie expression is suspended. If it is a future process, the activation is asynchronous (a process is forked), i.e., the activating process is not suspended.

The object receives an input parameter when activated. When its non-future process has terminated it returns an output parameter. In case of a future process it returns an output parameter immediately (before activation). Input and output parameters are passed by object reference. The output parameter of a future process is returned as a future result object with an undefined state. Requests to access a future result object will be delayed until the future process has terminated and defined the state of the output parameter. Requests to activate a new future object with a future result object (or an object referenced in its interface) as a

prototype will also be delayed but the new future result object will be returned immediately.

1.4 Fine Granularity

Ellie supports fine granularity, i.e., it supports definition and activation of very small/fine size objects having future and non-future processes to exploit distributed memory parallel computers with thousands of processor elements.

```
Fibonacci : [ future function <: N : integer
   :> fib :- integer;

   option [
      if : N > 1;
      then : [
         fib <- Fibonacci (N - 1) + Fibonacci (N - 2)
      ];
      else : [
         fib <- N
      ]
   ]
];
```

Figure 1.2: Fibonacci Function

As a fine-grained example, consider the Ellie function in Figure 1.2 that when activated will compute the *n*th Fibonacci number in parallel by recursion (the definition of objects is considered in section 1.2 and 4, and the activation of objects in section 1.3 and 5).

When activated, e.g., by evaluating the expression `Fibonacci 5`, a binary tree of objects with future processes (activations of the object bound to `Fibonacci`) will be spawn. In case of the expression `Fibonacci 5`, we will get a tree of 15 objects. The objects will be future functions which take an integer *n* as input parameter and return the *n*th Fibonacci number as output parameter (future result object). The name `option` refers to a control structure object that activates its input parameter and hereby binds the constant `if` to the value of `N > 1`. Depending on the value of `if`, the uninitialized object bound to either the constant `then` or the constant `else` will be activated.

To make the fine granularity obvious, it can be noticed that the operators >, −, and + are bound to very fine-grained objects with very fine-grained future processes that are also activated.

Chapter 2
The Type System

Central to Ellie are the concepts of *conformity* and *data abstraction*. These concepts are also present in the Emerald language [Black 87; Hutchinson 87a, 87b; Raj 91], but Ellie diverges from Emerald in the notion of types. In Emerald abstract types are defined explicitly by special definitions, whereas in Ellie they are defined implicitly in the sense that any Ellie object may be utilized as an abstract type. The abstract type of any object may be used for declaration of names; the object is said to be used as a *qualifier*. Two or more objects with different implementations, i.e., concrete types, may define the same abstract type.

In this report we sometimes talk about types as though they exist as special objects in Ellie, e.g., by the notion 'being of (concrete) type `name_T`' we really mean 'having the object `name_T` as prototype'. By the notion 'the object `name_T`' we really mean 'the object referred to by the reference that the name `name_T` is bound to'. If an object `name_T3` is of type `name_T2` and the object `name_T2` is of type `name_T` then all three objects have the same concrete type. By the notion 'being of abstract type `name_T`' we really mean 'defining the same abstract type as the object `name_T`'.

Data abstraction is accomplished by providing an object with an interface through which the object may make names and objects bound to these names accessible to any object having a reference. The implementation of local objects are *encapsulated* by the origin object. The state of an object can only be accessed by activation of encapsulated objects. The implementation of an object is always encapsulated by the object itself. This chapter will not elaborate on data abstraction in Ellie, only on the conformity type system.

2.1 Conformity Requirements

The concept of conformity requires that the abstract type of the object being passed as actual parameter must *conform* to the abstract type of the qualifier of the formal input parameter [3]. Informally, an object *A* conforms to an object *B*, i.e., *A* may be used wherever *B* is used, if the following requirements are satisfied:

1) *A*'s process conforms to *B*'s process (see section 2.2).

2) *B*'s input parameter conforms to *A*'s input parameter.

3) *A*'s output parameter conforms to *B*'s output parameter.

---

3. The formal input parameter is defined in the object definition and is analogous to a formal parameter in, e.g., Pascal.

4) *A* exports at least the exported or accepted [4] names of *B* or *A* accepts at least the accepted names of *B* (*A* may export/accept more names).

5) For each name exported or accepted by both *A* and *B*, the abstract type of *A*'s name conforms to the abstract type of *B*'s name.

The abstract type of a name is defined by the *name kind* (label, constant, or variable) and the abstract type of the name's qualifier. A name is a simple object that exports a few names of its own and delegates requests to the names exported by the qualifier to the object bound to the name at the time of the delegation (see section 9.12 for additional information on name objects).

The fact that `object_A *> object_B` (read `*>` as 'conforms to') does not imply `object_B *> object_A`. If both Ellie expressions are true then `object_A` and `object_B` refer to objects with the same abstract type, i.e., not necessarily the same concrete type or prototype. Conformity is the basis of type checking in Ellie.

Type checking may be done statically in most situations, however, in a few situations dynamic type checking is needed. For example, one may need a variable name to refer to an object of type `integer` at one time and refer to an object of type `real` at another time. This can be accomplished by declaring the variable to be qualified by an object `number` being of a common abstract type (see section 9.1):

```
num :- number;
i :- integer;
r :- real;
```

The three variables `num`, `i`, and `r` are qualified by the object specified after the `:-` operator. Both `integer` and `real` conform to `number`. Now we may assign objects of both types to `num` by using the assignment operator `<-` exported by variable names:

```
i <- 2; r <- 2.5;
num <- i; num <- r;
```

It may now be necessary to *change view* on a reference in order to *widen* the abstract type, i.e., make invisible parts of the interface visible for activations. For example, the assignment

```
r <- num;
```

is illegal because the qualifier of `num` does not conform to the qualifier of `r`. In this situation the solution is to change view by using the widening assignment operator `<-*` also exported by variable names:

```
r <-* num;
```

4. An object with an interface varying over time is said to *accept* accesses to the names in the interface (controlled access), whereas an object with a normal interface unrestricted *exports* the names in the interface (uncontrolled access). This is discussed in detail in section 6.2.

The change of the view must be dynamically checked by making an conformity check between the qualifier of `r` and the actual object referred to by `num`.

2.2 Process Conformity

The conformity relations between different kinds of processes are defined in Table 2.1. Note, that these conformity relations are included in the conformity requirements described in section 2.1. In the table, the value of the expression `horizontal *> vertical` is given for all combinations of process kinds. The letter *T* indicates that the expression evaluates to true, whereas the letter *F* indicates that the expression evaluates to false. For example, a function conforms to a future operation, whereas a future operation does not conform to a function. The `express` kind definition of a process does not affect the conformity relations, whereas the `future`, `function`, and `operation` kind definitions do.

`horizontal *> vertical`	1	2	3	4
1 - `future function`	*T*	*F*	*T*	*F*
2 - `function`	*T*	*T*	*T*	*T*
3 - `future operation`	*F*	*F*	*T*	*F*
4 - `operation`	*F*	*F*	*T*	*T*

Table 2.1: Conformity Relations

A non-future object will be activated as a future object when the variable name, that is used in the activation and refers to the non-future object, has been qualified by a future object. An example is shown below:

```
v :- [ future operation; write "hello world" ];
v [];  $ A future operation activated as it is
v <- [ operation; write "synchronous printing" ];
v [];  $ A non-future operation activated as future
```

The other way around is impossible, i.e., activating a future object as a non-future object, because future objects never conform to non-future objects. This is a design decision made for efficiency reasons.

Chapter 3
Name Declarations

A named attribute in Ellie is represented by an *implicit simple object* (see section 9.12). Every name, i.e., label name, constant name, or variable name, must be declared in order to implicitly create objects to represent them. A range of built-in label and constant names are declared by default; they may all be redeclared by the programmer, except when locally defined.

3.1 Binding Kinds

A name in an Ellie declaration is bound in the following three steps:

1) It is bound either to a *label name* (the binding selector `::`), to a *constant name* (the binding selector `:`), or to a *variable name* (the binding selector `:-`), i.e., the name kind is defined and one of the implicit simple objects to represent the name is created.

2) It is bound to a qualifier (see below).

3) It is bound to a value (see below) conforming to the qualifier (may be the qualifier itself). Variable names may later be rebound any number of times to any object conforming to the qualifier by means of expressions evaluated by processes.

A label name can only be used to activate new objects using the object bound constantly to it as a prototype. A constant name can also be read, and a variable name can also be read and rebound. Furthermore, objects bound to constant names and variable names can be conformity checked explicitly. The syntax for name declarations is (in EBNF notation, see section 8):

```
<binding_list>      ::=  <name_list> <binding_selector>
                         <expression>
<name_list>         ::=  <name> {"," <name>}
<binding_selector>  ::=  "::" | ":" | ":-"
```

The expression after the binding selector evaluates to a value that is bound to the name in 3) above, whereas the qualifier in 2) is the output parameter's qualifier of the expression, i.e., of the object activated in the expression. Qualifiers can always be determined statically. For example, consider an object that contains the following declarations:

```
first_int, second_int :- 10;
Fib_10 : Fibonacci 10;
```

The variable names `first_int` and `second_int` are both bound to the qualifier `integer` and the value 10 of type `integer`, i.e., the value is also determined

statically in this case. The qualifier of the output parameter of the expression `Fibonacci 10` will define the qualifier for the constant name `fib_10` to be `integer`. The value of the name `fib_10` is derived from the evaluation of the Fibonacci expression which is likely to be of concrete type `integer` but may be of some other type conforming to `integer` when actually evaluated.

Uninitialized objects have their locally defined inaccessible names bound to their qualifiers and their values are undefined. Initialized objects will have their locally defined names bound to values found by evaluation of the expressions in their declarations at the moment of activation. This is done by the process evaluating the expressions one by one from top to bottom.

The representation of a name object consists of two references, one to the qualifier, the other to the value. Furthermore, the representation may consist of references to built-in local objects referred to by the built-in names `<-`, `->`, `<-*`, `*->`, and a few others (see section 9.12).

3.2 Scope Rules

The scoping of names in Ellie follows the same rules as in traditional block structured languages. A name is known throughout the scope in which it is declared, not just textually after the declaration. The object constructor (see section 1.2 and 4) is the only construct that opens new scopes for names. Names are imported implicitly into new nested scopes, unless they are redeclared. Names are accessible out of scopes by definition of object interfaces. At object activation, the expressions of the declarations are evaluated one by one in the order of declaration to find the values.

Because a name is known throughout the scope in which it is declared, it is legal to have expressions in declarations that depend on other names declared in the same scope or the name itself, i.e., recursive declarations. A recursive declaration is illegal only when the qualifier can not be bound. For example, consider the following declaration:

```
t :- [ <: i : integer :> j : t ] 2;
```

The qualifier is `t` (the output parameter's qualifier), so the name `t` will be qualified by `t`! Obviously, `t` is unbound and the declaration is illegal. The following declaration is legal:

```
t :- [ <: i : integer :> j : t ];
```

In this case, the qualifier is the uninitialized object defined by the brackets, so the name `t` gets this qualifier and the output parameter also gets this qualifier.

Chapter 4
Defining Objects

Ellie objects are defined by means of the *object constructor*. The syntax is shown below:

<object_constructor> ::= "[" [<object_pattern>] "]"

The object pattern defines the properties and structure of the object and is optional. If missing then the constructor defines an *abstract object* (see section 4.4 and 9) or the simplest possible object.

The value of an object constructor is a reference to an uninitialized object with the properties and structure specified by the object pattern. This implies that the object constructor may be used in Ellie expressions. For example,

```
[ operation <: s :- string; write s ] "hello world";
```

defines and activates an object that writes its input parameter qualified by `string` out on the screen using the built-in operation `write`. The result of evaluating the Ellie expression is analogous to the evaluation of the expression `write "hello world"`. Consider another expression where the parameter is an object constructor:

```
new_write <- [ operation <: s :- string; write s ];
```

This expression binds the variable `new_write`, defined somewhere else, so it now may be used to activate objects with the same properties as the built-in object bound to the built-in constant `write`.

4.1 The Object Pattern

The object pattern defines the properties and the structure of an object. The syntax of the object pattern is shown below:

<object_pattern> ::= <object_header> [";" <object_tailer>] [";"]
| <object_tailer> [";"]
<object_header> ::= ["future"] ["express"]
<object_kind> [<parameter_part>]
| <parameter_part>
<object_kind> ::= "function" | "operation"
<object_tailer> ::= <interface> [";" <object_body>]
| <object_body>
<object_body> ::= <binding_section> [";" <expression_list>]
| <expression_list>
<binding_section> ::= <binding_list> {";" <binding_list>}
<expression_list> ::= <expression> {";" <expression>}

Non-future objects are default when nothing is defined, i.e., the keyword `future` is missing. Non-express is default when the keyword `express` is missing. If an

object is not defined to be a function or an operation, i.e., the keywords `function` and `operation` are missing, then a function is assumed when no expression list is present. If an expression list is present then an operation is assumed.

If the object pattern does not contain an object tailer then the object constructor defines an abstract object (see section 4.4 and 9). If the pattern defines no expression list then we have the typical notation for a record in Ellie, the fields of the record being the names declared and initialized in the binding section and exported by the interface. If there is no binding section, then only the names of input and/or output parameters can be made accessible by an interface.

For example, consider the definition of a record below:

```
coordinate : [
   x, y :- integer;
   angle :- real
];
```

When no expression list is present then the interface is implicitly defined by `export`. Definition of the interface by `export` is by default the same as `export x, y, angle` (see section 4.3).

Functions may be defined locally within operations and vice versa. A process may activate *trusted* processes only. A process can only supply a trusted actual parameter in an activation. The trust requirements for activations are described in details in section 5.3. The main idea of these requirements is preventing illegal activations of operations by functions in order to avoid illegal side-effects.

4.2 Input and Output Parameter

The syntax for definition of the input and output parameter in an operation header or in a function header is shown below:

<parameter_part>	::=	<input_parameter> [<output_parameter>]
	|	<output_parameter>
<input_parameter>	::=	"<:" <parameter>
<output_parameter>	::=	":>" <parameter>
<parameter>	::=	[<name> <binding_selector>] <expression>
<binding_selector>	::=	"::" | ":" | ":-"

When an object is activated, the actual parameter is evaluated and passed by object reference as the input parameter to the new object. The output parameter is evaluated and passed back by object reference.

The input parameter definition may be a name declaration. In this case, the actual parameter must conform to the qualifier of the label, constant, or variable name defined by the expression. The expression will not be evaluated because the input parameter name is bound to the passed actual parameter instead. If no input parameter name declaration is present (only an expression) then the actual parameter must conform to the qualifier expression present, but the actual parameter will be inaccessible. If no input parameter definition is present, the

qualifier `[ future operation ]`, which any actual parameter will conform to, is default.

The output parameter may also be defined by a name declaration. In this case, the expression will be evaluated and the resulting value bound to the label name, constant name, or variable name. If the name is a variable name, it may be rebound at any time, i.e., it behaves like a normal variable. If no output parameter name declaration is present (only an expression) then the value of the expression will be the output parameter. If no output parameter definition is present, the object bound to the implicitly defined constant `self` (a reference to the activated object itself) will be the qualifier and the value by default.

As an example, consider again a definition of a record:

```
coordinate : [
   export;          $ May be omitted
   x, y :- integer;
   angle :- real
];
```

All locally programmer defined names are exported by `export` and the object header is `function <: [ future operation ] :> self` by default. This means that a reference to the new record is returned when `coordinate` is activated. Any actual parameter may be supplied.

4.3 The Object Interface

Objects can let other objects access their locally defined names by the definition of an *export interface*. The syntax for definition of an interface is shown below:

```
<interface>          ::=   <interface_kind> [<access_list>]
<interface_kind>     ::=   "export" | "accept"
<access_list>        ::=   <access_item> {"," <access_item>}
<access_item>        ::=   <access_selector> [<remote_selector>]
<access_selector>    ::=   "(" <name> ")"
                     |     <name>
<remote_selector>    ::=   "." "(" [<name_list>] ")"
```

If an interface definition defines no access items then all locally defined names within the object are included in the interface, the exception being the implicitly defined names, such as `binary_value` and `self` (se section 9.12). Implicit names are not allowed to be in an interface.

The various kinds of ways to define interfaces will be explained by some examples. A *remote name* of a name `N` is defined to be a name that is accessible through the interface of the simple object representing `N` or objects bound to `N`, i.e., objects conforming to the qualifier of `N`. Consider the following interface definitions:

`export` — Exports all locally defined names for activation, except the implicit names. Also the remote names of these names (defined by the simple name objects or by the objects bound to the names), if any, are accessible.

`export N`	Exports the name `N` (must not be implicit) for activation only. Also the remote names of `N` (defined by the name object or by the object bound to `N`), if any, are accessible.
`export N.()`	Exports the name `N` (must not be implicit) for activation only. The remote names of `N` are inaccessible.
`export N.(M)`	Exports the name `N` (must not be implicit) for activation only. Also the remote name `M` of `N` is accessible.
`export (N)`	Does not export the name `N`, but exports the remote names of `N` (defined by `N` or by the object bound to `N`), if any, for activation (the Ellie interface inclusion mechanism).
`export (N).()`	No effect.
`export (N).(M)`	Does not export the name `N`, but exports the remote name `M` of `N` for activation (partial interface inclusion).

A number of access items may be present after `export` and a number of names may be present in the remote selector. Undefined/inaccessible names are illegal in the definition of an interface. The access list may contain name conflicts (the same name in the interface more than once). In case of a name conflict, the name textually first included in the interface will be accepted and the other(s) ignored. This handles name conflicts when implementing *multiple inheritance* by the interface inclusion mechanism in Ellie.

```
record : [
   export (a), b.(), c.(x, y), d, (e).(x2, y2), (f);
   a, b, c :- [
      x, y, z :- real
   ];
   d, e :- [
      x2, y2, z2 :- real
   ];
   f :: real
] [];
```

Figure 4.1: Record Object

As an example, consider the definition of a more complicated record in Figure 4.1. This record definition allows, for example, the activations shown in Figure 4.2 to take place (the addressing paths and their explanations are shown as comments) [5]. Some of the addressing paths are illegal in an activation, because the names `a`, `e`, and `f` are inaccessible from outside the definition of the object

5. Notice, that the operator `<-` is exported by variable names and the operator `*` by objects of type `integer` (see section 9.2 and 9.12).

```
record [];
record.<- real;     $ record.a.<- real, using export (a)
record.x [];        $ record.a.x [], using export (a)
record.y.<- real;   $ record.a.y.<- real, using export(a)
record.b [];        $ record.b [], using export b.()
record.c.x [];      $ record.c.x [], using export c.(x)
record.d.x2 [];     $ record.d.x2 [], using export d
record.y2.* 2;      $ record.e.y2.* 2, using export (e).(y2)
record.* 2;         $ record.f.* 2, using export (f)
```

Figure 4.2: Record Interface Inclusion

bound to `record`. Inside the object constructor, the interface has no influence on the accessibility of the locally defined names.

4.4 Defining Abstract Objects

When defining an abstract object, only the properties in terms of the definition of the object header, the definition of interface, and the name declarations must be present. Abstract object definitions are useful for generation of abstract types when no concrete types are needed. For example, we may use the following expression to detect if an object bound to `block` of unknown abstract type exports the names `if`, `then`, and `else`:

```
block *> [ if : Boolean; then, else :: any ];
```

If the expression evaluates to the object `true` then we can widen the view of `block` without a type error:

```
b :- [ if : Boolean; then, else :: any ];
b <-* block;
```

Activation of an abstract object is legal because a simple concrete type in fact also has been defined. This may be useful for investigation of the abstract type of the output parameter. Since the definition of an expression list is not present, the result of such an activation will usually be insignificant but may have side-effects if qualifier expressions are evaluated.

Chapter 5
Activating Objects

The Ellie expression is the fundamental mechanism for activation of new objects in Ellie. The process of an Ellie object is entirely described by a number of (embedded) Ellie expressions.

The order of activation of future objects is usually the order of evaluation, i.e., two requests for activation of future objects through the interface of an object *B* coming from an object *A* will be served in the same order as the requests are generated by *A*.

An Ellie expression can be expressed in two ways: As a field expression and as an operator expression (in both cases including an actual parameter). In the following sections, we describe these. Furthermore, we present the trust requirements for activations and describe the definition of operators and control structures.

5.1 The Ellie Field Expression

The Ellie field expression consist of a root expression that may be followed by a field list. The expression is terminated by an actual parameter that may be implicit. The syntax is shown below:

```
<expression>          ::=   <root_expression> [<field_list>]
                      |     <expression> <expression>
                      |     ...
<root_expression>     ::=   "(" <expression> ")"
                      |     <simple_expression>
                      |     <vector>
<simple_expression>   ::=   <name> | <const_constructor> |
                            <object_constructor>
<field_list>          ::=   <field> {<field>}
<field>               ::=   "." <name>
```

An expression may contain the built-in name operator `->` for the notation of 'read value of a name'. For example,

```
prototype parameter.-> [];
```

evaluates to an activated object initialized with the value referred to by the name `parameter`. The object referred to by `prototype` is used as a prototype (there is no field list). The actual parameter `parameter.-> []`, being a subexpression, is evaluated first. For convenience, Ellie allows the read value operator `->` (here being the field list) to be left out of expressions, yielding a condensed version of the Ellie expression. Therefore, `parameter.-> []` may be reduced by the programmer to just `parameter`.

A field list consist of a repeated sequence of a period and a label, constant, or variable name. A root expression followed by a field list evaluates to the object bound to the last name in the field list. This object will be the prototype. For example,

```
root_expr.remote_name_1.remote_name_2.remote_name_3
parameter_expr;
```

shows an Ellie expression where `root_expr` and `parameter_expr` may be embedded Ellie expressions or they may be names referring to objects in scope. The embedded expression `parameter_expr` is always evaluated before `root_expr` is accessed or evaluated. The names `remote_name_1, remote_name_2,` and `remote_name_3` (referring to the prototype) must be local names accessible through the interface of the object referred to by the name on the left, e.g., `remote_name_3` must be accessible through the interface of the object bound to `remote_name_2`. A closed interface will delay the access.

A root expression may also be an object constructor or a constant constructor. A constant constructor defines one of the literal objects: Integer numeral, real numeral, character, vector, or string, i.e., it evaluates to one of the built-in object types bound to one of the built-in constants: `integer`, `real`, `char`, `vector`, or `string` by an activation with a literal (its binary representation) as actual input parameter. [6]

5.2 The Ellie Operator Expression

In this section we describe the Ellie operator expression. The operators are mainly built-in constants bound to built-in objects. Operators facilitate varying precedence levels in compound expressions. The syntax is shown below:

```
<expression>        ::=  <expression> <arith_op> <expression>
                       | <expression> <relational_op>
                         <expression>
                       | <expression> <boolean_op> <expression>
                       | <expression> <index_op> <expression>
                       | <expression> <access_op> <expression>
                       | <expression> <op_name> <expression>
                       | ...
<arith_op>          ::=  "*" | "/" | "%" | "+" | "-"
<relational_op>     ::=  "<" | "=" | ">" | "<=" | "<>" | ">=" |
                         "==" | "<<>>" | "<*" | "*>"
<boolean_op>        ::=  "&" | "|"
<index_op>          ::=  "#"
<access_op>         ::=  "<-" | "->" | "<-*" | "*->"
```

6. Note, that the built-in constants may be redefined in a block different from the outermost block where they are defined already. In this way, the binary representation of the literals may be encapsulated in new types of objects to represent the literal objects.

The programmer may declare constants and variables with operator names and then take advantage of the operator expression, i.e., the precedence levels of the operators. The operators, their precedence levels, and their default semantics are shown in Table 5.1 which also shows the precedence level of programmer declared new operator names (see section 5.4).

Precedence level	Operator	Default semantics
1	# ->> *->>	Array index operator Read operator View read operator
2	<op_name>	(Programmer declared)
3	*, /, % &	Multiplication operators Logical and
4	+, - |	Additive operators Logical or
5	<, =, >, <=, <>, >= ==, <<>> <*, *> <- <-*	Relational operators Object compare operators Conformity operators Assignment operator View assignment operator

Table 5.1: Operator Precedence Levels

In compound expressions, embedded expressions with the lowest operator precedence level are evaluated first, then embedded expressions with next lowest operator precedence level, etc. Note, that an expression (or embedded expression), with all operators at the same precedence level, is evaluated from right to left. The actual parameter is always evaluated before the prototype expression on the left side. A field list is regarded as one unit, i.e., as one alphabetical name. An operator can be appended to a field list by a period, then extending the field list and losing the advantage of the operators precedence level. An operator can also be at any other place in a field list.

5.3 Trusted Activations

An object activation must always be a trusted activation. To be trusted means that both the prototype object and the actual parameter in an activation must be trusted. The main idea is to prevent illegal activations of operations by functions and to define when such activations are legal.

Let us say that the process of the object *A* wants to use the object *B* as a prototype in an activation supplying the actual parameter *P*. Subsequently, both

B and *P* must be trusted by *A*. In Table 5.2, the trust dependencies between all combinations of functions and operations defined at different block levels are presented. The letter *T* indicates trusted, whereas the letter *F* indicates distrusted. Note, that an object bound to an input parameter is regarded as defined in the same block as the input parameter name.

horizontal trusts vertical	1	2	3	4	5	6
1 - function *f* in block *B*	*T*	*F*	*T*	*T*	*T*	*F*
2 - operation *o* in block *B*	*T*	*T*	*T*	*T*	*T*	*T*
3 - function *fx* in *f*'s block	*T*	*F*	*T*	*F*	*T*	*F*
4 - operation *ox* in *f*'s block	*T*	*F*	*T*	*T*	*T*	*F*
5 - function *fy* in *o*'s block	*T*	*F*	*T*	*F*	*T*	*F*
6 - operation *oy* in *o*'s block	*T*	*T*	*T*	*T*	*T*	*T*

Table 5.2: Trust Dependencies

Ellie requires in an activation where a distrust is possible that an operation to be tested for legal block of definition is either defined by a constructor or bound to a label/constant with *manifest* value, i.e., the block of definition must be determinable statically for operations activated or supplied as actual parameters by functions (or by operations defined inside functions).

5.4 Defining Operators

The programmer is allowed to define new operator names and bind them to new objects. The built-in operators may also be redefined by the programmer in the same manner that other built-in not locally defined names may be redefined. This allows the programmer to exploit the precedence levels to reduce the number of parenthesis in compound expressions.

```
complex : [ function <: c : parameter :> self;
   export *, /, +, -, =, <>, ...;
   parameter : [
      r, i : number
   ];
   *    : [ future function <: p : complex :> complex ];
   /    : [ future function <: p : complex :> complex ];
   +    : [ future function <: p : complex :> complex ];
   -    : [ future function <: p : complex :> complex ];
   =    : [ future function <: p : complex :> Boolean ];
   <>   : [ future function <: p : complex :> Boolean ];
   ...
];
```

Figure 5.1: Complex Object

As an example, consider the abstract definition in Figure 5.1 of a complex number object (see definition of `number` in section 9.1). This definition allows us to write complex operator expressions in exactly the same style as integer and real operator expressions, for example:

```
x : complex [ r : 5; i : 2 ] [];
y : complex [ r : 9; i : 1 ] [];
v :- complex;
v <- x * y + x / y;
```

The names `x` and `y` are complex constants, whereas the name `v` is a complex variable that is bound to the value of the expression `x * y + x / y`.

5.5 Defining Control Structures

The programmer is also allowed to define new control structures. A control structure in Ellie is an object, referred to by a name bound to it, that when activated usually takes an uninitialized object (a block) as input parameter (or a vector of uninitialized objects). The control structure may then activate the block and local objects in the block depending on some values to be tested, etc. The built-in control structures (bound to the constants `option` and `loop`) are defined by using this method (see section 9.8 and 9.9).

```
for : [ function <: block : for_block;
   for_block : [ times : integer; do : any ];
   act_block : block [];  $ Activation of the block
   i :- 0;

   loop [
      while : [ :> act_block.times >= i <- i + 1 ];
      do : act_block.do
   ]
];
```

Figure 5.2: For-Loop Control Object

As an example, consider the definition of a new loop structure referred to by the name `for` shown in Figure 5.2. This definition allows us to make use of a new control structure implementing a for loop as known from, e.g., Pascal:

```
for [
   times : 5;
   do : [ write "hello world\n" ]
];
```

The evaluation of this expression, i.e., activation of the object bound to `for`, will print `hello world` on the screen five times on separate lines.

Chapter 6
Parallelism

Ellie supports parallelism by forking processes. Synchronization is supported by means of dynamic interfaces that can change over time, i.e., an object may be allowed to select between incoming requests as it pleases. By using this powerful mechanism, any other synchronization mechanism such as conditions, semaphores, and monitors can be implemented or simulated in Ellie by objects.

6.1 Forking Processes

In Ellie, forking of a new process is supported by means of asynchronous activation using the future concept. When activating a future function or future operation control is immediately returned to the activating process.

By activating a future object, the output parameter will be returned immediately (before the activation) as a future result object. A future result object will not be accessible through its interface until the future process of the activated object has terminated and hereby defined the state of the output parameter. The output parameter can not be used as a prototype in an activation before the state has been defined.

6.2 Synchronization

Parallel processes synchronize by means of an *accept interface* of another object. The syntax of interfaces is defined in section 4.3 (the keyword `accept` instead of `export`). An accept interface, in comparison to the export interface, accepts only activation requests to names explicitly included in the interface and only when the interface is open.

The interface is prepared for acceptation of activation requests to some names by activation of the object bound to the name `include`. This name refers to a built-in object defined locally within objects having accept interfaces (see section 9.12). The part of the interface to be prepared is specified by the actual parameter in the activation of `include`. The interface will be prepared to accept requests to names defined in the interface of the abstract type of the actual parameter to the activation. Names not defined in the accept interface at all will be ignored. Names not included by the activation of `include` are excluded, i.e., not prepared for acceptation of activation requests.

Like the export interface, the accept interface is closed while the process of the object is active. When the process becomes passive, the accept interface is opened (like the export interface) and all names defined in the interface will initially be included. When an activation request is accepted, the accept interface is closed.

When the process of the object activated becomes passive, the accept interface is subsequently opened again.

The process of the object with the accept interface or another process running locally within the object may activate the implicitly defined object bound to the name `include` in order to modify the interface. When the process becomes passive, the modified interface takes over. Only the process of an object activated through the interface will close and open the accept interface. The interface is not affected by a local future activation by a local process, i.e., mutual exclusion can only be guaranteed for activations through the accept interface, not locally within the object defining the interface.

An incoming request, that can not be served because the accept interface is closed or the requested name excluded, is delayed. When the interface is modified by the activation of `include` and the interface is already open, the delayed requests are inspected in FIFO order to look for the first request that can be served, if any. If a request is found then it will be served immediately. If the interface is closed when the process of the activation of `include` terminates then the delayed requests are not inspected before the interface is opened.

```
semaphore : [ function <: val :- integer;
   accept wait, signal;

   wait :: [ operation;
      val <- val - 1;
      option [
         if : val = 0;
         then : [ include [ signal :: any ]]
      ]
   ];

   signal :: [ operation;
      val <- val + 1;
      option [
         if : val = 1;
         then : [ include [ wait, signal :: any ]]
      ]
   ]
];
```

Figure 6.1: Semaphore Object

As an example, consider the Ellie implementation of semaphores shown in Figure 6.1. In this example, the queue of the semaphore consists of suspended requests for activation of `wait`. The implementation is used in the solution to the Dining Philosophers problem (see section 7.3).

If an object with an accept interface is referred to by a name `accept_obj` in a field list and `accept_obj` is not one of the two last names of the list then a remote addressing will be suspended at `accept_obj` if `accept_obj` is excluded. The addressing will continue when `accept_obj` has been included.

Chapter 7
Writing Programs

Ellie is block structured and object constructors define blocks within which new objects/blocks may be locally defined. An Ellie program consist of at least the declaration of the name `program` bound to an object constructor as the qualifier and value. Furthermore, there may be some more names defined in the same scope as `program`. This is a technique for fragmenting programs in modules. This section illustrates Ellie programs written in this style. The section also describes polymorphism in Ellie and shows a solution to the Dining Philosophers problem.

7.1 Modules

The outermost object in the Ellie block structure is called the *block zero object*. It is a built-in object that is activated by the Ellie program development system when this is entered. The block zero object encapsulates some built-in objects defining standard types, etc. (see section 9).

```
[
   $ The block zero object

   $ Abstract definition of built-in names and objects
   number  : [ ... ];
   integer : [ ... ];
   real    : [ ... ];
   ...
   any     : [ ... ];
   none    : [ ... ];

   $ Programmer definition of modules
   user_module_1 : [ ... ];
   user_module_2 : [ ... ];
   ...
   user_module_N : [ ... ];

   program : [
      $ The first object to be activated when the program
      $ starts its execution
   ];

   program []
];
```

Figure 7.1: Block Zero Skeleton Object

The block zero object also encapsulates the name `program` bound to a programmer defined object to be activated as the first object when a program is executed. This object may be parted in local modules since modules are only

objects. Modules may also be defined at the same block level as the name `program` and they may also be parted in local modules.

A block zero skeleton object is shown in Figure 7.1. The highlighted part of this skeleton is the Ellie program. The remaining parts of the block zero object is implicitly defined by the Ellie program development system and merged with the programmers program at compile-time. The block zero object will be activated when the program is executed.

7.2 Polymorphism

Ellie supports type polymorphism, i.e., a manifest object may statically be specialized by the actual parameter in an activation. A specialization requires that the qualifier and/or value of at least one local defined name depends on the actual parameter.

```
square : [ future function <: p : parameter :> result :- p;
   parameter : [ future operation;
      export *;
      * :: [ future function <: any :> none ]
   ];
   result <- p * p
];
```

Figure 7.2: Polymorphic Square Function

Figure 7.2 shows the implementation of a type independent square function. This polymorphic future function can multiply any input parameter with itself. The input parameter must export a name `*` that in an activation returns a value conforming to the actual parameter to the activation of `square`. As an example, consider the following activations:

```
square 10;
square 2.5;
square (v * 2);
```

The output parameter of the first activation will always conform to the type `integer`, whereas it will conform to the type `real` in the second activation. The output parameter of the last activation will conform to the type of the output parameter of the operator `*` exported by the object bound to variable `v`.

7.3 Dining Philosophers

The program example shown in Figure 7.3 is a deadlock and starvation free implementation in Ellie of Dijkstra's Dining Philosophers problem. This is an example where process synchronization is done by an Ellie implementation of semaphores (see section 6.2).

```
program :: [

   fork :: vector
      [ length : 5; of : [ elem : semaphore 1 ] []] [];

   room : semaphore 4;

   philosopher :: [ future operation <: no : integer;
      loop [
         $ Think
         room.wait [];
         (fork#no).wait [];
         (fork#((no + 1) % 5)).wait [];
         $ Eat
         (fork#no).signal [];
         (fork#((no + 1) % 5)).signal [];
         room.signal []
      ]
   ];

   philosopher 0;
   philosopher 1;
   philosopher 2;
   philosopher 3;
   philosopher 4
];
```

Figure 7.3: Dining Philosophers Program

Chapter 8
Notation and Vocabulary

The syntax of Ellie programs is defined in *Extended Backus Naur Form* (EBNF) in section 8.6. Terminal symbols in Ellie are shown as strings enclosed in quotes, e.g., `":"` represents a colon (reserved word) and `"function"` represents the reserved word `function`. Non-terminal symbols are denoted by names (may include underscores) enclosed in angled brackets, e.g., <int_numeral> represents an integer numeral.

In EBNF, alternatives are indicated by | (vertical bar), whereas optional symbols are enclosed in square brackets and (possibly empty) repeated sequences of symbols in are enclosed in braces.

8.1 Legal Names

Names in Ellie are used for keywords, special symbols, programmer declared label, constant, and variable names, and built-in constant names. Alphabetical names consist of letters from the Roman alphabet, underscores, and digits, whereas operator names consist of the special operator characters (see grammar rule <op_character> in section 8.6). Alphabetical names may not have a digit as the first character (underscore is allowed).

The following characters are reserved characters and may not be used in operator names:

```
.   ,   ;   [   ]   (   )   {   }   "   '   $
```

These characters all have a special meaning when defining Ellie objects.

Uppercase and lowercase characters are considered identical. Names can be arbitrary long and all characters are significant.

8.2 Reserved Names

Only a small number of names are reserved for keywords and special symbols. This is because built-in objects and names define standard language constructs such as assignment, loop structures, and selective structures, etc., and types such as integers and characters, etc. The reserved names can not be used for declaration of new names. The reserved names are shown below (keywords first, special symbols next):

```
future      express     function    operation
export      accept
::          :           :-          <:          :>
```

These reserved names can all be found in the context-free grammar in contrast to the built-in names presented below.

8.3 Built-in Names

Built-in constant names are declared by the compiler and bound to built-in objects and objects defined by the programmer. The built-in names are shown below:

```
number      integer     real        Boolean     char
vector      lit_vector
string      lit_string
option      loop        read        write
binary_value
include     none        any         any_fct     self
==          <<>>        <*          *>
<-          ->          <-*         *->
```

The names may all be used in the same way as names defined by the programmer. The names `binary_value`, `include`, `none`, `any`, `any_fct` and `self` can not be redefined because they are all implicitly defined locally in any object, except the name `include` which is only defined locally in objects having an accept interface. Many of the built-in objects bound to these names locally declare more built-in names that may be reached through the interfaces. These names (labels and constants), and abstract definitions of the built-in objects, are listed in section 9.

8.4 Literal Objects

Literal objects in Ellie may be divided into the following categories:

Integer numerals The syntax of integer numerals is defined by grammar rule <int_numeral> in section 8.6. Integer numerals are defined as activations of the built-in object bound to the built-in constant `integer` (or a local redefinition of it) with the binary representation as parameter. The range is from -2147483648 to 2147483647 (32 bit).

Real numerals The syntax of real numerals is defined by grammar rule <real_numeral> in section 8.6. Real numerals are defined as activations of the built-in object bound to the built-in constant `real` (or a local redefinition of it) with the binary representation as parameter. Reals are defined by the IEEE 754 64 bit double precision floating point standard [IEEE 85].

Booleans The Boolean literals are the built-in constants `true` and `false` bound to activations of the built-in object bound to the built-in constant `Boolean` (these constants may all be locally redefined).

Characters The syntax of characters is defined by grammar rule <character> in section 8.6. Characters are defined as activations of the built-in object bound to the built-in constant `char` (or a local redefinition of it) with the binary representation as

parameter. The printable range is the 7-bit ASCII character set; the value range is from 0 to 255.

Vectors The syntax of constant vectors is defined by grammar rule <vector> in section 8.6. Constant vectors are created by activation of the built-in object bound to the built-in constant `lit_vector` (or a local redefinition of it) with the binary representation of the objects to be put in the vector as parameter. The abstract element type of the constant vector is the *greatest common abstract type* [7] that all the elements conform to. Vectors can have at most 255 element objects.

Strings The syntax of constant strings, which are constant vectors of characters, is defined by grammar rule <string> in section 8.6. Strings are created by activation of the built-in object bound to the built-in constant `lit_string` (or a local redefinition of it) with the binary representation of the characters to be put in the string as parameter.

8.5 Separators

Distinct names must be separated by at least one separator or one reserved character. [8] Because digits may be used in a name (except the first character of the name), a name and a succeeding numeral, must also be separated. The same is the case for a real numeral without exponential part followed by the letter *e* or *E* in order to express the correct semantics. Also an operator name and a following signed numeral must be separated from each other. The allowed separators are the following:

NL/LF	(newline/linefeed)
TAB	(tabulator)
SP	(space)
`$ string`	(comment)

A comment is defined as a dollar sign followed by a character string terminated by *NL* or *LF* (the same ASCII character value). Other control characters in an Ellie program text are ignored.

8.6 The Syntax of Ellie

The context-free grammar for Ellie programs defined in *Extended Backus Naur Form* (EBNF) is shown below:

---

7. The greatest common abstract type is found by including all common properties of the abstract types of each element. This means names that are in all interfaces etc.

8. An operator name followed by an alphabetical name, or vice versa, do not need to be separated from each other.

```
1.   <program>              ::=  <name> <binding_selector>
                                  <object_constructor> [";"]
2.   <object_constructor>   ::=  "[" [<object_pattern>] "]"
3.   <object_pattern>       ::=  <object_header> [";" <object_tailer>] [";"]
                           |    <object_tailer> [";"]
4.   <object_header>        ::=  ["future"] ["express"]
                                  <object_kind> [<parameter_part>]
                           |    <parameter_part>
5.   <object_kind>          ::=  "function" | "operation"
6.   <parameter_part>       ::=  <input_parameter> [<output_parameter>]
                           |    <output_parameter>
7.   <input_parameter>      ::=  "<:" <parameter>
8.   <output_parameter>     ::=  ":>" <parameter>
9.   <parameter>           ::=  [<name> <binding_selector>] <expression>
10.  <binding_selector>     ::=  "::" | ":" |":-"
11.  <object_tailer>        ::=  <interface> [";" <object_body>]
                           |    <object_body>
12.  <interface>            ::=  <interface_kind> [<access_list>]
13.  <interface_kind>       ::=  "export" | "accept"
14.  <access_list>          ::=  <access_item> {"," <access_item>}
15.  <access_item>          ::=  <access_selector> [<remote_selector>]
16.  <access_selector>      ::=  "(" <name> ")"
                           |    <name>
17.  <remote_selector>      ::=  "." "(" [<name_list>] ")"
18.  <object_body>          ::=  <binding_section> [";" <expression_list>]
                           |    <expression_list>
19.  <binding_section>      ::=  <binding_list> {";" <binding_list>}
20.  <binding_list>         ::=  <name_list> <binding_selector>
                                  <expression>
21.  <name_list>            ::=  <name> {"," <name>}
22.  <expression_list>      ::=  <expression> {";" <expression>}
23.  <expression>           ::=  <expression> <arith_op> <expression>
                           |    <expression> <relational_op>
                                  <expression>
                           |    <expression> <boolean_op> <expression>
                           |    <expression> <index_op> <expression>
                           |    <expression> <access_op> <expression>
                           |    <expression> [<op_name>] <expression>
                           |    <root_expression> [<field_list>]
24.  <arith_op>             ::=  "*" | "/" | "%" | "+" | "-"
25.  <relational_op>        ::=  "<" | "=" | ">" | "<=" | "<>" | ">=" |
                                  "==" | "<<>>" | "<*" | "*>"
26.  <boolean_op>           ::=  "&" | "|"
27.  <index_op>             ::=  "#"
28.  <access_op>            ::=  "<-" | "->" | "<-*" | "*->"
29.  <root_expression>      ::=  "(" <expression> ")"
                           |    <simple_expression>
                           |    <vector>
30.  <simple_expression>    ::=  <name> | <const_constructor> |
                                  <object_constructor>
31.  <field_list>           ::=  <field> {<field>}
```

32. <field> ::= "." <name>
33. <const_constructor> ::= <numeral> | <character> | <string>
34. <numeral> ::= <int_numeral> | <real_numeral>
35. <vector> ::= "{" [<element_list>] "}"
36. <element_list> ::= <expression> {"," <expression>}
37. <name> ::= <alpha_name> | <op_name>

**** There must be no separators in the grammar rules below ****

38. <alpha_name> ::= <alpha_character>
{<alpha_character> | <digit>}
39. <op_name> ::= <op_character> {<op_character>}
40. <int_numeral> ::= ["+" | "-"] <digit> {<digit>}
41. <real_numeral> ::= <int_numeral> "." <digit> {<digit>}
[<eksponent>]
42. <eksponent> ::= ["E" | "e"] <int_numeral>
43. <character> ::= "'" <ascii_character> "'"
44. <string> ::= """ {<ascii_character>} """
45. <alpha_character> ::= "A" | "B" | ... | "Z" |
"a" | "b" | ... | "z" | "_"
46. <op_character> ::= "!" | "#" | "%" | "&" | "*" | "+" | "-" |
"/" | ":" | "<" | "=" | ">" | "?" | "@" |
"\" | "^" | "`" | "~"
47. <ascii_character> ::= "\\" | "\n"
| "\" <digit> [<digit> [<digit>]]
| <graphic_character>
48. <digit> ::= "0" | "1" | ... | "9"

Since built-in objects and names define standard language constructs such as assignment, loop structures, and selective structures, etc., and types such as integers and characters, etc., these language constructs and types are not included in the context-free grammar. They all are objects and behave like user defined objects implying that they may even be redefined by the programmer.

Chapter 9
Built-in Objects

This section defines the built-in objects by means of abstract objects bound to built-in names. Most of these built-in objects have local definitions of other built-in objects and declarations of built-in label and constant (operator) names bound to these local objects. The abstract object definitions are commented in order to define the default semantics of the processes.

9.1 Number

The built-in constant `number` defined at the outermost block level is bound to the object defined in Figure 9.1.

9.2 Integer

The built-in constant `integer` defined at the outermost block level is bound to the object defined in Figure 9.2.

9.3 Real

The built-in constant `real` defined at the outermost block level is bound to the object defined in Figure 9.3.

9.4 Boolean

The built-in constant `Boolean` and the constants `false` and `true` of the same type as `Boolean` are defined at the outermost block level and bound to the object defined in Figure 9.4 (and activations of it).

The values of `false` and `true` (serving as Boolean literals) are defined by the activations below:

```
false : Boolean 0.to_binary [];
true  : Boolean 1.to_binary [];
```

9.5 Char

The built-in constant `char` defined at the outermost block level is bound to the object defined in Figure 9.5.

9.6 Vector

The built-in constant `vector` defined at the outermost block level is bound to the object defined in Figure 9.6.

The index operator # serves for vector indexing. This built-in operator takes a left-side operand of type `vector` and a right-side operand of type `integer` (the index). The index expression evaluates to a vector element object of the element type to which the vector was initialized when activated. [9] When an index is out of range an exception is generated.

In case of a vector literal, for example

```
{ integer, real, number, 3.1415, 10 }
```

an object referred to by the name `lit_vector` is activated with the binary representation of the vector literal as actual parameter, returning a constant vector:

```
lit_vector : [ function <: val : binary_value
   :> vector
];
```

The abstract element type of the returned vector is the greatest common abstract type that all the elements conform to which in the example is the type `number`.

9.7 String

The built-in constant `string` defined at the outermost block level is bound to the object defined below (a non-abstract implementation activating `vector`):

```
string : [ function <: len : integer
   :> vector [ length : len; of : [ elem : char ]]
];
```

The first vector element has the index 0, the second the index 1, etc. For example,

```
"This is a vector of characters" # 2
```

evaluates to an element object of type `char` initialized to the character *i*. In case of a string literal, as in this example, an object referred to by the name `lit_string` is activated with the binary representation of the string literal as actual parameter, returning a constant string:

```
lit_string : [ function <: val : binary_value
   :> string
];
```

9.8 Option

The built-in constant `option` defined at the outermost block level is bound to the object defined in Figure 9.7.

When activated, the selective control object will activate `block` and then test the Boolean value of `block.if`. If this is true then `block.then` is activated,

---

9. The expression may evaluate to an element object that only conforms to the element type the vector was initialized to, but in this case it is necessary to change view (widen it).

otherwise, if the expression `block *> else_block` is true then `block.else` is activated.

9.9 Loop

The built-in constant `loop` defined at the outermost block level is bound to the object defined in Figure 9.8.

When activated, the loop control object will activate `block`. If the expression `block *> while_block` is true then `block.while` is activated. While this activation returns true, `block.do` will be activated. Otherwise, if the expression `block *> until_block` is true then `block.do` is activated. While the activation of `block.until` returns false, `block.do` will be activated. If `block` does not conform to any of the two block types then `block` will be activated an infinite number of times.

9.10 Read

The built-in constant `read` defined at the outermost block level is bound to the object defined below:

```
read : [ operation <: length : integer :> string ];
```

When activated, this object will try to read the number of characters given by the actual parameter from the input stream. If fewer characters are available then a shorter constant string will be returned. The characters are put one by one into the constant string in the index sequence. In case of parallel activations, one string will be read after another in an undeterminable order but strings will not be read interleaved.

9.11 Write

The built-in constant `write` defined at the outermost block level is bound to the object defined below:

```
write : [ operation <: str : string :> none ];
```

When activated, this object will write all the characters from the actual parameter to the output stream. The characters are written one by one in the index sequence. In case of parallel activations, one string will be written after another in an undeterminable order but strings will not be written interleaved.

```
number : [ function <: val : binary_value :> self;
   export to_integer, to_real, to_string, to_binary;
   to_integer : [ future function :> integer ];
   to_real    : [ future function :> real ];
      $ Converts the number to an integer number or a real
      $ (exception if a real number can not be represented)
   to_string  : [ future function :> string ];
      $ Converts the number to a character string using the
      $ format of 'integer.to_string' or 'real.to_string'
   to_binary  : [ future function :> binary_value ]
      $ Returns the binary representation
];
```

Figure 9.1: Number Object

```
integer : [ function <: val : binary_value :> self;
   export *, /, %, +, -, <, =, >, <=, <>, >=,
      to_integer, to_real, to_char, to_string, to_binary;
   *  : [ future function <: p : integer :> integer ];
      $ Returns the product (exception on overflow)
   /  : [ future function <: p : integer :> integer ];
      $ Returns the whole part (exception on zero)
   %  : [ future function <: p : integer :> integer ];
      $ Returns the rest from division (exception on zero)
   +  : [ future function <: p : integer :> integer ];
      $ Returns the sum (exception on overflow)
   -  : [ future function <: p : integer :> integer ];
      $ Returns the difference (exception on overflow)
   <  : [ future function <: p : integer :> Boolean ];
   =  : [ future function <: p : integer :> Boolean ];
   >  : [ future function <: p : integer :> Boolean ];
   <= : [ future function <: p : integer :> Boolean ];
   <> : [ future function <: p : integer :> Boolean ];
   >= : [ future function <: p : integer :> Boolean ];
      $ Compares the two integers
   to_integer : [ future function :> integer ];
      $ Returns the integer number itself
   to_real    : [ future function :> real ];
      $ Converts the integer number to a real number
   to_char    : [ future function :> char ];
      $ Converts the integer number to a character
      $ (exception on integer outside the range 0-255)
   to_string  : [ future function n : integer :> string ];
      $ Converts the integer number to a string of n
      $ characters (right justified)
      $ If n is negative then the number is converted to -n
      $ characters (left justified)
      $ If the integer number extends the length of the
      $ string then digit(s) are omitted in the unjustified
      $ side of the field
      $ If n is zero then the string length will be equal to
      $ the number of digits in the number
   to_binary  : [ future function :> binary_value ]
      $ Returns the binary representation
];
```

Figure 9.2: Integer Object

```
real : [ function <: val : binary_value :> self;
   export *, /, +, -, <, =, >, <=, <>, >=,
      to_integer, to_real, to_string, to_binary;
   *  : [ future function <: p : real :> real ];
      $ Returns the product (exception on overflow)
   /  : [ future function <: p : real :> real ];
      $ Returns a fraction (exception on zero)
   +  : [ future function <: p : real :> real ];
      $ Returns the sum (exception on overflow)
   -  : [ future function <: p : real :> real ];
      $ Returns the difference (exception on overflow)
   <  : [ future function <: p : real :> Boolean ];
   =  : [ future function <: p : real :> Boolean ];
   >  : [ future function <: p : real :> Boolean ];
   <= : [ future function <: p : real :> Boolean ];
   <> : [ future function <: p : real :> Boolean ];
   >= : [ future function <: p : real :> Boolean ];
      $ Compares the two real numbers
   to_integer : [ future function :> integer ];
      $ Converts the real number to the nearest integer
      $ number (exception if it can not be represented by an
      $ integer number)
   to_real    : [ future function :> real ];
      $ Returns the real number itself
   to_string  : [ future function n : real :> string ];
      $ The whole part of n defines the string length and
      $ the justification in the same way as for
      $ integer.to_string
      $ The fraction part of n defines the number of digits
      $ following the decimal point
      $ If the fraction part of n is zero then a scientific
      $ normalized signed notation is used and the number of
      $ digits following the decimal point is the string
      $ length minus 8, e.g., +3.1415e+000
      $ (Rounding is used)
   to_binary  : [ future function :> binary_value ]
      $ Returns the binary representation
];
```

Figure 9.3: Real Object

```
Boolean : [ function <: val : binary_value :> self;
   export &, |, not, to_integer, to_string;
   &   : [ future function <: p : Boolean :> Boolean ];
      $ If both binary values are different from zero, the
      $ result is true, otherwise it is false
   |   : [ future function <: p : Boolean :> Boolean ];
      $ If one of the binary values are different from zero,
      $ the result is true, otherwise it is false
   not : [ future function :> Boolean ];
      $ If the binary value is equal to zero, the result is
      $ true, otherwise it is false
   to_integer : [ future function :> integer ];
      $ Converts the boolean value to an integer number (may
      $ be different from zero or one)
   to_string  : [ future function :> string ];
      $ Converts the Boolean value to the character string
      $ "false" (5 characters) when the binary value is
      $ equal to zero, otherwise "true" (4 characters)
   to_binary  : [ future function :> binary_value ]
      $ Returns the binary representation (may be different
      $ from zero or one)
];
```

Figure 9.4: Boolean Object

```
char : [ function <: val : binary_value :> self;
   export +, -, <, =, >, <=, <>, >=, to_integer, to_binary;
   +  : [ future function <: p : integer :> char ];
      $ Returns the increased character
      $ (exception on overflow)
   -  : [ future function <: p : integer :> char ];
      $ Returns the decreased character
      $ (exception on overflow)
   <  : [ future function <: p : char :> Boolean ];
   =  : [ future function <: p : char :> Boolean ];
   >  : [ future function <: p : char :> Boolean ];
   <= : [ future function <: p : char :> Boolean ];
   <> : [ future function <: p : char :> Boolean ];
   >= : [ future function <: p : char :> Boolean ];
      $ Compares the character with an integer number
   to_integer : [ future function :> integer ];
      $ Converts the character to an integer number
   to_binary  : [ future function :> binary_value ]
      $ Returns the binary representation
];
```

Figure 9.5: Character Object

```
vector : [ function <: p : parameter :> self;
   export length, elem_type, #;
   parameter   : [ length : integer; of : [ elem : any ]];
   length      : p.length;
   elem_type   : p.of.elem;
   return_type : [ export (of); of :: p.of ];
   #           :
      [ future function <: i : integer :> return_type ]
      $ Returns an object that behaves like a constant or a
      $ variable name (same interface)
      $ For i = 0, the first element of the vector will be
      $ returned; for i = 1, the second will be returned,
      $ etc. (exception on index i outside the legal range)
];
```

Figure 9.6: Vector Object

```
option : [ function <: block : if_block :> none;
   if_block   : [ if : Boolean; then :: any ];
   else_block : [ if : Boolean; then, else :: any ]
];
```

Figure 9.7: Selective Control Object

```
loop : [ function <: block : loop_block :> none;
   loop_block  : any;
   while_block : [ while ::
      [ future operation :> Boolean ]; do :: any ];
   until_block : [ until ::
      [ future operation :> Boolean ]; do :: any ]
];
```

Figure 9.8: Loop Control Object

9.12 Implicit Objects

The built-in implicit simple objects defined and activated when declaring names (in the new scope of the names) are not bound to names because they represent names themselves. The abstract definitions are shown by three figures (a label name in Figure 9.9, a constant name in Figure 9.10, and a variable name in Figure 9.11).

Note, that a name exports the interface of its value as though the object was included in the name object (inheritance implemented by interface inclusion). Because the implicit object representing a label name only exports the interface of the value (the part of the interface defined by the labels qualifier), all the operator names of constant names and variable names can be declared and exported by the value. By using this, constants and variables can be simulated, and new kind of protected names, etc., can be implemented.

The `*>` operator compares the two operands for conformity. For example,

```
a_number *> integer
```

evaluates to `true` only if the object bound to `a_number` conforms to the object bound to `integer`. The operator `<*` compares for conformity in the other direction.

The operator == represents the object compare operator. For example,

```
integer_1 == integer_2
```

evaluates to `true` only if the objects bound to `integer_1` and `integer_2` are *equal*, otherwise to `false`. Two objects are said to be equal, i.e., the same object, if they have the same reference or if they are immutable and binary identical. Note that, in the latter case, the objects bound to local names are also compared. This continues down to any level until a binary difference is found or until all objects in the two object structures have been compared. The object compare operator does not guarantee that two identical immutable objects are indeed the same object. This can not be verified in Ellie.

Within programmer defined objects five or six names are implicitly declared and bound to implicitly defined (simple) objects:

```
binary_value
include    none       any        any_fct    self
```

The constant name `self` is always bound to activations of the programmer defined object within which it is defined implicitly. The label name `include` is only defined implicitly if the programmer defined object has an accept interface. The abstract definitions of the implicitly defined simple objects with a semantic description are shown in Figure 9.12 [10].

10. Note, that the definitions of the objects bound to `any` and `any_fct` are fully implemented (not abstract). Also note, that the abstract definition of the object bound to `none` is impossible to write in Ellie because the object exports an infinite number of names.

```
[ function <: p : parameter :> self;
   $ A label name
   export (value);
   parameter : [ function;
      export qualifier, value;
      qualifier : any;
      value     : qualifier
   ];
   value : p.value
];
```

Figure 9.9: Label Name Object

```
[ function <: p : parameter :> self;
   $ A constant name
   export ->, *->, ==, <<>>, <*, *>, (value);
   parameter : [ function;
      export qualifier, value;
      qualifier : any;
      value     : qualifier
   ];
   value : p.value;
   ->    : [ future function :> value ];
      $ Read the value bound to the name
   *->   : [ future function :> value_type ];
      $ Read the name qualifier
   ==    : [ future function <: o : any :> Boolean ];
      $ Compares for object equality
   <<>>  : [ future function <: o : any :> Boolean ];
      $ Compares for object non-equality
   <*    : [ future function <: o : any :> Boolean ];
      $ Compares for input parameter to value conformity
   *>    : [ future function <: o : any :> Boolean ]
      $ Compares for value to input parameter conformity
];
```

Figure 9.10: Constant Name Object

```
[ function <: p : parameter :> self;
   $ A variable name
   export ->, *->, ==, <<>>, <*, *>, <-, <-*, (value);
   parameter : [ function;
      export qualifier, value;
      qualifier : any;
      value     : qualifier
   ];
   value :- p.value;
   ->    : [ future function :> value ];
      $ Read the value bound to the name
   *->   : [ future function :> qualifier ];
      $ Read the name qualifier
   ==    : [ future function <: o : any :> Boolean ];
      $ Compares for object equality
   <<>>  : [ future function <: o : any :> Boolean ];
      $ Compares for object non-equality
   <*    : [ future function <: o : any :> Boolean ];
      $ Compares for input parameter to value conformity
   *>    : [ future function <: o : any :> Boolean ];
      $ Compares for value to input parameter conformity
   <-    :
      [ operation <: new_value : qualifier :> value ];
      $ Rebinds the name to a new value as an atomic
      $ operation (one machine instruction)
   <-*   : [ operation <: new_value : any :> value ]
      $ Rebinds the name to a new value as an atomic
      $ operation (one machine instruction) if the new value
      $ conforms to the qualifier, otherwise an exception is
      $ generated
];
```

Figure 9.11: Variable Name Object

```
binary_value : [ function <: none :> none;
   $ A binary representation of an integer number, a real
   $ number, a character, or a vector of objects of any type
   $ (activation results in an exception)
];

include :: [ operation <: p : any :> integer;
   $ Prepares the accept interface to accept names which are
   $ in the interface of the input parameter object; this
   $ is done as an atomic operation to avoid undeterminable
   $ results of parallel activations
];

none : [ function <: any :> none;
   export a, b, c, ..., z, _, aa, ab, ac, ..., a_, a0, ...
      a9, aaa, aab, aac, ..., aa_, aa0, ..., aa9, ...;
   a, b, c, ..., z, _, aa, ab, ac, ..., a_, a0, ... a9,
   aaa, aab, aac, ..., aa_, aa0, ..., aa9, ... :: none;
   $ An object exporting all names which can be defined in
   $ a program (activation results in an exception)
];

any : [ future operation <: none :> self ];

any_fct : [ future function <: none :> self ];
```

Figure 9.12: Simple Objects

References

[Agha 87] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49-74, The MIT Press, 1987

[Andersen 91a] Birger Andersen. *Fine-Grained Parallelism in Ellie*. PhD thesis (partial), DIKU Report no. 91/4, Department of Computer Science, University of Copenhagen, Denmark, June 1991. Also to appear in *Journal of Object-Oriented Programming*.

[Andersen 91b] Birger Andersen. *Grain-Size Adaption in the Fine-Grained Object-Oriented Language Ellie*. PhD thesis (partial), DIKU Report no. 91/5, Department of Computer Science, University of Copenhagen, Denmark, 1991. In preparation.

[Athas 87] William C. Athas. *Fine Grain Concurrent Computations*. PhD thesis, Technical Report 5242, Department of Computer Science, California Institute of Technology, Pasadena, California, May 1987.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65-76, January 1987. Also Technical Report 86-02-04, Department of Computer Science, University of Washington.

[Brinch Hansen 75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199-207, June 1975.

[Brinch Hansen 77] Per Brinch Hansen. *The Architecture of Concurrent Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[Brinch Hansen 87] Per Brinch Hansen. Joyce - a programming language for distributed systems. *Software - Practice and Experience*, 17(1):29-50, January 1987.

[Campbell 74] R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions in operating systems. In *Operating Systems Proceedings*, April 1974. Lecture Notes in Computer Science, (16):89-102, Springer-Verlag, 1974.

[Goldberg 89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Publishing Company, 1989.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, Technical Report 87-01-01, Department of Computer Science, University of Washington, Seattle, Washington, January 1987.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language*. DIKU Report no. 87/22, Department of Computer Science, University of Copenhagen, Denmark, October 1987. Also Technical Report 87-10-07, Department of Computer Science, University of Washington.

[IEEE 85] IEEE. *Binary Floating-Point Arithmetic*. IEEE Standard 754, IEEE, New York, 1985.

[Kristensen 87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7-48, The MIT Press, 1987.

[Madsen 87] Ole Lehrmann Madsen. Block-structure and object-oriented Languages. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 113-128, The MIT Press, 1987.

[Raj 91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software - Practice and Experience*, 21(1):91-118, January 1991.

[Shibayama 87] Etsuya Shibayama and Akinori Yonezawa. Distributed computing in ABCL/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 91-128, The MIT Press, 1987.

[Yokote 86] Yasuhiko Yokote and Mario Tokoro. The design and implementation of ConcurrentSmalltalk. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1986. ACM SIGPLAN Notices, 21(11):331-340, November 1986.

Index

: 17, 37
:: 17, 37
:- 17, 37
:> 20, 37
$ 39
-> 27, 38
*-> 27, 38
*> 27, 38, 50
<: 20, 37
<- 27, 38
<-* 27, 38
<* 27, 38, 50
<<>> 27, 38
== 27, 38, 50
Abstract definitions 23, 50
Abstract object 2, 13, 14, 19, 20, 23, 29, 31, 33, 38, 39, 43, 44, 50, 55
Abstraction 8, 13
Accept 37
Accept interface 31, 32, 38, 50, 53
Access list 22
Activation 8, 25
Activation requests 31
Active objects 8
Actors 7, 55
Actual input parameter 26
Actual parameter 10, 11, 13, 20, 21, 25, 27, 31, 34, 44, 45
Alphabetical names 37
Any 38, 50
Any_fct 38, 50
Beta 7, 56
Binary representation 26, 38, 39, 44, 46-48, 53
Binary_value 38, 50
Binding kinds 17
Binding section 20
Binding selector 17
Block zero 33, 34
Boolean 38, 43
Boolean literals 38
Built-in names 18, 38, 43
Built-in objects 38
Change view 14, 44
Char 38, 43
Character literals 38
Child processes 8
Classes 7, 8
Closed interface 9
Coarse-grained 7
Compound expressions 27
Conformity 7, 13-15, 17, 27, 50-52
Constant constructor 26
Constant name 17, 50
Context-free grammar 39
Control structures 29
Data abstraction 13
Delayed requests 32
Delegation 7, 14
Dining Philosophers 32-35
Distributed computer 7
Distributed memory 2, 7, 11
Dynamic interface 8
EBNF 17, 37, 39
Element type 44
Ellie expression 11, 25
Ellie objects 8
Emerald 7, 13, 55, 56
Encapsulation 13, 33
Excluded names 31
Export 21, 37
Export interface 21, 31
Express 37
Express objects 10
Expression 25
Expression list 20, 23
False 43
Fibonacci function 12
FIFO order 32
Fine-grained 2, 7, 8, 11, 12, 55
First class objects 8
For-loop control object 29
Formal parameter 13
Function 37
Functional process 10
Functions 8, 11, 20, 27, 28
Future 37
Future objects 10, 15, 19, 25
Future process 9
Future result object 11, 31
Grain-size adaption 7, 8, 55
Granularity 7, 11, 12
Immutable object 9

Implicit objects 17, 21, 22, 25, 50
Include 38, 50
Included names 31
Inclusion 7, 22, 23, 50
Inheritance 7, 22, 50
Initialized objects 8
Initializing objects 8
Input parameter 11-13, 19, 20, 26, 28, 29, 34, 51-53
Input stream 45
Integer 38, 43
Interface inclusion 7, 22, 23, 50
Interfaces 9
Keywords 2, 20, 37
Label name 17, 50
Legal names 37
Lit_string 38, 44
Lit_vector 38, 44
Literal objects 26, 38
Local object 10, 11
Loop 38, 45
Loop control object 45
Manifest values 28
Master processes 8
Medium-grained 7
Modules 33, 34
Multiple inheritance 22
Mutable object 9
Mutual exclusion 8, 32
Name conformity 14
Name Declarations 17, 23
Name kind 14, 17
None 38, 50
Number 38, 43
Numeral literals 38
Object activation 10, 18, 27
Object compare operator 50
Object constructor 9, 10, 18-20, 23, 26, 33
Object definition 9, 13, 19
Object header 21, 23
Object interface 21
Object pattern 19, 20
Object relations 10
Object tailer 20
Objects 8
Opened interface 9
Operation 37
Operational process 10
Operations 8, 20, 27, 28
Operator characters 37
Operator definition 28
Operator expression 26
Operator names 37
Operators 12, 25-28
Option 38, 44
Origin object 10, 11, 13
Output parameter 9, 11, 13, 17, 18, 20, 21, 23, 31, 34
Output stream 45
Parallelism 31
Parameter passing 20
Passive objects 8
Path Expressions 8, 55
Polymorphic square function 34
Polymorphism 33, 34
Precedence levels 26-28
Process conformity 13, 15
Process forking 8, 31
Processes 8
Programs 33
Prototypes 7, 8, 10
Qualifier 13
Read 38, 45
Real 38, 43
Record object 20
References 8
Remote name 21, 22
Remote procedure call 9
Remote selector 22
Representations 8
Reserved characters 37
Reserved names 37, 39
Reserved words 37
RPC 9
Scope rules 18
Selective control object 44
Self 38, 50
Semaphore 9, 32, 35
Separators 39, 41
Side-effects 8, 10, 20, 23
Simple objects 17, 50, 53
Smalltalk 7, 55
Special symbols 37
String 38, 44
String literals 39, 44
Suspended process 9
Synchronization 8, 9, 31, 34, 55
Syntax of Ellie 39
True 43
Trust dependencies 28
Trusted activations 27

Trusted processes 20
Type checking 7, 14
Type system 13
Unbounded remote procedure call 9
Uninitialized objects 8
URPC 9
Variable name 17, 50
Vector 38, 43
Vector literals 39, 44
Vocabulary 37
Widen view 14
Write 38, 45