

Operation	Description
<code>R.F</code>	Return field <code>F</code> from <code>R</code>
<code>{HasFeature R F}</code>	Return boolean saying whether feature <code>F</code> is in <code>R</code>
<code>{IsRecord R}</code>	Return boolean saying whether <code>R</code> is of record type
<code>{MakeRecord L Fs}</code>	Return record with label <code>L</code> and features <code>Fs</code>
<code>{Label R}</code>	Return the label of <code>R</code>
<code>{Arity R}</code>	Return the list of features (arity) of <code>R</code>
<code>{Record.toList R}</code>	Return the list of fields of <code>R</code> , in <code>Arity</code> order
<code>{Width R}</code>	Return the number of features (width) of <code>R</code>
<code>{AdjoinAt R F X}</code>	Return <code>R</code> augmented with feature <code>F</code> and value <code>X</code>
<code>{Adjoin R1 R2}</code>	Return <code>R1</code> augmented with all fields of <code>R2</code>

Table B.8: Some record operations

B.3.2 Operations on records

Table B.8 gives a few basic record operations. Many more operations exist in the Base module `Record`. This appendix shows only a few, namely those concerning extracting information from records and building new records. To select a field of a record component, we use the infix dot operator, e.g., `tree(key:I value:Y LT RT).value` returns `Y`. To compare two records, we use the equality test operation. Two records are the same if they have the same set of features and the language references for each feature are the same.

The *arity* of a record is a list of the features of the record sorted lexicographically. To display the arity of a record we use the function `Arity`. Calling `{Arity R}` will execute as soon as `R` is bound to a record, and will return the arity of the record. Feeding the statement:

```

declare T W L R in
  T=tree(key:a left:L right:R value:1)
  W=tree(a L R 1)
  {Browse {Arity T}}
  {Browse {Arity W}}
```

will display:

```

[key left right value]
[1 2 3 4]
```

The function `{AdjoinAt R1 F X}` returns the record resulting from adjoining (i.e., adding) the field `X` to `R1` at feature `F`. The record `R1` is unchanged. If `R1` already has the feature `F`, then the result is identical to `R1` except for the field `R1.F`, whose value becomes `X`. Otherwise the feature `F` is added to `R1`. For example:

```

declare T W L R in
  T=tree(key:a left:L right:R value:1)
  W=tree(a L R 1)
```

Operation	Description
<code>{MakeTuple L N}</code>	Return tuple with label <code>L</code> and features <code>1, ..., N</code>
<code>{IsTuple T}</code>	Return boolean saying whether <code>T</code> is of tuple type

Table B.9: Some tuple operations

$\langle \text{expression} \rangle$	$::=$	$\text{'['} \{ \langle \text{expression} \rangle \}^+ \text{' '}$	$ \dots$
$\langle \text{consBinOp} \rangle$	$::=$	' '	$ \dots$

Table B.10: List syntax (*in part*)

```
{Browse {AdjoinAt T 1 b}}
{Browse {AdjoinAt W key b}}
```

will display:

```
tree(b key:a left:L right:R value:1)
tree(a L R 1 key:b)
```

The `{Adjoin R1 R2}` operation gives the same result as if `AdjoinAt` were called successively, starting with `R1` and iterating through all features of `R2`.

B.3.3 Operations on tuples

All record operations also work for tuples. There are a few additional operations that work only on tuples. Table B.9 lists some of them. The Base module `Tuple` gives them all.

B.4 Chunks (limited records)

A *chunk* is Mozart terminology for a record type with a limited set of operations. Chunks are not a fundamental concept; they can be implemented with procedure values and names, as explained in Section 3.7.5. For improved efficiency, Mozart provides chunks directly as a data type. We describe them here because some library modules use them (in particular, the module `ObjectSupport`). There are only two basic operations: create a chunk from any record and extract information with the field selection operator `“.”`:

```
declare
C={Chunk.new anyrecord(a b c)}    % Chunk creation
F=C.2                             % Chunk field selection
```

The `Label` and `Arity` operations are not defined and unification is not possible. Chunks give a way of “wrapping” information so that access to the information is restricted, i.e., not all computations can access the information. This makes chunks useful for defining secure abstract data types.

Operation	Description
{Append L1 L2}	Return the concatenation of L1 and L2
{Member X L}	Return boolean saying whether X is in L
{Length L}	Return the length of L
{List.drop L N}	Return L minus the first N elements, or nil if it is shorter
{List.last L}	Return the last element of non-empty list L
{Sort L F}	Return L sorted according to boolean comparison function F
{Map L F}	Return the list obtained by applying F to each element of L
{ForAll L P}	Apply the unary procedure P to each element of L
{Filter L F}	Return the list of elements of L for which F gives true
{FoldL L F N}	Return the value obtained by inserting F between all elements of L
{Flatten L}	Return the list of all non-list elements of L, at any nesting depth
{List.toTuple A L}	Return tuple with label A and ordered fields from L
{List.toRecord A L}	Return record with label A and features/fields F#X in L

Table B.11: Some list operations

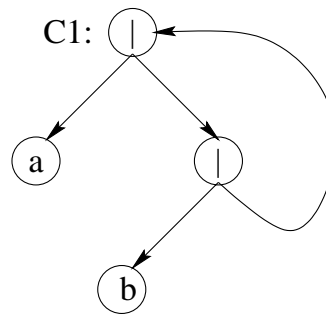
B.5.1 Operations on lists

Table B.11 gives a few basic list operations. Many more operations exist in the Base module `List`. Here is a simple symbolic calculation with lists:

```
declare A B in
A=[a b c]
B=[1 2 3 4]
{Browse {Append A B}}
```

This displays the list `[a b c 1 2 3 4]`. Like all operations, these all have correct dataflow behavior. For example, `{Length a|b|x}` blocks until `x` is bound. The operations `Sort`, `Map`, `ForAll`, `Filter`, and `FoldL` are examples of *higher-order* operations, i.e., operations that take functions or procedures as arguments. We will talk about higher-order execution in Chapter 3. For now, here's an example to give a flavor of what is possible:

```
declare L in
L=[john paul george ringo]
```

Figure B.1: Graph representation of the infinite list $C1 = a | b | C1$

$\langle \text{expression} \rangle$	$::= \langle \text{string} \rangle \dots$
$\langle \text{string} \rangle$	$::= \text{' ' } \{ \langle \text{stringChar} \rangle \langle \text{pseudoChar} \rangle \} \text{' '}$
$\langle \text{stringChar} \rangle$	$::= \text{(any inline character except " , \, and NUL)}$
$\langle \text{pseudoChar} \rangle$	$::= (\text{'\'} \text{ followed by three octal digits})$ $ (\text{'\x'} \text{ or '\X' followed by two hexadecimal digits})$ $ \text{'\a'} \text{'\b'} \text{'\f'} \text{'\n'} \text{'\r'} \text{'\t'}$ $ \text{'\v'} \text{'\\'} \text{'\''} \text{'\"'} \text{'\`'} \text{'\&'}$

Table B.12: String lexical syntax

```
{Browse {Sort L Value.<<}}
```

sorts `L` according to the comparison function `<<` and displays the result:

```
[george john paul ringo]
```

As an infix operator, comparison is written as `X<Y`, but the comparison operation itself is in the Base module `Value`. Its full name is `Value.<<`. Modules are explained in Section 3.9.

B.6 Strings

Lists whose elements are character codes are called *strings*. For example:

```
"Mozart 1.2.3"
```

is the list:

```
[77 111 122 97 114 116 32 49 46 50 46 51]
```

or equivalently:

```
[&M &o &z &a &r &t &  &1 &. &2 &. &3]
```

Using lists to represent strings is convenient because all list operations are available for doing symbolic calculations with strings. Character operations can be used together with list operations to calculate on the internals of strings. String

Operation	Description
{VirtualString.toString VS}	Return a string with the same characters as VS
{VirtualString.toAtom VS}	Return an atom with the same characters as VS
{VirtualString.length VS}	Return the number of characters in VS
{Value.toVirtualString X D W}	Return a string representing the partial value X, where records are limited in depth to D and in width to W

Table B.13: Some virtual string operations

syntax is shown in Table B.12. The NUL character mentioned in the table has character code 0 (zero). See Section B.1 for an explanation of the meaning of `^a`, `^b`, etc.

There exists another, more memory-efficient representation for character sequences called *bytestring*. This representation should only be used if memory limitations make it necessary.

B.7 Virtual strings

A *virtual string* is a tuple with label `^#` that represents a string. The virtual string brings together different substrings that are concatenated with virtual concatenation. That is, the concatenation is never actually performed, which saves time and memory. For example, the virtual string:

```
123#"-"#23#" is "#(123-23)
```

represents the string:

```
"123-23 is 100"
```

Except in special cases, a library operation that expects a string can always be given a virtual string instead. For example, virtual strings can be used for all I/O operations. The components of a virtual string can be numbers, strings, virtual strings (i.e., `^#`-labeled tuples), and all atoms except for `nil` and `^#`. Table B.13 gives a few virtual string operations.

Appendix C

Language Syntax

“The devil is in the details.”

– Traditional proverb.

“God is in the details.”

– Traditional proverb.

“I don’t know what is in those details,
but it must be something important!”

– Irreverent proverb.

This appendix defines the syntax of the complete language used in this book, including all syntactic conveniences. The language is a subset of the Oz language as implemented by the Mozart system. The appendix is divided into six sections:

- Section C.1 defines the syntax of interactive statements, i.e., statements that can be fed into the interactive interface.
- Section C.2 defines the syntax of statements and expressions.
- Section C.3 defines the syntax of the nonterminals needed to define statements and expressions.
- Section C.4 lists the operators of the language with their precedence and associativity.
- Section C.5 lists the keywords of the language.
- Section C.6 defines the lexical syntax of the language, i.e., how a character sequence is transformed into a sequence of tokens.

To be precise, this appendix defines a *context-free* syntax for a *superset* of the language. This keeps the syntax simple and easy to read. The disadvantage of a context-free syntax is that it does not capture all syntactic conditions for legal programs. For example, take the statement **local** *x* **in** *<statement>* **end**. The

$\langle \text{interStatement} \rangle$	$::= \langle \text{statement} \rangle$ $\quad \text{declare } \{ \langle \text{declarationPart} \rangle \}^+ [\langle \text{interStatement} \rangle]$ $\quad \text{declare } \{ \langle \text{declarationPart} \rangle \}^+ \text{in } \langle \text{interStatement} \rangle$
-----------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table C.1: Interactive statements

$\langle \text{statement} \rangle$	$::= \langle \text{nestCon}(\text{statement}) \rangle \mid \langle \text{nestDec}(\langle \text{variable} \rangle) \rangle$ $\quad \text{skip} \mid \langle \text{statement} \rangle \langle \text{statement} \rangle$
$\langle \text{expression} \rangle$	$::= \langle \text{nestCon}(\text{expression}) \rangle \mid \langle \text{nestDec}(\text{'\$'}) \rangle$ $\quad \langle \text{expression} \rangle \langle \text{evalBinOp} \rangle \langle \text{expression} \rangle$ $\quad \text{'$'} \mid \langle \text{term} \rangle \mid \text{'@'} \langle \text{expression} \rangle \mid \text{self}$
$\langle \text{inStatement} \rangle$	$::= [\{ \langle \text{declarationPart} \rangle \}^+ \text{in}] \langle \text{statement} \rangle$
$\langle \text{inExpression} \rangle$	$::= [\{ \langle \text{declarationPart} \rangle \}^+ \text{in}] [\langle \text{statement} \rangle] \langle \text{expression} \rangle$
$\langle \text{in}(\text{statement}) \rangle$	$::= \langle \text{inStatement} \rangle$
$\langle \text{in}(\text{expression}) \rangle$	$::= \langle \text{inExpression} \rangle$

Table C.2: Statements and expressions

statement that contains this one must declare all the free variable identifiers of $\langle \text{statement} \rangle$, possibly minus x . This is not a context-free condition.

This appendix defines the syntax of a subset of the full Oz language, as defined in [77, 47]. This appendix differs from [77] in several ways: it introduces *nestable constructs*, *nestable declarations*, and *terms* to factor the common parts of statement and expression syntax, it defines interactive statements and **for** loops, it leaves out the translation to the kernel language (which is given for each linguistic abstraction in the main text of the book), and it makes other small simplifications for clarity (but without sacrificing precision).

C.1 Interactive statements

Table C.1 gives the syntax of interactive statements. An interactive statement is a superset of a statement; in addition to all regular statements, it can contain a **declare** statement. The interactive interface must always be fed interactive statements. All free variable identifiers in the interactive statement must exist in the global environment, otherwise the system gives a “variable not introduced” error.

C.2 Statements and expressions

Table C.2 gives the syntax of statements and expressions. Many language constructs be used in either a statement position or an expression position. We

```

⟨nestCon(α)⟩ ::= ⟨expression⟩ ( '=' | ':=' | ', ' ) ⟨expression⟩
| ' { ' ⟨expression⟩ { ⟨expression⟩ } ' } '
| local { ⟨declarationPart⟩ }+ in [ ⟨statement⟩ ] ⟨α⟩ end
| ' ( ' ⟨in(α)⟩ ' ) '
| if ⟨expression⟩ then ⟨in(α)⟩
  { elseif ⟨expression⟩ then ⟨in(α)⟩ }
  [ else ⟨in(α)⟩ ] end
| case ⟨expression⟩ of ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨in(α)⟩
  { ' [ ] ' ⟨pattern⟩ [ andthen ⟨expression⟩ ] then ⟨in(α)⟩ }
  [ else ⟨in(α)⟩ ] end
| for { ⟨loopDec⟩ }+ do ⟨in(α)⟩ end
| try ⟨in(α)⟩
  [ catch ⟨pattern⟩ then ⟨in(α)⟩
    { ' [ ] ' ⟨pattern⟩ then ⟨in(α)⟩ } ]
  [ finally ⟨in(α)⟩ ] end
| raise ⟨inExpression⟩ end
| thread ⟨in(α)⟩ end
| lock [ ⟨expression⟩ then ] ⟨in(α)⟩ end

```

Table C.3: Nestable constructs (no declarations)

```

⟨nestDec(α)⟩ ::= proc ' { ' α { ⟨pattern⟩ } ' } ' ⟨inStatement⟩ end
| fun [ lazy ] ' { ' α { ⟨pattern⟩ } ' } ' ⟨inExpression⟩ end
| functor α
  [ import { ⟨variable⟩ [ at ⟨atom⟩ ]
    | ⟨variable⟩ ' ( '
      { (⟨atom⟩ | ⟨int⟩) [ ': ' ⟨variable⟩ ] }+ ' ) '
    }+ ]
  [ export { [ (⟨atom⟩ | ⟨int⟩) ': ' ] ⟨variable⟩ }+ ]
  define { ⟨declarationPart⟩ }+ [ in ⟨statement⟩ ] end
| class α { ⟨classDescriptor⟩ }
  { meth ⟨methHead⟩ [ '=' ⟨variable⟩ ]
    ( ⟨inExpression⟩ | ⟨inStatement⟩ ) end }
end

```

Table C.4: Nestable declarations

$\langle \text{term} \rangle$	$::= [\text{'!'}] \langle \text{variable} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{character} \rangle$
	$\mid \langle \text{atom} \rangle \mid \langle \text{string} \rangle \mid \text{unit} \mid \text{true} \mid \text{false}$
	$\mid \langle \text{label} \rangle \text{'('} \{ [\langle \text{feature} \rangle \text{' : ' }] \langle \text{expression} \rangle \} \text{') '}$
	$\mid \langle \text{expression} \rangle \langle \text{consBinOp} \rangle \langle \text{expression} \rangle$
	$\mid \text{'[' } \{ \langle \text{expression} \rangle \}^+ \text{'] '}$
$\langle \text{pattern} \rangle$	$::= [\text{'!'}] \langle \text{variable} \rangle \mid \langle \text{int} \rangle \mid \langle \text{float} \rangle \mid \langle \text{character} \rangle$
	$\mid \langle \text{atom} \rangle \mid \langle \text{string} \rangle \mid \text{unit} \mid \text{true} \mid \text{false}$
	$\mid \langle \text{label} \rangle \text{'('} \{ [\langle \text{feature} \rangle \text{' : ' }] \langle \text{pattern} \rangle \} [\text{'...'}] \text{') '}$
	$\mid \langle \text{pattern} \rangle \langle \text{consBinOp} \rangle \langle \text{pattern} \rangle$
	$\mid \text{'[' } \{ \langle \text{pattern} \rangle \}^+ \text{'] '}$

Table C.5: Terms and patterns

call such constructs *nestable*. We write the grammar rules to give their syntax just once, in a way that works for both statement and expression positions. Table C.3 gives the syntax for nestable constructs, not including declarations. Table C.4 gives the syntax for nestable declarations. The grammar rules for nestable constructs and declarations are *templates* with one argument. The template is instantiated each time it is used. For example, $\langle \text{nestCon}(\alpha) \rangle$ defines the template for nestable constructs without declarations. This template is used twice, as $\langle \text{nestCon}(\text{statement}) \rangle$ and $\langle \text{nestCon}(\text{expression}) \rangle$, and each corresponds to one grammar rule.

C.3 Nonterminals for statements and expressions

Tables C.5 and C.6 defines the nonterminal symbols needed for the statement and expression syntax of the preceding section. Table C.5 defines the syntax of terms and patterns. Note the close relationship between terms and patterns. Both are used to define partial values. There are just two differences: (1) patterns can contain only variable identifiers whereas terms can contain expressions, and (2) patterns can be partial (using '... ') whereas terms cannot.

Table C.6 defines nonterminals for the declaration parts of statements and loops, for binary operators (“constructing” operators $\langle \text{consBinOp} \rangle$ and “evaluating” operators $\langle \text{evalBinOp} \rangle$), for records (labels and features), and for classes (descriptors, attributes, methods, etc.).

C.4 Operators

Table C.7 gives the precedence and associativity of all the operators used in the book. All the operators are binary infix operators, except for three cases. The minus sign '- ' is a unary prefix operator. The hash symbol '# ' is an n -ary mixfix operator. The “ ' . := ' ” is a ternary infix operator that is explained in the

$\langle \text{declarationPart} \rangle$	$::= \langle \text{variable} \rangle \mid \langle \text{pattern} \rangle \text{ '=' } \langle \text{expression} \rangle \mid \langle \text{statement} \rangle$
$\langle \text{loopDec} \rangle$	$::= \langle \text{variable} \rangle \text{ in } \langle \text{expression} \rangle [\text{'.'.''} \langle \text{expression} \rangle] [\text{';''} \langle \text{expression} \rangle]$ $\mid \langle \text{variable} \rangle \text{ in } \langle \text{expression} \rangle \text{';''} \langle \text{expression} \rangle \text{';''} \langle \text{expression} \rangle$ $\mid \text{break '':'' } \langle \text{variable} \rangle \mid \text{continue '':'' } \langle \text{variable} \rangle$ $\mid \text{return '':'' } \langle \text{variable} \rangle \mid \text{default '':'' } \langle \text{expression} \rangle$ $\mid \text{collect '':'' } \langle \text{variable} \rangle$
$\langle \text{binaryOp} \rangle$	$::= \langle \text{evalBinOp} \rangle \mid \langle \text{consBinOp} \rangle$
$\langle \text{consBinOp} \rangle$	$::= \text{'\#'} \mid \text{' '} \mid \text{'.'}$
$\langle \text{evalBinOp} \rangle$	$::= \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \text{div} \mid \text{mod} \mid \text{'.'} \mid \text{andthen} \mid \text{orelse}$ $\mid \text{'::='} \mid \text{' ,' } \mid \text{'='} \mid \text{'=='} \mid \text{'\='} \mid \text{'<'} \mid \text{'<='} \mid \text{'>'} \mid \text{'>='}$
$\langle \text{label} \rangle$	$::= \text{unit} \mid \text{true} \mid \text{false} \mid \langle \text{variable} \rangle \mid \langle \text{atom} \rangle$
$\langle \text{feature} \rangle$	$::= \text{unit} \mid \text{true} \mid \text{false} \mid \langle \text{variable} \rangle \mid \langle \text{atom} \rangle \mid \langle \text{int} \rangle$
$\langle \text{classDescriptor} \rangle$	$::= \text{from } \{ \langle \text{expression} \rangle \}^+ \mid \text{prop } \{ \langle \text{expression} \rangle \}^+$ $\mid \text{attr } \{ \langle \text{attrInit} \rangle \}^+$
$\langle \text{attrInit} \rangle$	$::= ([\text{'!''}] \langle \text{variable} \rangle \mid \langle \text{atom} \rangle \mid \text{unit} \mid \text{true} \mid \text{false})$ $[\text{'::=' } \langle \text{expression} \rangle]$
$\langle \text{methHead} \rangle$	$::= ([\text{'!''}] \langle \text{variable} \rangle \mid \langle \text{atom} \rangle \mid \text{unit} \mid \text{true} \mid \text{false})$ $[\text{'(' } \{ \langle \text{methArg} \rangle \} [\text{'...'}] \text{')' }]$ $[\text{'=' } \langle \text{variable} \rangle]$
$\langle \text{methArg} \rangle$	$::= [\langle \text{feature} \rangle \text{'::='}] (\langle \text{variable} \rangle \mid \text{'_'} \mid \text{'\$'}) [\text{'<=' } \langle \text{expression} \rangle]$

Table C.6: Other nonterminals needed for statements and expressions

next section. There are no postfix operators. The operators are listed in order of increasing precedence, i.e., tightness of binding. The operators lower in the table bind tighter. We define the associativities as follows:

- Left. For binary operators, this means that repeated operators group to the left. For example, $1+2+3$ means the same as $((1+2)+3)$.
- Right. For binary operators, this means that repeated operators group to the right. For example, $a|b|x$ means the same as $(a|(b|x))$.
- Mixfix. Repeated operators are actually just one operator, with all expressions being arguments of the operator. For example, $a\#b\#c$ means the same as $\text{'\#'}(a \ b \ c)$.
- None. For binary operators, this means that the operator cannot be repeated. For example, $1<2<3$ is an error.

Parentheses can be used to override the default precedence.

andthen	default	false	lock	return
at	define	feat (*)	meth	self
attr	dis (*)	finally	mod	skip
break	div	for	not (*)	then
case	do	from	of	thread
catch	else	fun	or (*)	true
choice	elsecase (*)	functor	orelse	try
class	elseif	if	prepare (*)	unit
collect	elseof (*)	import	proc	
cond (*)	end	in	prop	
continue	export	lazy	raise	
declare	fail	local	require (*)	

Table C.8: Keywords

C.4.1 Ternary operator

There is one ternary (three-argument) operator, “**.** :=”, which is designed for dictionary and array updates. It has the same precedence and associativity as **:=**. It can be used in an expression position like **:=**, where it has the effect of an exchange. The statement $S.I := X$ consists of a ternary operator with arguments S , I , and X . This statement is used for updating dictionaries and arrays. This should not be confused with $(S.I) := X$, which consists of the two nested binary operators **.** and **:=**. The latter statement is used for updating a cell that is inside a dictionary. The parentheses are highly significant! Figure C.1 shows the difference in abstract syntax between $S.I := X$ and $(S.I) := X$. In the figure, *(cell)* means any cell or object attribute, and *(dictionary)* means any dictionary or array.

The distinction is important because dictionaries can contain cells. To update a dictionary D , we write $D.I := X$. To update a cell in a dictionary containing cells, we write $(D.I) := X$. This has the same effect as **local** $C = D.I$ **in** $C := X$ **end** but is more concise. The first argument of the binary operator **:=** must be a cell or object attribute.

C.5 Keywords

Table C.8 lists the keywords of the language in alphabetic order. Keywords marked with (*) exist in Oz but are not used in this book. Keywords in boldface can be used as atoms by enclosing them in quotes. For example, ‘**then**’ is an atom whereas **then** is a keyword. Keywords not in boldface can be used as atoms directly, without quotes.

$\langle \text{variable} \rangle$	$::= (\text{uppercase char}) \{ (\text{alphanumeric char}) \}$ $ ' ' ' \{ \langle \text{variableChar} \rangle \mid \langle \text{pseudoChar} \rangle \} ' ' '$
$\langle \text{atom} \rangle$	$::= (\text{lowercase char}) \{ (\text{alphanumeric char}) \}$ (except no keyword) $ ' ' ' \{ \langle \text{atomChar} \rangle \mid \langle \text{pseudoChar} \rangle \} ' ' '$
$\langle \text{string} \rangle$	$::= ' " ' \{ \langle \text{stringChar} \rangle \mid \langle \text{pseudoChar} \rangle \} ' " '$
$\langle \text{character} \rangle$	$::= (\text{any integer in the range } 0 \dots 255)$ $ '\&' \langle \text{charChar} \rangle \mid '\&' \langle \text{pseudoChar} \rangle$

Table C.9: Lexical syntax of variables, atoms, strings, and characters

$\langle \text{variableChar} \rangle$	$::= (\text{any inline character except } \backslash, \backslash, \text{ and NUL})$
$\langle \text{atomChar} \rangle$	$::= (\text{any inline character except } ', \backslash, \text{ and NUL})$
$\langle \text{stringChar} \rangle$	$::= (\text{any inline character except } ", \backslash, \text{ and NUL})$
$\langle \text{charChar} \rangle$	$::= (\text{any inline character except } \backslash \text{ and NUL})$
$\langle \text{pseudoChar} \rangle$	$::= '\backslash' \langle \text{octdigit} \rangle \langle \text{octdigit} \rangle \langle \text{octdigit} \rangle$ $ (' \backslash x' \mid '\backslash X') \langle \text{hexdigit} \rangle \langle \text{hexdigit} \rangle$ $ '\backslash a' \mid '\backslash b' \mid '\backslash f' \mid '\backslash n' \mid '\backslash r' \mid '\backslash t'$ $ '\backslash v' \mid '\backslash \backslash' \mid '\backslash '' \mid '\backslash "' \mid '\backslash '' \mid '\backslash \&'$

Table C.10: Nonterminals needed for lexical syntax

$\langle \text{int} \rangle$	$::= ['\sim'] \langle \text{nzdigit} \rangle \{ \langle \text{digit} \rangle \}$ $ ['\sim'] 0 \{ \langle \text{octdigit} \rangle \}^+$ $ ['\sim'] ('0x' \mid '0X') \{ \langle \text{hexdigit} \rangle \}^+$ $ ['\sim'] ('0b' \mid '0B') \{ \langle \text{bindigit} \rangle \}^+$
$\langle \text{float} \rangle$	$::= ['\sim'] \{ \langle \text{digit} \rangle \}^+ '\.' \{ \langle \text{digit} \rangle \} [('e' \mid 'E') ['\sim'] \{ \langle \text{digit} \rangle \}^+]$
$\langle \text{digit} \rangle$	$::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$\langle \text{nzdigit} \rangle$	$::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
$\langle \text{octdigit} \rangle$	$::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7$
$\langle \text{hexdigit} \rangle$	$::= \langle \text{digit} \rangle \mid 'a' \mid 'b' \mid 'c' \mid 'd' \mid 'e' \mid 'f'$ $ 'A' \mid 'B' \mid 'C' \mid 'D' \mid 'E' \mid 'F'$
$\langle \text{bindigit} \rangle$	$::= 0 \mid 1$

Table C.11: Lexical syntax of integers and floating point numbers

C.6 Lexical syntax

This section defines the lexical syntax of Oz, i.e., how a character sequence is transformed into a sequence of tokens.

C.6.1 Tokens

Variables, atoms, strings, and characters

Table C.9 defines the lexical syntax for variable identifiers, atoms, strings, and characters in strings. An *alphanumeric* character is a letter (uppercase or lowercase), a digit, or an underscore character. Unlike the previous sections which define token sequences, this section defines character sequences. It follows from this syntax that an atom cannot have the same character sequence as a keyword unless the atom is quoted. Table C.10 defines the nonterminals needed for Table C.9. “Any inline character” includes control characters and accented characters. The NUL character has character code 0 (zero).

Integers and floating point numbers

Table C.11 defines the lexical syntax of integers and floating point numbers. Note the use of the `~` (tilde) for the unary minus symbol.

C.6.2 Blank space and comments

Tokens may be separated by any amount of blank space and comments. *Blank space* is one of the characters tab (character code 9), newline (code 10), vertical tab (code 11), form feed (code 12), carriage return (code 13), and space (code 32). A *comment* is one of three possibilities:

- A sequence of characters starting from the character `%` (percent) until the end of the line or the end of the file (whichever comes first).
- A sequence of characters starting from `/*` and ending with `*/`, inclusive. This kind of comment may be nested.
- The single character `?` (question mark). This is intended to mark the output arguments of procedures, as in:

```
proc {Max A B ?C} ... end
```

where `C` is an output. An *output argument* is an argument that gets bound inside the procedure.

Appendix D

General Computation Model

“The removal of much of the accidental complexity of programming means that the intrinsic complexity of the application is what’s left.”

– Security Engineering, *Ross J. Anderson* (2001)

“If you want people to do something the right way, you must make the right way the easy way.”

– Traditional saying.

This appendix brings together all the general concepts introduced in the book.¹ The resulting computation model is the shared-state concurrent model of Chapter 8. For convenience we call it the *general computation model*. While this model is quite general, it is certainly not the final word in computation models. It is just a snapshot that captures our current understanding of programming. Future research will certainly change or extend it. The book mentions dynamic scoping and transaction support as two areas which require more support from the model.

The general computation model was designed in a layered fashion, by starting from a simple base model and successively adding new concepts. Each time we noted a limitation in the expressiveness of a computation model, we had the opportunity to add a new concept. There was always a choice: either to keep the model as is and make programs more complicated, or to add a concept and keep programs simple. The decision to add the concept or not was based on our judgement of how complicated the model and its programs would be, when considered together. “Complexity” in this sense covers both the expressiveness and ease of reasoning of the combination.

There is a strong element of creativity in this approach. Each concept brings something novel that was not there before. We therefore call it the *creative extension principle*. Not all useful concepts end up in the general model. Some concepts were added only to be superseded by later concepts. For example, this is the case for nondeterministic choice (Section 5.7.1), which is superseded

¹Except for computation spaces, which underlie the relational computation model and the constraint-based computation model.

by explicit state. The general model is just one among many possible models of similar expressiveness. Your judgement in this process may be different from ours. We would be interested to hear from any reader who has reached significantly different conclusions.

Because earlier computation models are subsets of later ones, the later ones can be considered as *frameworks* inside of which many computation models can coexist. In this sense, the general computation model is the most complete framework of the book.

D.1 Creative extension principle

We give an example to explain and motivate the creative extension principle. Let us start with the simple declarative language of Chapter 2. In that chapter, we added two concepts to the declarative language: functions and exceptions. But there was something fundamentally different in how we added each concept. Functions were added as a linguistic abstraction by defining a new syntax and showing how to translate it into the kernel language (see Section 2.5.2). Exceptions were added to the kernel language itself by adding new primitive operations and defining their semantics (see Section 2.6). Why did we choose to do it this way? We could have added functions to the kernel language and defined exceptions by translation, but we did not. There is a simple but profound reason for this: functions can be defined by a local translation but exceptions cannot. A translation of a concept is *local* if it requires changes only to the parts of the program that use the concept.

Starting with the declarative kernel language of Chapter 2, this book added concepts one by one. For each concept we had to decide whether to add it as a linguistic abstraction (without changing the kernel language) or to add it to the kernel language. A linguistic abstraction is a good idea if the translation is local. Extending the kernel language is a good idea if there is no local translation.

This choice is always a trade-off. One criterium is that the overall scheme, including both the kernel language and the translation scheme into the kernel language, should be as simple as possible. This is what we call the *creative extension principle*. To some degree, simplicity is a subjective judgement. This book makes one particular choice of what should be in the kernel languages and what should be outside. Other reasonable choices are certainly possible.

An additional constraint on the kernel languages of this book is that they are all carefully chosen to be subsets of the full Oz language. This means that they are all implemented by the Mozart system. Users can verify that the kernel language translation of a program behaves in exactly the same way as the program. The only difference between the two is efficiency. This is useful both for learning the kernel languages and for debugging programs. The Mozart system implements certain constructs more efficiently than their representation in the kernel language. For example, classes and objects in Oz are implemented more

$\langle s \rangle ::=$	
skip	Empty statement
$\langle s \rangle_1 \langle s \rangle_2$	Statement sequence
local $\langle x \rangle$ in $\langle s \rangle$ end	Variable creation
$\langle x \rangle_1 = \langle x \rangle_2$	Variable-variable binding
$\langle x \rangle = \langle v \rangle$	Value creation
$\{ \langle x \rangle \langle y \rangle_1 \dots \langle y \rangle_n \}$	Procedure application
if $\langle x \rangle$ then $\langle s \rangle_1$ else $\langle s \rangle_2$ end	Conditional
case $\langle x \rangle$ of $\langle \text{pattern} \rangle$ then $\langle s \rangle_1$ else $\langle s \rangle_2$ end	Pattern matching
thread $\langle s \rangle$ end	Thread creation
$\{ \text{ByNeed } \langle x \rangle \langle y \rangle \}$	Trigger creation
$\{ \text{NewName } \langle x \rangle \}$	Name creation
$\langle y \rangle = ! ! \langle x \rangle$	Read-only view
try $\langle s \rangle_1$ catch $\langle x \rangle$ then $\langle s \rangle_2$ end	Exception context
raise $\langle x \rangle$ end	Raise exception
$\{ \text{FailedValue } \langle x \rangle \langle y \rangle \}$	Failed value
$\{ \text{NewCell } \langle x \rangle \langle y \rangle \}$	Cell creation
$\{ \text{Exchange } \langle x \rangle \langle y \rangle \langle z \rangle \}$	Cell exchange
$\{ \text{IsDet } \langle x \rangle \langle y \rangle \}$	Boundness test

Table D.1: The general kernel language

efficiently than their kernel definitions.

D.2 Kernel language

Table D.1 gives the kernel language of the general computation model. For clarity, we divide the table into five parts:

- The first part is the *descriptive declarative model*. This model allows to build complex data structures (rooted graphs whose nodes are records and procedure values) but does not allow to calculate with them.
- The first and second parts taken together form the *declarative concurrent model*. This is the most general purely declarative model of the book. All programs written in this model are declarative.
- The third part adds security: the ability to build secure ADTs and program with capabilities.
- The fourth part adds exceptions: the ability to handle exceptional situations by doing non-local exits.
- The fifth part adds explicit state, which is important for building modular programs and programs that can change over time.

Taking all parts gives the *shared-state concurrent model*. This is the most general model of the book. Chapter 13 gives the semantics of this model and all its subsets.

D.3 Concepts

Let us now recapitulate the design methodology of the general model by starting with a simple base model and briefly explaining what new expressiveness each concept brings. All models are *Turing complete*, that is, they are equivalent in computing power to a Turing machine. However, Turing completeness is only a small part of the story. The ease in which programs can be written or reasoned about differs greatly in these models. Increased expressiveness typically goes hand in hand with increased difficulty to reason about programs.

D.3.1 Declarative models

Strict functional model The simplest practical model is strict functional programming with values. This model is defined in Section 2.7.1. In this model there are no unbound variables; each new variable is immediately bound to a value. This model is close to the λ calculus, which contains just procedure definition and application and leaves out the conditional and pattern matching. The λ calculus is Turing complete but is much too cumbersome for practical programming.

Sequential declarative model The sequential declarative model is defined in Chapter 2. It contains all concepts in Table D.1 up to and including procedure application, conditionals, and pattern matching. It extends the strict functional model by introducing dataflow variables. Doing this is a critical step because it prepares the way for declarative concurrency. For binding dataflow variables, we use a general operation called *unification*. This means that the sequential declarative model does both deterministic logic programming and functional programming.

Threads The thread concept is defined in Section 4.1. Adding threads allows the model to express activities that execute independently. This model is still declarative: adding threads leaves the result of a calculation unchanged. Only the order in which the calculations are done is more flexible. Programs become more incremental: incrementally building an input results in an incrementally-built output. This is one form of declarative concurrency.

Triggers The trigger concept is defined in Section 4.5.1. Adding triggers allows the model to express demand-driven computations (laziness). This model is still declarative since the result of a calculation is unchanged. Only the amount of calculation done to achieve the result changes (it can become smaller). Sometimes

the demand-driven model can give results in cases where the data-driven model would go into an infinite loop. This is a second form of declarative concurrency.

D.3.2 Security

Names The name concept is defined in Section 3.7.5. A name is an unforgeable constant that does not exist outside of a program. A name has no data or operations attached to it; it is a first-class “key” or “right”. Names are the basis of programming techniques such as unbundled secure ADTs (see Section 6.4) and encapsulation control (see Section 7.3.3).

In the declarative model, secure ADTs can be built without names by using procedure values to hide values. The hidden value is an external reference of the procedure. But names add a crucial additional expressiveness. They make it possible to program with rights. For example, separating data from operations in a secure ADT or passing keys to programs to enable secure operations.

Strictly speaking, names are not declarative since successive calls to `NewName` give different results. That is, the same program can return two different results if names are used to identify the result uniquely. But if names are used only to enforce security properties, then the model is still declarative.

Read-only views The read-only view concept is defined in Section 3.7.5. A read-only view is a dataflow variable that can be read but not bound. It is always paired with another dataflow variable that is equal to it but that can be bound. Read-only views are needed to construct secure ADTs that export unbound variables. The ADT exports the read-only view. Since it cannot be bound outside the ADT, this allows the ADT to maintain its invariant property.

D.3.3 Exceptions

Exception handling The exception concept is defined in Section 2.6.2. Adding exceptions allows to exit in one step from an arbitrarily large number of nested procedure calls. This allows to write programs that treat rare cases correctly, without complicating the program in the common case.

Failed values The failed value concept is defined in Section 4.9.1. A failed value is a special kind of value that encapsulates an exception. Any attempt to use the value or to bind it to a determined value will raise the exception. While exception handling happens within a single thread, failed values allow to pass exceptions from the thread that detected the problem to other threads.

Failed values are useful in models that have both exceptions and triggers. Assume that a program calculates a value by a demand-driven computation, but the computation raises an exception instead. What should the value be? It can be a failed value. This will cause any thread that needs the value to raise an exception.

D.3.4 Explicit state

Cells (explicit state) Explicit state is defined in Section 6.3. Adding state gives a program a memory: a procedure can change its behavior over successive calls. In the declarative model this is not possible since all knowledge is in the procedure's arguments.

The stateful model greatly improves program modularity when compared to models without state. It increases the possibilities for changing a module's implementation without changing its interface (see Section 4.7).

Ports (explicit state) Another way to add explicit state is by means of *ports*, which are a kind of asynchronous communication channel. As explained in Section 7.8, ports and cells are equivalent: each can implement the other in a simple way. Ports are useful for programming message passing with active objects. Cells are useful for programming atomic actions with shared state.

Boundness test (weak state) The boundness test `IsDet` lets us use dataflow variables as a weak form of explicit state. The test checks whether a variable is bound or still unbound, without waiting when the variable is unbound. For many programming techniques, knowing the binding status of a dataflow variable is unimportant. However, it can be important when programming a time-dependent execution, i.e., to know what the instantaneous state is of an execution (see Section 4.7.3).

Object-oriented programming Object-oriented programming is introduced in Chapter 7. It has the same kernel language as the stateful models. It is a rich set of programming techniques that uses ideas from knowledge representation to improve program structure. The two main ideas are to consider programs as collections of interacting ADTs (which can be grouped together in associations) and to allow building ADTs incrementally (using inheritance, delegation, and forwarding).

D.4 Different forms of state

Adding explicit state is such a strong change to the model that it is important to have weaker forms of state. In the above models we have introduced four forms of state. Let us summarize these forms in terms of how many times we can assign a variable, i.e., change its state. In order of increasing strength, they are:

- No assignment, i.e., programming with values only (monotonic execution). This is functional programming as it is usually understood. Programs are completely deterministic, i.e., the same program always gives the same result.

- Single assignment, i.e., programming with dataflow variables (monotonic execution). This is also functional programming, but more flexible since it allows declarative concurrency (with both lazy and eager execution). Programs are completely deterministic, but the result can be given incrementally.
- Single assignment with boundness test, i.e., programming with dataflow variables and `IsDet` (nonmonotonic execution). Programs are no longer deterministic.
- Multiple assignment, i.e., programming with cells or ports (nonmonotonic execution). Programs are no longer deterministic. This is the most expressive model.

We can understand these different forms of state in terms of an important property called *monotonicity*. At any time, a variable can be assigned to an element of some set S of values. Assignment is monotonic if as execution progresses, values can be removed from S but not added. For example, binding a dataflow variable x to a value reduces S from all possible values to just one value. A function f is monotonic if $S_1 \subset S_2 \implies f(S_1) \subset f(S_2)$. For example, `IsDet` is nonmonotonic since $\{\text{IsDet } x\}$ returns **false** when x is unbound and **true** when x is bound, and $\{\text{true}\}$ is not a subset of $\{\text{false}\}$. A program's execution is monotonic if all its operations are monotonic. Monotonicity is what makes declarative concurrency possible.

D.5 Other concepts

D.5.1 What's next?

The general computation model of this book is just a snapshot of an ongoing process. New concepts will continue to be discovered in the future using the creative extension principle. What will these new concepts be? We cannot tell for sure, since anticipating a discovery is the same as making that discovery! But there are hints about a few of the concepts. Two concepts that we are fairly sure about, even though we do not know their final form, are dynamic scoping and transaction support. With dynamic scoping the behavior of a component depends on its context. With transaction support the execution of a component can be canceled if it cannot complete successfully. According to the creative extension principle, both of these concepts should be added to the computation model.

D.5.2 Domain-specific concepts

This book gives many general concepts that are useful for all kinds of programs. In addition to this, each application domain has its own set of concepts that are useful only in that domain. These extra concepts complement the general

concepts. For example, we can cite artificial intelligence [160, 136], algorithm design [41], object-oriented design patterns [58], multi-agent programming [205], databases [42], and numerical analysis [153].

D.6 Layered language design

The general computation model has a layered design. Each layer offers its own special trade-off of expressiveness and ease of reasoning. The programmer can choose the layer that is best-adapted to each part of the program. From the evidence presented in the book, it is clear that this layered structure is beneficial for a general-purpose programming language. It makes it easier for the programmer to say directly what he or she wants to say, without cumbersome encodings.

The layered design of the general computation model can be found to some degree in many languages. Object-oriented languages such as Smalltalk, Eiffel, and Java have two layers: an object-oriented core and a second layer providing shared-state concurrency [60, 122, 10]. The functional language Erlang has two layers: an eager functional core and a second layer providing message-passing concurrency between active objects [9] (see also Section 5.6). Active objects are defined within the functional core. The logic language Prolog has three layers: a logical core that is a simple theorem prover, a second layer modifying the theorem prover's operation, and a third layer providing explicit state [182] (see also Section 9.7). The functional language Concurrent ML has three layers: an eager functional core, a second layer providing explicit state, and a third layer providing concurrency [158]. The multiparadigm language Oz has many layers, which is why it was used as the basis for this book [180].

Bibliography

- [1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, Mass, 1985.
- [2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. The MIT Press, Cambridge, Mass, 1996.
- [3] Iliès Alouini and Peter Van Roy. Le protocole réparti du langage Distributed Oz (The distributed protocol of the Distributed Oz language). In *Colloque Francophone d'Ingénierie de Protocoles (CFIP 99)*, pages 283–298, April 1999.
- [4] Edward G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall, 1994.
- [5] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.
- [6] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.
- [7] Joe Armstrong. Higher-order processes in Erlang, January 1997. Unpublished talk.
- [8] Joe Armstrong. Concurrency oriented programming in Erlang, November 2002. Invited talk, Lightweight Languages Workshop 2002.
- [9] Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.
- [10] Ken Arnold and James Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1998.
- [11] Arvind and R. E. Thomas. I-Structures: An efficient data type for functional languages. Technical Report 210, MIT, Laboratory for Computer Science, 1980.

- [12] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.
- [13] John Backus. The history of FORTRAN I, II and III. *ACM SIGPLAN Notices*, 13(8), August 1978.
- [14] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.
- [15] Holger Bär, Markus Bauer, Oliver Ciupke, Serge Demeyer, Stéphane Ducasse, Michele Lanza, Radu Marinescu, Robb Nebbe, Oscar Nierstrasz, Michael Przybiski, Tamar Richner, Matthias Rieger, Claudio Riva, Anne-Marie Sassen, Benedikt Schulz, Patrick Steyaert, Sander Tichelaar, and Joachim Weisbrod. *The FAMOOS Object-Oriented Reengineering Handbook*. October 1999. Result of ESPRIT project FAMOOS.
- [16] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery In Database Systems*. Addison-Wesley, 1987.
- [17] Richard Bird. *Introduction to Functional Programming using Haskell, Second Edition*. Prentice Hall, 1998.
- [18] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [19] Darius Blasband. Language engineering: from a hobby, to a research activity, to a trade, March 2002. Unpublished talk.
- [20] Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klinskog. Path redundancy in a mobile-state protocol as a primitive for language-based fault tolerance. Technical Report RR2000-01, Département d'Ingénierie Informatique, Université catholique de Louvain, 2000. Available at <http://www.info.ucl.ac.be>.
- [21] Ivan Bratko. *PROLOG Programming for Artificial Intelligence, Third Edition*. Addison-Wesley, 2000.
- [22] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.
- [23] Per Brinch Hansen. *Operating System Principles*. Prentice Hall, 1973.
- [24] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.
- [25] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

- [26] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, 1995.
- [27] Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.
- [28] Luca Cardelli. A language with distributed scope. In *Principles of Programming Languages (POPL)*, pages 286–297, 1995.
- [29] Mats Carlsson *et al.* SICStus Prolog 3.8.1, December 1999. Available at <http://www.sics.se>.
- [30] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
- [31] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.
- [32] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, France, 2000.
- [33] Randy Chow and Theodore Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley, San Francisco, Calif., 1997.
- [34] Keith L. Clark. PARLOG: the language and its applications. In A. J. Nijman J. W. de Bakker and P. C. Treleaven, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 30–53, Eindhoven, The Netherlands, June 1987. Springer.
- [35] Keith L. Clark and Frank McCabe. The control facilities of IC-Prolog. In D. Michie, editor, *Expert Systems in the Micro-Electronic Age*, pages 122–149. Edinburgh University Press, Edinburgh, Scotland, 1979.
- [36] Keith L. Clark, Frank G. McCabe, and Steve Gregory. IC-PROLOG — language features. In Keith L. Clark and Sten-Åke Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.
- [37] Arthur C. Clarke. *Profiles of the Future*. Pan Books, 1973. Revised edition.
- [38] William Clinger and Jonathan Rees. The revised⁴ report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.
- [39] Helder Coelho and José C. Cotta. *Prolog by Example: How to Learn, Teach, and Use It*. Springer-Verlag, 1988.
- [40] Alain Colmerauer. The birth of Prolog. In *The Second ACM-SIGPLAN History of Programming Languages Conference*, pages 37–52, March 1993. ACM SIGPLAN Notices.

- [41] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, McGraw-Hill, 1990.
- [42] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 1994.
- [43] Harvey M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, 1984.
- [44] Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz, and Ralph E. Johnson. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [45] Edsger W. Dijkstra. *A Primer of Algol 60 Programming*. Academic Press, 1962.
- [46] Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
- [47] Denys Duchier. Loop support. Technical report, DFKI and Saarland University, December 2001. Available at <http://www.mozart-oz.org/>.
- [48] Denys Duchier, Claire Gardent, and Joachim Niehren. Concurrent constraint programming in Oz for natural language processing. Technical report, Saarland University, Saarbrücken, Germany, 1999. Available at <http://www.ps.uni-sb.de/Papers/abstracts/oznlp.html>.
- [49] Denys Duchier, Leif Kornstaedt, and Christian Schulte. The Oz base environment. Technical report, Mozart Consortium, December 2001. Available at <http://www.mozart-oz.org/>.
- [50] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A Higher-order Module Discipline with Separate Compilation, Dynamic Linking, and Pickling. Technical report, Programming Systems Lab, DFKI and Saarland University, 1998. DRAFT. Available at <http://www.mozart-oz.org/papers/>.
- [51] R. Kent Dybvig, Carl Bruggeman, and David Eby. Guardians in a generation-based garbage collector, June 1993.
- [52] E. W. Elcock. Absys: The first logic programming language—a retrospective and a commentary. *Journal of Logic Programming*, 9(1):1–17, 1990.
- [53] Robert W. Floyd. Nondeterministic algorithms. *Journal of the ACM*, 14(4):636–644, October 1967.
- [54] Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman, Inc., 2000.
- [55] Michael J. French. *Invention and evolution: design in nature and engineering*. Cambridge University Press, 1988.

- [56] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press, 1992.
- [57] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. KLIC: A portable implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94)*, pages 66–79, December 1994.
- [58] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [59] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
- [60] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1983.
- [61] Danny Goodman. *Dynamic HTML: The Definitive Reference, Second Edition*. O'Reilly & Associates, 2002.
- [62] James Edward Gordon. *The Science of Structures and Materials*. Scientific American Library, 1988.
- [63] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at <http://www.javasoft.com>.
- [64] Jim Gray and Andreas Reuter. *Transaction Processing – Concepts and Techniques*. Morgan Kaufmann, 1993.
- [65] Donatien Grolaux. QtK module, 2000. Available at <http://www.mozart-oz.org/mozart-stdlib/index.html>.
- [66] Donatien Grolaux, Peter Van Roy, and Jean Vanderdonckt. QtK – a mixed declarative/procedural approach for designing executable user interfaces. In *8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01)*, Lecture Notes in Computer Science, Toronto, Canada, May 2001. Springer-Verlag. Short paper.
- [67] Donatien Grolaux, Peter Van Roy, and Jean Vanderdonckt. QtK – an integrated model-based approach to designing executable user interfaces. In *8th Workshop on Design, Specification, and Verification of Interactive Systems (DSVIS 2001)*, Lecture Notes in Computer Science, Glasgow, Scotland, June 2001. Springer-Verlag.
- [68] Robert H. Halstead, Jr. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.