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 the 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 statement that contains this one must declare all the free variable identifiers of ⟨statement⟩, 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 [55, 87]. This appendix differs from [87] 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

⟨interStatement⟩ ::= ⟨statement⟩ | **declare** { ⟨declarationPart⟩ }+ [⟨interStatement⟩] | **declare** { ⟨declarationPart⟩ }+ **in** ⟨interStatement⟩

Table C.1: Interactive statements.

⟨statement⟩	::= ⟨nestCon(statement)⟩ | ⟨nestDec(⟨variable⟩)⟩ | **skip** | ⟨statement⟩ ⟨statement⟩
⟨expression⟩	::= ⟨nestCon(expression)⟩ | ⟨nestDec(´$´)⟩ \| ⟨unaryOp⟩ ⟨expression⟩ \| ⟨expression⟩ ⟨evalBinOp⟩ ⟨expression⟩ \| ´$´ | ⟨term⟩ | **self**
⟨inStatement⟩	::= [{ ⟨declarationPart⟩ }+ **in**] ⟨statement⟩
⟨inExpression⟩	::= [{ ⟨declarationPart⟩ }+ **in**] [⟨statement⟩] ⟨expression⟩
⟨in(statement)⟩	::= ⟨inStatement⟩
⟨in(expression)⟩	::= ⟨inExpression⟩

Table C.2: Statements and expressions.

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 can be used in either a statement position or an expression position. We call such constructs nestable. We write the grammar rules to give their syntax just once,

⟨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** ⟨inStatement⟩] **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.

⟨term⟩	::= [´!´] ⟨variable⟩ | ⟨int⟩ | ⟨float⟩ | ⟨character⟩
	| ⟨atom⟩ | ⟨string⟩ | **unit** | **true** | **false**
	| ⟨label⟩ ´(´ { [⟨feature⟩ ´:´] ⟨expression⟩ } ´)´
	| ⟨expression⟩ ⟨consBinOp⟩ ⟨expression⟩
	| ´[´ { ⟨expression⟩ }+ ´]´
⟨pattern⟩	::= [´!´] ⟨variable⟩ | ⟨int⟩ | ⟨float⟩ | ⟨character⟩
	| ⟨atom⟩ | ⟨string⟩ | **unit** | **true** | **false**
	| ⟨label⟩ ´(´ { [⟨feature⟩ ´:´] ⟨pattern⟩ } [´...´] ´)´
	| ⟨pattern⟩ ⟨consBinOp⟩ ⟨pattern⟩
	| ´[´ { ⟨pattern⟩ }+ ´]´

Table C.5: Terms and patterns.

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, ⟨nestCon(α)⟩ defines the template for nestable constructs without declarations. This template is used twice, as ⟨nestCon(statement)⟩ and ⟨nestCon(expression)⟩, and each corresponds to one grammar rule.

C.3 Nonterminals for statements and expressions

Tables C.5 and C.6 define 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 unary operators, for binary operators ("constructing" operators ⟨consBinOp⟩ and "evaluating" operators ⟨evalBinOp⟩), 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

```
⟨declarationPart⟩ ::= ⟨variable⟩ | ⟨pattern⟩ ´=´ ⟨expression⟩ | ⟨statement⟩
⟨loopDec⟩         ::= ⟨variable⟩ in ⟨expression⟩ [ ´..´ ⟨expression⟩ ] [ ´;´ ⟨expression⟩ ]
                    | ⟨variable⟩ in ⟨expression⟩ ´;´ ⟨expression⟩ ´;´ ⟨expression⟩
                    | break ´:´ ⟨variable⟩ | continue ´:´ ⟨variable⟩
                    | return ´:´ ⟨variable⟩ | default ´:´ ⟨expression⟩
                    | collect ´:´ ⟨variable⟩
⟨unaryOp⟩         ::= ´~´ | ´@´ | ´!!´
⟨binaryOp⟩        ::= ⟨consBinOp⟩ | ⟨evalBinOp⟩
⟨consBinOp⟩       ::= ´#´ | ´|´
⟨evalBinOp⟩       ::= ´+´ | ´-´ | ´*´ | ´/´ | div | mod | ´.´ | andthen | orelse
                    | ´:=´ | ´,´ | ´=´ | ´==´ | ´\=´ | ´<´ | ´=<´ | ´>´ | ´>=´
                    | ´::´ | ´=:´ | ´\=:´ | ´=<:´
⟨label⟩           ::= unit | true | false | ⟨variable⟩ | ⟨atom⟩
⟨feature⟩         ::= unit | true | false | ⟨variable⟩ | ⟨atom⟩ | ⟨int⟩
⟨classDescriptor⟩ ::= from { ⟨expression⟩ }+ | prop { ⟨expression⟩ }+
                    | attr { ⟨attrInit⟩ }+
⟨attrInit⟩        ::= ( [ ´!´ ] ⟨variable⟩ | ⟨atom⟩ | unit | true | false )
                      [ ´:´ ⟨expression⟩ ]
⟨methHead⟩        ::= ( [ ´!´ ] ⟨variable⟩ | ⟨atom⟩ | unit | true | false )
                      [ ´(´ { ⟨methArg⟩ } [ ´...´ ] ´)´ ]
                      [ ´=´ ⟨variable⟩ ]
⟨methArg⟩         ::= [ ⟨feature⟩ ´:´ ] ( ⟨variable⟩ | ´_´ | ´$´ ) [ ´<=´ ⟨expression⟩ ]
```

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

mixfix operator. The "`. :=`" is a ternary infix operator that is explained in the 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 `´#´(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.

Operator	Associativity
`=`	right
`:=` "`. :=`"	right
`orelse`	right
`andthen`	right
`== \= < =< > >= =: \=: =<:`	none
`::`	none
`|`	right
`#`	mixfix
`+ -`	left
`* / div mod`	left
`,`	right
`~`	left
`.`	left
`@ !!`	left

Table C.7: Operators with their precedence and associativity.

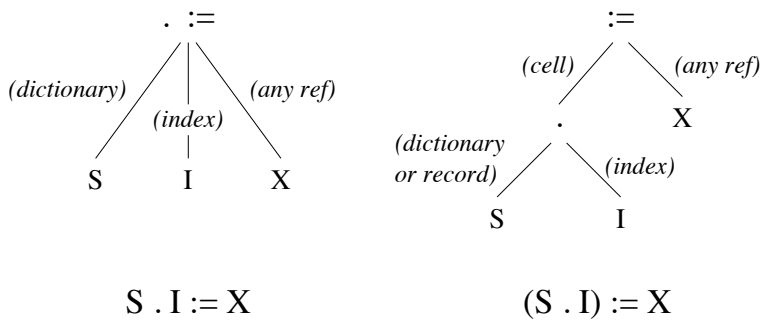

Figure C.1: The ternary operator "`. :=`".

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

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

Table C.8: Keywords.

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 an 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 the 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.

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.

⟨variable⟩ ::= (uppercase char) { (alphanumeric char) }
| ´`´ { ⟨variableChar⟩ | ⟨pseudoChar⟩ } ´`´
⟨atom⟩ ::= (lowercase char) { (alphanumeric char) } (except no keyword)
| ´´´ { ⟨atomChar⟩ | ⟨pseudoChar⟩ } ´´´
⟨string⟩ ::= ´"´ { ⟨stringChar⟩ | ⟨pseudoChar⟩ } ´"´
⟨character⟩ ::= (any integer in the range 0...255)
| ´&´ ⟨charChar⟩ | ´&´ ⟨pseudoChar⟩

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

⟨variableChar⟩ ::= (any inline character except `, \, and NUL)
⟨atomChar⟩ ::= (any inline character except ´, \, and NUL)
⟨stringChar⟩ ::= (any inline character except ", \, and NUL)
⟨charChar⟩ ::= (any inline character except \ and NUL)
⟨pseudoChar⟩ ::= ´\´ ⟨octdigit⟩ ⟨octdigit⟩ ⟨octdigit⟩
| (´\x´ | ´\X´) ⟨hexdigit⟩ ⟨hexdigit⟩
| ´\a´ | ´\b´ | ´\f´ | ´\n´ | ´\r´ | ´\t´
| ´\v´ | ´\\´ | ´\´´ | ´\"´ | ´\`´ | ´\&´

Table C.10: Nonterminals needed for lexical syntax.

⟨int⟩ ::= [´~´] ⟨nzdigit⟩ { ⟨digit⟩ }
| [´~´] 0 { ⟨octdigit⟩ }+
| [´~´] (´0x´ | ´0X´) { ⟨hexdigit⟩ }+
| [´~´] (´0b´ | ´0B´) { ⟨bindigit⟩ }+
⟨float⟩ ::= [´~´] { ⟨digit⟩ }+ ´.´ { ⟨digit⟩ } [(´e´ | ´E´) [´~´] { ⟨digit⟩ }+]
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨nzdigit⟩ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨octdigit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨hexdigit⟩ ::= ⟨digit⟩ | ´a´ | ´b´ | ´c´ | ´d´ | ´e´ | ´f´
| ´A´ | ´B´ | ´C´ | ´D´ | ´E´ | ´F´
⟨bindigit⟩ ::= 0 | 1

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

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. Unlike the previous sections which define token sequences, this section defines character sequences. An alphanumeric character is a letter (uppercase or lowercase), a digit, or an underscore character. Single quotes are used to delimit atom representations that may contain nonalphanumeric characters and backquotes are used in the same way for variable identifiers. Note 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.