The Oz Notation

Martin Henz
Leif Kornstaedt

Version 1.3.2
June 15, 2006

mozart

Abstract

Oz is a concurrent language providing for functional, object-oriented, and constraint programming. This document defines how Oz program text is transformed into an Oz Core program. Oz Core is a sublanguage of Oz designed to minimize syntactic complexity. Oz Core serves as the base for the definition of the semantics of Oz.

Technically, Oz Core allows to use several programming paradigms, including functional, constraint and object-oriented programming. Being a purely relational language, however, Oz Core does not provide easy notational access to programming methods from these paradigms, making it hard to fully exploit the capacities of the language.

It is such ergonomic considerations that lead to the development of the Oz Notation, where syntactic extensions provide convenient constructs for functional and object-oriented programming. The semantics of these extensions is defined in this document by their stepwise translation to Oz Core.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1

Introduction

This report defines how Oz program text, which is a sequence of characters, is transformed into an Oz Core program. This transformation is performed in three steps.

1. *Lexical Syntax* First, a given program text is transformed into a sequence of words. Each word represents a sequence of tokens. We call this process tokenizing.
2. *Context-free Syntax* The resulting sequence of tokens is transformed into a parse tree. We call this process parsing, and the resulting parse tree program.
3. *Core Programs* The program is translated to a Core program, eliminating a number of abbreviations and nesting.

At each step, errors may occur. A text represents an Oz program if it can be tokenized and parsed into a program which can be translated without error into a Core program.

Meta Notation In a document like this one, it is helpful to make use of notational conventions in order to provide for concise and precise descriptions.

1.1 Fonts

We make use of fonts to distinguish the different kinds of symbols occurring in this document:

Meaning	Examples
terminal or nonterminal symbol	⟨variable⟩, ⟨statement⟩
keyword	**`local`**, **`skip`**

1.2 Regular Expressions and Context-Free Grammars

Regular expressions and context free grammars describe sets of words. We use the following notation to describe one such set in terms of others (in increasing order of precedence):

Notation	Meaning
ε	singleton containing the empty word
(w)	grouping of regular expressions
[w]	union of ε with the set of words w
{ w }	set of words containing all concatenations of zero or more elements of w
{ w }+	set of words containing all concatenations of one or more elements of w
w_1 w_2	set of words containing all concatenations of an element of w_1 with an element of w_2
w_1 | w_2	union of w_1 and w_2
w_1 - w_2	difference of w_1 and w_2

2

Lexical Syntax

A program text is a sequence of characters represented by integers following ISO 8859-1 [1], also called ‘Latin 1’. In this section, we describe how such a sequence is split into a sequence of words. Each word represents zero or more tokens such that the result is a sequence of tokens. We call this process tokenization. In this section, we give regular expressions for the different kinds of words and describe the resulting tokens.

Resolving Ambiguities The splitting of a sequence of characters using these regular expressions is not unique. We use the usual left to right longest match tokenization obtaining either error or a unique sequence of tokens from a given sequence of characters. Longest match means that if two or more prefixes of the remaining character string are matched by (possibly different) regular expressions, we select the match that accepts the longest prefix. Note that the regular expressions are designed such that left to right longest match tokenization is unique.

Lexical Errors When no regular expression matches a prefix of the remaining character string, we speak of a lexical error. Such an input sequence does not represent a valid Oz program.

2.1 Character Class Definitions

This section defines character classes used in the regular expressions given in the remainder of the chapter. Note that these regular expressions do not–on their own–define any splitting of the input into words.

We use NUL to denote the ISO character with code 0 and ⟨any character⟩ to denote the set of all ISO characters.

⟨upper-case letter⟩ ::= `A` | ... | `Z` | `À` | ... | `Ö` | `Ø` | ... | `Þ`

⟨lower-case letter⟩ ::= `a` | ... | `z` | `ß` | ... | `ö` | `ø` | ... | `ÿ`

⟨digit⟩ ::= `0` | ... | `9`

⟨non-zero digit⟩ ::= `1` | ... | `9`

⟨alphanumerical⟩ ::= ⟨upper-case letter⟩ | ⟨lower-case letter⟩ | ⟨digit⟩ | `_`

⟨atom char⟩ ::= ⟨any character⟩ - (`'` | `\` | NUL)

⟨string char⟩ ::= ⟨any character⟩ - (`"` | `\` | NUL)

⟨variable char⟩ ::= ⟨any character⟩ - (`` ` `` | `\` | NUL)

⟨escape character⟩ ::= `a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `'` | `"` | `` ` `` | `&`

⟨octal digit⟩ ::= `0` | ... | `7`

⟨hex digit⟩ ::= `0` | ... | `9` | `A` | ... | `F` | `a` | ... | `f`

⟨binary digit⟩ ::= `0` | `1`

Pseudo-Characters In the classes of words ⟨variable⟩, ⟨atom⟩, ⟨string⟩, and ⟨character⟩ we use pseudo-characters, which represent single characters in different notations.

⟨pseudo char⟩ ::= `\` ⟨octal digit⟩ ⟨octal digit⟩ ⟨octal digit⟩
| `\` (`x` | `X`) ⟨hex digit⟩ ⟨hex digit⟩
| `\` ⟨escape character⟩

Pseudo-characters allow to enter any ISO 8859-1 character using octal or hexadecimal notation. Octal notation is restricted to numbers less than 256. The NUL character (ISO code 0) is forbidden. The pseudo-characters `\a` (= `\007`), `\b` (= `\010`), `\f` (= `\014`), `\n` (= `\012`), `\r` (= `\015`), `\t` (= `\011`), `\v` (= `\013`) denote special purpose characters, and `\\` (= `\134`), `\'` (= `\047`), `\"` (= `\042`), ``\` `` (= `\140`), `\&` (= `\046`) denote their second component character.

2.2 Spaces and Comments

Spaces are tab (code 9), newline (code 10), vertical tab (code 11), form feed (code 12), carriage return (code 13), and blank (code 32).

A comment is:

- a sequence of characters from `%` until the end of the line or file,
- a sequence of characters within and including the comment brackets `/*` and `*/`, in which `/*` and `*/` are properly nested, and
- the character `?`.

Spaces and comments produce no tokens. This means that they are ignored, except that they separate words form each other.

2.3 Keywords

```
⟨keyword⟩  ::=  andthen | at | attr | case | catch | choice
             |  class | cond | declare | define | dis
             |  div | else | elsecase | elseif | end
             |  export | fail | false | feat | finally | from
             |  fun | functor | if | import | in | local
             |  lock | meth | mod | not | of | or | orelse
             |  prepare | proc | prop | raise | require
             |  self | skip | then | thread | true | try
             |  unit | ( | ) | [ | ] | { | }
             |  | | # | : | ... | = | . | := | ^ | [] | $
             |  ! | _ | ~ | + | - | * | / | @ | <-
             |  , | !! | <= | == | \= | < | =< | >
             |  >= | =: | \=: | <: | =<: | >: | >=: | :: | :::
```

Each keyword represents itself as token.

2.4 Variables

```
⟨variable⟩  ::=  ⟨upper-case letter⟩ { ⟨alphanumerical⟩ }
              |  ` { ⟨variable char⟩ | ⟨pseudo char⟩ } `
```

A word of the form ⟨variable⟩ represents a variable token of the form ⌊`variable`, *n*+⌋, where *n*+ is the sequence of characters that make up the word, including the possibly surrounding ` characters.

For example, the word `Xs` represents the token ⌊`variable`, 88 115⌋ and the word `` `\n` `` represents the token ⌊`variable`, 96 10 96⌋. Variable tokens are denoted by the terminal symbol ⟨variable⟩ in the following context-free grammars.

2.5 Atoms

```
⟨atom⟩  ::=  ⟨lower-case letter⟩ { ⟨alphanumerical⟩ } - ⟨keyword⟩
          |  ' { ⟨atom char⟩ | ⟨pseudo char⟩ } '
```

A word of the form ⟨atom⟩ represents an atom token of the form ⌊`atom`, *n**⌋, where *n** is the sequence of characters that make up the word, excluding the possibly surrounding ' characters.

For example, the word `atom` represents the token ⌊`atom`, 97 116 111 109⌋ and the word `'\n'` represents the token ⌊`atom`, 10⌋. Atom tokens are denoted by the terminal symbol ⟨atom⟩ in the following context-free grammars.

2.6 Labels

```
⟨label⟩  ::=  ( ⟨variable⟩ | ⟨atom⟩ | true | false | unit ) (
```

A word of the form ⟨label⟩ represents a sequence of two tokens. The first is a label token of the form ⌊`variablelabel`, n+⌋, ⌊`atomlabel`, n*⌋ (similar to the corresponding tokens for words of the form ⟨variable⟩ and ⟨atom⟩), `truelabel`, `falselabel`, or `unitlabel`. The second token is the keyword `(`. For example, the word `Xs(` represents the tokens ⌊`variablelabel`, 88 115⌋ and `(`, and the word **`true(`** represents the tokens `truelabel` and `(`. The label tokens are denoted by the terminal symbols ⟨variable label⟩, ⟨atom label⟩, ⟨unit label⟩, ⟨true label⟩, and ⟨false label⟩ in the following context-free grammars.

2.7 Integers

⟨int⟩	::=	[`~`] (`0` | ⟨non-zero digit⟩ { ⟨digit⟩ })	*% decimal representation*
	|	[`~`] `0` { ⟨octal digit⟩ }+	*% octal representation*
	|	[`~`] `0` (`x` | `X`) { ⟨hex digit⟩ }+	*% hexadecimal representation*
	|	[`~`] `0` (`b` | `B`) { ⟨binary digit⟩ }+	*% binary representation*

A word of the form ⟨int⟩ represents an integer token of the form ⌊`int`, n⌋, where n represents the integer for which ⟨int⟩ is the representation.

For example, the word `~159` represents the token ⌊`int`, -159⌋, the word `077` the token ⌊`int`, 63⌋, the word `0xFF` the token ⌊`int`, 255⌋, and the word `~0b11111` the token ⌊`int`, -31⌋. Integer tokens are denoted by the terminal symbol ⟨int⟩ in the following context-free grammars.

2.8 Floats

⟨float⟩ ::= [`~`] { ⟨digit⟩ }+ `.` { ⟨digit⟩ } [(`e` | `E`) [`~`] { ⟨digit⟩ }+]

A word of the form ⟨float⟩ represents a float token of the form ⌊`float`, f⌋, where f represents the floating point number for which the word is the decimal representation. The letters `e` and `E` both indicate the exponent to 10.

For example, the word `~1.5e2` represents the token ⌊`float`, -150.0⌋. Float tokens are denoted by the terminal symbol ⟨float⟩ in the following context-free grammars.

The syntax of floats is implementation-dependent in that syntactically correct floats may be approximated by the compiler if they cannot be represented by the implementation.

2.9 Strings

⟨string⟩ ::= `"` { ⟨string char⟩ | ⟨pseudo char⟩ } `"`

The word `""` represents the token ⌊`atom`, 110 105 108⌋, which denotes the empty list `nil`. A word of the form `"`$c_1 \ldots c_m$`"`, where $m \geq 1$, represents a sequence of $m+2$ tokens of the form `[` $n_1 \ldots n_m$ `]`, where the n_i represent integer tokens according to the ISO 8859-1 code of c_i.

For example, the word `"ab"` represents the sequence of tokens `[` ⌊`int`, 97⌋ ⌊`int`, 98⌋ `]`.

2.10 Characters

$$\langle \text{character} \rangle \quad ::= \quad \texttt{\&} \; (\; \langle \text{any character} \rangle \text{ - } (\; \texttt{\textbackslash} \mid \text{NUL} \;) \mid \langle \text{pseudo char} \rangle \;)$$

A word of the form ⟨character⟩ represents the integer token according to the code of the character denoted by the word without the `&` prefix.

For example, the word `&a` represents the token $\lfloor$`int`, 97$\rfloor$.

3

Context-Free Syntax

In this section, we give a context-free grammar for a superset of Oz programs. Any sequence of tokens that is not a member of the language described by this grammar, starting from the $\langle\textsf{statement}\rangle$ nonterminal, is considered erroneous.

Implementations may accept a larger language, e.g., something more than only a statement at top-level, or treat lexical syntax that has no assigned meaning in the report as compiler directives.

3.1 The Base Language

Statements

```
⟨statement⟩ ::= ⟨statement⟩ ⟨statement⟩
              | local ⟨in statement⟩ end
              | ‘(’ ⟨in statement⟩ ‘)’
              | proc { ⟨atom⟩ } ‘{’ ⟨expression⟩ { ⟨pattern⟩ } ‘}’
                ⟨in phrase⟩
                end
              | fun { ⟨atom⟩ } ‘{’ ⟨expression⟩ { ⟨pattern⟩ } ‘}’
                ⟨in expression⟩
                end
              | ‘{’ ⟨expression⟩ { ⟨expression⟩ } ‘}’
              | if ⟨expression⟩ then ⟨in statement⟩
                [ ⟨else statement⟩ ]
                end
              | case ⟨expression⟩ of ⟨case statement clause⟩
                { [] ⟨case statement clause⟩ }
                [ ⟨else statement⟩ ]
                end
              | lock ⟨expression⟩ then ⟨in statement⟩ end
              | thread ⟨in statement⟩ end
              | try ⟨in statement⟩
                [ catch ⟨case statement clause⟩ { ‘[]’ ⟨case statement clause⟩ } ]
                [ finally ⟨in statement⟩ ]
                end
              | raise ⟨in expression⟩ end
              | ⟨expression⟩ ‘=’ ⟨expression⟩
              | ⟨expression⟩ ‘:=’ ⟨expression⟩
              | ⟨expression⟩ ‘.’ ⟨expression⟩ ‘:=’ ⟨expression⟩
              | skip
```

Expressions

```
⟨expression⟩ ::= local ⟨in expression⟩ end
               | ‘(’ ⟨in expression⟩ ‘)’
               | proc { ⟨atom⟩ } ‘{’ ‘$’ { ⟨pattern⟩ } ‘}’
                 ⟨in phrase⟩
                 end
               | fun { ⟨atom⟩ } ‘{’ ‘$’ { ⟨pattern⟩ } ‘}’
                 ⟨in expression⟩
                 end
               | ‘{’ ⟨expression⟩ { ⟨expression⟩ } ‘}’
               | if ⟨expression⟩ then ⟨in expression⟩
                 ⟨else expression⟩
                 end
               | case ⟨expression⟩ of ⟨case expression clause⟩
                 { [] ⟨case expression clause⟩ }
                 [ ⟨else expression⟩ ]
                 end
               | lock ⟨expression⟩ then ⟨in expression⟩ end
               | thread ⟨in expression⟩ end
               | try ⟨in expression⟩
                 [ catch ⟨case expression clause⟩ { ‘[]’ ⟨case expression clause⟩ } ]
                 [ finally ⟨in statement⟩ ]
                 end
               | raise ⟨in expression⟩ end
               | ⟨expression⟩ ‘=’ ⟨expression⟩
               | ⟨expression⟩ orelse ⟨expression⟩
               | ⟨expression⟩ andthen ⟨expression⟩
               | ⟨monop⟩ ⟨expression⟩
               | ⟨expression⟩ ⟨binop⟩ ⟨expression⟩
               | ⟨expression⟩ ‘:=’ ⟨expression⟩
               | ⟨expression⟩ ‘.’ ⟨expression⟩ ‘:=’ ⟨expression⟩
               | ⟨possibly escaped variable⟩
               | ‘_’
               | ⟨atom⟩ | ⟨int⟩ | ⟨float⟩
               | unit | true | false
               | ⟨label⟩ ‘(’ { ⟨subtree⟩ } [ ‘...’ ] ‘)’
               | ‘[’ { ⟨expression⟩ }+ ‘]’
               | ⟨expression⟩ ‘|’ ⟨expression⟩
               | ⟨expression⟩ { ‘#’ ⟨expression⟩ }+
               | ‘$’

⟨label⟩ ::= ⟨variable label⟩ | ⟨atom label⟩
          | ⟨unit label⟩ | ⟨true label⟩ | ⟨false label⟩

⟨feature⟩ ::= ⟨variable⟩ | ⟨atom⟩ | ⟨int⟩
            | unit | true | false

⟨subtree⟩ ::= [ ⟨feature⟩ ‘:’ ] ⟨expression⟩
```

Precedence Note that in both ⟨statement⟩s and ⟨expression⟩s there is potential ambiguity between ⟨expression⟩ ‘`:=`’ ⟨expression⟩ and ⟨expression⟩ ‘`.`’ ⟨expression⟩ ‘`:=`’ ⟨expression⟩. In fact ‘`. :=`’ is a ternary operator and has precedence. Parenthesis must be used for the alternate parse, that is, (⟨expression⟩‘`.`’⟨expression⟩) ‘`:=`’ ⟨expression⟩.

The assignment operators ‘`. :=`’ and ‘`:=`’, when used in expression position, perform an atomic exchange, the result of the operation being the previous value of the stateful entity assigned to.

Operators Expressions with operators need additional disambiguating rules introduced in Section 3.5.

⟨monop⟩ ::= ‘`~`’ | ‘`!!`’ | ‘`@`’

⟨binop⟩ ::= ‘`.`’ | ‘`^`’
| ‘`==`’ | ‘`\=`’ | ‘`<`’ | ‘`=<`’ | ‘`>`’ | ‘`>=`’
| ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | **`div`** | **`mod`**

Declarations A ⟨declaration part⟩ is a sequence of variables and statements. Singleton variables serve only for explicit declaration and are otherwise ignored. Variables within statements are implicitly declared if they occur at a pattern position. A prefixed escape (`!`) prevents implicit declaration.

⟨declaration part⟩ ::= ⟨variable⟩
| ⟨statement⟩
| ⟨declaration part⟩ ⟨declaration part⟩

⟨in statement⟩ ::= [⟨declaration part⟩ **`in`**] ⟨statement⟩

⟨in expression⟩ ::= [⟨declaration part⟩ **`in`**] [⟨statement⟩] ⟨expression⟩

⟨possibly escaped variable⟩ ::= [‘`!`’] ⟨variable⟩

As procedure body either a statement or an expression may be possible, depending on whether the procedure’s formal parameter patterns contain a nesting marker (`$`) or not.

⟨in phrase⟩ ::= ⟨in statement⟩
| ⟨in expression⟩

Patterns Pattern matching is performed as a top-down left-to-right sequence of tests. Record patterns test a value’s constructor; constant patterns and escaped variable patterns test for equality with the given value; nonlinearities (variables occurring multiply in one pattern) test for equality of the corresponding subtrees. Equation patterns and non-escaped variables introduce variable bindings.

⟨pattern⟩ ::= ⟨label⟩ '(' { ⟨subpattern⟩ } ['...'] ')'
| '[' { ⟨pattern⟩ }+ ']'
| ⟨pattern⟩ '|' ⟨pattern⟩
| ⟨pattern⟩ { '#' ⟨pattern⟩ }+
| ⟨atom⟩ | ⟨int⟩ | ⟨float⟩
| `unit` | `true` | `false`
| ⟨possibly escaped variable⟩
| '_'
| ⟨pattern⟩ '=' ⟨pattern⟩
| '(' ⟨pattern⟩ ')'

⟨subpattern⟩ ::= [⟨feature⟩ ':'] ⟨pattern⟩

Following the pattern an additional side condition can be given. It is only evaluated if the pattern matched, in the environment extended by the bindings introduced by the pattern. The variables introduced in the optional ⟨declaration part⟩ are also visible in the clause's body.

⟨case statement clause⟩ ::= ⟨pattern⟩ [`andthen` [⟨declaration part⟩ `in`] ⟨expressio
`then` ⟨in statement⟩

⟨case expression clause⟩ ::= ⟨pattern⟩ [`andthen` [⟨declaration part⟩ `in`] ⟨expressi
`then` ⟨in expression⟩

Else Clauses If the `else` part to an `if` statement is omitted, it is taken to be `else skip`. The `else` part to an `if` expression is mandatory.

If the `else` part to a `case` statement or expression is omitted and no pattern matches, an error exception is raised.

⟨else statement⟩ ::= `elseif` ⟨expression⟩ `then` ⟨in statement⟩
[⟨else statement⟩]
| `elsecase` ⟨expression⟩ `of` ⟨case statement clause⟩
{ '[]' ⟨case statement clause⟩ }
[⟨else statement⟩]
| `else` ⟨in statement⟩

⟨else expression⟩ ::= `elseif` ⟨expression⟩ `then` ⟨in expression⟩
⟨else expression⟩
| `elsecase` ⟨expression⟩ `of` ⟨case expression clause⟩
{ '[]' ⟨case expression clause⟩ }
[⟨else expression⟩]
| `else` ⟨in expression⟩

3.2 Constraint Extensions and Combinators

Statements

⟨statement⟩ += ⟨fd compare⟩
| ⟨fd in⟩
| **fail**
| **not** ⟨in statement⟩ **end**
| **cond** ⟨cond statement clause⟩
{ ‘**[]**’ ⟨cond statement clause⟩ }
[**else** ⟨in statement⟩]
end
| **or** ⟨dis statement clause⟩ { ‘**[]**’ ⟨dis statement clause⟩ }+ **end**
| **dis** ⟨dis statement clause⟩ { ‘**[]**’ ⟨dis statement clause⟩ }+ **end**
| **choice** ⟨in statement⟩ { ‘**[]**’ ⟨in statement⟩ } **end**

⟨cond statement clause⟩ ::= [⟨declaration part⟩ **in**] ⟨statement⟩ **then** ⟨in statemen

⟨dis statement clause⟩ ::= [⟨declaration part⟩ **in**] ⟨statement⟩ [**then** ⟨in statement

Expressions

⟨expression⟩ += ⟨fd compare⟩
| ⟨fd in⟩
| **fail**
| **cond** ⟨cond expression clause⟩
{ ‘**[]**’ ⟨cond expression clause⟩ }
[**else** ⟨in expression⟩]
end
| **or** ⟨cond expression clause⟩ { ‘**[]**’ ⟨cond expression clause⟩ }+ **e**
| **dis** ⟨cond expression clause⟩ { ‘**[]**’ ⟨cond expression clause⟩ }+
| **choice** ⟨in expression⟩ { ‘**[]**’ ⟨in expression⟩ } **end**

⟨cond expression clause⟩ ::= [⟨declaration part⟩ **in**] ⟨statement⟩ **then** ⟨in express

⟨fd compare⟩ ::= ⟨expression⟩ (‘**=:**’ | ‘**\=:**’ | ‘**<:**’ | ‘**=<:**’ | ‘**>:**’ | ‘**>=:**’) ⟨expression

⟨fd in⟩ ::= ⟨expression⟩ (‘**::**’ | ‘**:::**’) ⟨expression⟩

3.3 Class Extensions

Class Definitions

⟨statement⟩ += **class** ⟨expression⟩
{ ⟨class descriptor⟩ }
{ ⟨method⟩ }
end

```
⟨expression⟩  +=  class [ '$' ]
                  { ⟨class descriptor⟩ }
                  { ⟨method⟩ }
                  end
```

```
⟨class descriptor⟩  ::=  from { ⟨expression⟩ }+
                      |  prop { ⟨expression⟩ }+
                      |  attr { ⟨attr or feat⟩ }+
                      |  feat { ⟨attr or feat⟩ }+
```

Non-escaped variables are implicitly introduced with class scope, bound to new names. This allows to model private components.

```
⟨attr or feat⟩  ::=  [ '!' ] ⟨variable⟩ | ⟨atom⟩ | ⟨int⟩
                  |  unit | true | false
```

Methods The first-class message used to invoke a method can be referenced by appending `=` ⟨variable⟩ to the method head. This message does not contain defaulted arguments (see below) if they have not been explicitly given.

```
⟨method⟩  ::=  meth ⟨method head⟩ [ '=' ⟨variable⟩ ]
               ⟨in phrase⟩
               end
```

If dots are given, any additional features are allowed in the first-class message; else, extraneous features cause an error exception to be raised.

```
⟨method head⟩  ::=  [ '!' ] ⟨variable⟩ | ⟨atom⟩
                 |  unit | true | false
                 |  ⟨method head label⟩ '(' { ⟨method formal⟩ } [ '...' ] ')'
```

```
⟨method head label⟩  ::=  [ '!' ] ⟨variable label⟩ | ⟨atom label⟩
                       |  ⟨unit label⟩ | ⟨true label⟩ | ⟨false label⟩
```

A default `<=` after a formal argument allows for the corresponding actual argument to be omitted from a first-class method. In this case, the default expression will be evaluated (inside the method) and the formal argument variable bound to the result.

```
⟨method formal⟩  ::=  [ ⟨feature⟩ ':' ] ( ⟨variable⟩ | '_' | '$' )
                      [ '<=' ⟨expression⟩ ]
```

Operations To the following operators, `self` is an implicit operand. Their use is syntactically restricted to the body of method definitions.

```
⟨statement⟩  +=  lock ⟨in statement⟩ end
              |  ⟨expression⟩ ':=' ⟨expression⟩
              |  ⟨expression⟩ '<-' ⟨expression⟩
              |  ⟨expression⟩ ',' ⟨expression⟩
```

The assignment operators ‘`:=`’, ‘`<-`’, when used in an expression position, perform an atomic exchange, the result of the operation being the previous value of the attribute assigned to.

⟨expression⟩ += **lock** ⟨in expression⟩ **end**
| ‘`@`’⟨expression⟩
| ⟨expression⟩ ‘`:=`’ ⟨expression⟩
| ⟨expression⟩ ‘`<-`’ ⟨expression⟩
| ⟨expression⟩ ‘`,`’ ⟨expression⟩
| **self**

3.4 Functor Extensions

A functor definition creates a chunk with (at least) features `'import'` and `'export'` describing its interface and a feature `apply` containing a procedure mapping an import record to an export module.

⟨statement⟩ += **functor** ⟨expression⟩ { ⟨functor descriptor⟩ } **end**

⟨expression⟩ += **functor** [‘`$`’] { ⟨functor descriptor⟩ } **end**

Import Specification The import specification names values (usually modules) to be made available to the body. They represent formal arguments to the body abstraction. The additional **at** clause allows to specify where the actual argument is to come from. This must be an atom (interpreted as a relative URL) so that a functor creating the referenced module may be located at compile time.

⟨functor descriptor⟩ ::= **import** { ⟨import clause⟩ }+

⟨import clause⟩ ::= ⟨variable⟩ [**at** ⟨atom⟩]
| ⟨variable label⟩ ⟨import features⟩ [**at** ⟨atom⟩]

If the expected structure of an imported value is partially specified, occurrences of the module name are restricted to a single syntactic context: the first operand in applications of the dot operator, where the second operand is one of the features mentioned in the import specification.

⟨import features⟩ ::= ‘`(`’ { ⟨module feature⟩ ⟨import alias⟩ }+ ‘`)`’

⟨module feature⟩ ::= ⟨atom⟩ | ⟨int⟩

An import alias introduces a variable bound to one of the imported module’s subtrees.

⟨import alias⟩ ::= [‘`:`’ ⟨variable⟩]

Functor Body The body of the functor is a statement (usually a sequence of definitions that compute the exported values). This statement is a pattern position. Note the difference between this abbreviated declaration and the ⟨in statement⟩ rule: The ⟨statement⟩ following the `in` keyword is optional, not the ⟨declaration part⟩ preceding it.

⟨functor descriptor⟩ += `define` ⟨declaration part⟩ [`in` ⟨statement⟩]

Export Specification The export specification specifies the structure the modules created by applications of this functor will have.

⟨functor descriptor⟩ += `export` { [⟨module feature⟩ ‘`:`’] ⟨variable⟩ }+

The value of the variables mentioned in the export declaration are made available under the given features. If a feature is omitted, then it is computed from the corresponding variable's print name by changing its initial capital letter into a lower-case letter (unless it's a backquote variable, in which case the print name is taken as-is).

All variables introduced in the import and the body are visible in the export declaration.

Computed Functors A functor that contains one of the following additional functor descriptors is called a computed functor. The `require` and `prepare` clauses correspond to the `import` and `define` clauses respectively, only they are executed upon functor definition instead of functor application. The variables introduced by these clauses are visible in the `define` and `export` clauses.

⟨functor descriptor⟩ += `require` { ⟨import clause⟩ }+
| `prepare` ⟨declaration part⟩ [`in` ⟨statement⟩]

3.5 Operator Associativity and Precedence

The grammar given above is ambiguous. Some ambiguities do not affect the semantics (such as associativity of ⟨statement⟩s and ⟨declaration part⟩s). Those that do are resolved according to the following table stating the associativity of operators in increasing order of precedence:

Operators	Associativity
X=Y	right
X<-Y X:=Y X.Y:=Z	right
X **orelse** Y	right
X **andthen** Y	right
X==Y X\=Y X<Y X=<Y X>Y X>=Y	
X=:Y X\=:Y X<:Y X=<:Y X>:Y X>=:Y	none
X::Y X:::Y	none
X|Y	right
X#Y	mixfix
X+Y X-Y	left
X*Y X/Y X **div** Y X **mod** Y	left
X,Y	right
~X	prefix
X.Y X^Y	left
@X !!X	prefix

'Having higher precedence' means 'binding tighter'; e.g., the expression `c#X.g = Y` is parsed as `(c#(X.g)) = Y`.

Attempts to exploit associativity of non-associative operators (without using parentheses to make the intention clear), as in `X < Y < Z`, are considered erroneous.

4

Core Programs

In this section, we give a context-free grammar for Core Oz programs.

4.1 The Base Language

Statements

⟨statement⟩ ::= ⟨statement⟩ ⟨statement⟩
| **local** { ⟨variable⟩ }+ **in** ⟨statement⟩ **end**
| **proc** { ⟨atom⟩ } '`{`' ⟨variable⟩ { ⟨variable⟩ } '`}`'
⟨statement⟩
end
| '`{`' ⟨variable⟩ { ⟨variable⟩ } '`}`'
| **lock** ⟨variable⟩ **then** ⟨statement⟩ **end**
| **thread** ⟨statement⟩ **end**
| **try** ⟨statement⟩
catch ⟨variable⟩ **then** ⟨statement⟩
end
| ⟨variable⟩ '`=`' '`@`' ⟨variable⟩
| ⟨variable⟩ '`:=`' ⟨variable⟩
| ⟨variable⟩ '`=`' ⟨variable⟩ '`:=`' ⟨variable⟩
| ⟨variable⟩ '`=`' ⟨expression⟩
| **skip**

Expressions

⟨expression⟩ ::= ⟨variable⟩
| ⟨atom⟩ | ⟨int⟩ | ⟨float⟩
| ⟨label⟩ '`(`' { ⟨feature⟩ '`:`' ⟨expression⟩ } ['`...`'] '`)`'

⟨label⟩ ::= ⟨variable label⟩ | ⟨atom label⟩

⟨feature⟩ ::= ⟨variable⟩ | ⟨atom⟩ | ⟨int⟩

4.2 Class Extensions

```
⟨statement⟩  +=  class ⟨variable⟩
                 [ from { ⟨variable⟩ }+ ]
                 [ prop { ⟨variable⟩ }+ ]
                 [ attr { ‘!’ ⟨variable⟩ [ ‘:’ ⟨variable⟩ ] }+ ]
                 [ feat { ‘!’ ⟨variable⟩ [ ‘:’ ⟨variable⟩ ] }+ ]
                 { ⟨method⟩ }
                 end
              |  lock ⟨statement⟩ end
              |  ⟨variable⟩ ‘<-’ ⟨variable⟩
              |  ⟨variable⟩ ‘=’ ⟨variable⟩ ‘<-’ ⟨variable⟩
              |  ⟨variable⟩ ‘,’ ⟨variable⟩
              |  ⟨variable⟩ ‘=’ self
```

Methods

```
⟨method⟩  ::=  meth ‘!’ ⟨variable⟩ ‘(’ ‘...’ ‘)’ ‘=’ ⟨variable⟩
               ⟨statement⟩
               end
```

5

Translation of Oz Programs to Oz Core Programs

Oz programs are translated to Oz core programs by repeatedly applying the rules given in this chapter to subtrees of the parse tree, replacing the subtree with the result of the rule. A rule consists of the following:

A set of nonterminals. The rule is only applicable to subtrees generated by a rule of one of these nonterminals.

A left-hand side. The rule is only applicable if the subtree's structure matches the left-hand side pattern. Additionally, variables are introduced. Some parts may be left out (replaced by an ellipsis) if they reappear unmodified in the output.

A right-hand side. When the rule is applied to a subtree, the latter is replaced by the subtree specified by the right-hand side. This may contain variables written as X, Y, or Z not appearing in the left-hand side: These variables are supposed to be fresh such that no capturing can occur.

Optionally, a side condition. The rule is only applicable if the side-condition is satisfied.

Meta Variables Inside rewrite rules, we use meta variables for terminals and phrases generated by nonterminals as shown in the following table:

Meta Variables	Corresponding Terminals and Nonterminals
x, *x1*, . . . , *xn*	⟨variable⟩
D	⟨declaration part⟩
S	⟨statement⟩
E, *E1*, . . . , *Ek*, *En*	⟨expression⟩
SE	⟨statement⟩ or ⟨expression⟩
P, *P1*, . . . , *Pk*, *Pn*	⟨pattern⟩
EP, *EP1*, . . . , *EPn*	⟨expression⟩ or ⟨pattern⟩
C, *C1*, . . . , *Cn*	⟨case statement clause⟩ or ⟨case expression clause⟩
L, *L1*, . . . , *Ln*	⟨cond statement clause⟩ or ⟨cond expression clause⟩ or ⟨dis statement c
l	⟨label⟩
f1, . . . , *fn*	⟨feature⟩
s1, . . . , *sn*	⟨subtree⟩ or ⟨subpattern⟩

Core Variables The result of the transformation may have references to so-called Core variables. We indicate this by writing them in backquotes; they are not bound lexically, but are looked up in static environment. Examples are `‘List.toRecord‘` and `‘RaiseDebugCheck‘`. If the print name of a Core variable contains a dot, then it is supposed to be looked up (without the backquotes) in the Base Environment (see *“The Oz Base Environment”*).

Errors When no rule is applicable and the program is not an Oz Core program, we speak of a syntax error. Such a program is not a valid Oz program.

5.1 The Base Language

Declarations

⟨in statement⟩ ::=		
D **in** *S*	→	**local** *D* **in** *S* **end**

⟨in expression⟩ ::=		
D **in** [*S*] *E*	→	**local** *D* **in** [*S*] *E* **end**

The following rule makes implicit declarations explicit, i.e., declarations only name variables between `local` and `in`. We need an auxiliary definition: The function *PV* returns the set of pattern variables of a statement (or expression). Furthermore, we call a position *p* in a given statement *S* a pattern position iff the following holds: If the subterm at position *p* of *S* is replaced by a fresh variable X, then $X \in PV(S[X/p])$.

D	*PV*(*D*)
D1 D2	$PV(D1) \cup PV(D2)$
x	$\{x\}$
`(`*S*`)`	$PV(S)$
`(`*D* `in` *S*`)`	$PV(S)$ - $PV(D)$
`local` *D* `in` *S* `end`	$PV(S)$ - $PV(D)$
`proc ... {`*E* `...} ... end`	$PV(E)$
`fun ... {`*E* `...} ... end`	$PV(E)$
`class` *E* `... end`	$PV(E)$
`functor` *E* `... end`	$PV(E)$
E `= ...`	$PV(E)$
otherwise	$\emptyset$

E	$PV(E)$
x	$\{x\}$
`(`E`)`	$PV(E)$
`(`D `in` [S] E`)`	$(PV(S) \cup PV(E))$ - $PV(D)$
`local` D `in` [S] E `end`	$(PV(S) \cup PV(E))$ - $PV(D)$
$E1$ `=` $E2$	$PV(E1) \cup PV(E2)$
`[`$E1$ ... En`]`	$PV(E1) \cup \ldots \cup PV(En)$
$E1$`|`$E2$	$PV(E1) \cup PV(E2)$
$E1$`#`...`#`En	$PV(E1) \cup \ldots \cup PV(En)$
l`(`[$f1$`:`] $E1$ ... [fn`:`] En [`...`]`)`	$PV(E1) \cup \ldots \cup PV(En)$
otherwise	$\emptyset$

⟨statement⟩ ::=		
local D **in** S **end**	→	**local** $x1 \ldots xn$ **in** D’ S **end**
if D is not a sequence of distinct variables and where $\{x1, \ldots, xn\}$ = $PV(D)$ and D’ is D with singleton variables and escapes in pattern position removed.		

⟨statement⟩ ::=		
x = **local** D **in** [S] E **end**	→	**local** X **in** X = x **local** D **in** [S] X = E **end** **end**

Grouping

⟨statement⟩ ::=		
(S)	→	S

⟨expression⟩ ::=		
(E)	→	E

Procedure Definitions

⟨statement⟩ ::=		
proc ... $\{E\ P1 \ldots Pn\}$ SE **end**	→	**local** X **in** X = E **proc** ... {X $P1 \ldots Pn$} SE **end** **end**
if E is no variable.		

⟨statement⟩ ::=		
⟨expression⟩ ::=		

fun ... lazy ... {*E1 P1* ... *Pn*}
  E2
end
	→	**fun** ... {*E1* X1 ... X*n*}
		  {‘Value.byNeed‘
		  fun {$}
		    case X1#...#X*n* **of** *P1*#...#*Pn* **then** *E2*
		    end
		  end}
		end

where all occurrences of `lazy` are removed from the procedure flags.		

⟨statement⟩ ::=		
⟨expression⟩ ::=		

fun ... {*E1 P1* ... *Pn*}
  E2
end
	→	**proc** ... {*E1 P1* ... *Pn* $}
		  E2
		end

if no `$` occurs in *P1* ... *Pn* and no `lazy` occurs in the procedure flags.		

⟨statement⟩ ::=		
⟨expression⟩ ::=		

proc ... {*E1 P1* ... *Pk* ... *Pn*}
  E2
end
	→	**proc** ... {*E1 P1* ... *Pk’* ... *Pn*}
		  X = *E2*
		end

if `$` occurs in *Pk* and no other `$` occurs in *P1* ... *Pn* and no `lazy` occurs in the procedure flags. *Pk’* is the result of replacing the `$` in *Pk* by X.		

⟨statement⟩ ::=		
⟨expression⟩ ::=		

proc ... {*E P1* ... *Pn*}
  S
end
	→	**proc** ... {*E* X1 ... X*n*}
		  case X1#...#X*n* **of** *P1*#...#*Pn* **then** *S*
		  end
		end

if *P1* ... *Pn* are not distinct variables and no `$` occurs in *P1* ... *Pn* and no `lazy` occurs in the procedure flags.		

⟨statement⟩ ::=		
x = **proc** ... {$ ...} *SE* **end**	→	**proc** ... {*x* ...} *SE* **end**

Applications

Actual arguments are evaluated from left to right and after the designator expression.

⟨statement⟩ ::=		
{*E1* ... *Ek* ... *En*}	→	**local** X **in** X = *Ek* {*E1* ... X ... *En*} **end**
if *Ek* is no variable and all *Ei* with $i < k$ are variables.		

⟨statement⟩ ::=		
x = {*E E1* ... *En*}	→	{*E E1* ... *En x*}
if no $ occurs in *E1* ... *En* in pattern position.		

⟨statement⟩ ::=		
x = {*E E1* ... *Ek* ... *En*}	→	{*E E1* ... *Ek'* ... *En*}
if $ occurs in *Ek* in pattern position and no other $ occurs in *E1* ... *En* in pattern position. *Ek'* is the result of replacing the $ in pattern position in *Ek* by *x*.		

Boolean and Pattern-Matching Conditionals

⟨else statement⟩ ::=		
⟨else expression⟩ ::=		
elseif ...	→	**else if** ... **end**

⟨else statement⟩ ::=		
⟨else expression⟩ ::=		
elsecase ...	→	**else case** ... **end**

⟨statement⟩ ::=		
if *E* **then** *S* **end**	→	**if** *E* **then** *S* **else skip** **end**

⟨expression⟩ ::=		
if *E1* **then** *E1* **end**	→	**if** *E1* **then** *E2* **else** **raise** error(kernel(noElse ...) ...) **end** **end**
where the omitted parts of the exception are implementation-dependent.		

⟨statement⟩ ::=		
⟨expression⟩ ::=		
if *E* **then** *SE1* **else** *SE2* **end**	→	**case** *E* **of true then** *SE1* **[] false then** *SE2* **else** **raise** error(kernel(boolCaseType ...) ...) **end** **end**
where the omitted parts of the exception are implementation-dependent.		

```
⟨statement⟩ ::=
⟨expression⟩ ::=
case E of ... end   →   local X in
                           X = E
                           case X of ... end
                        end
    if E is no variable.
```

```
⟨statement⟩ ::=
⟨expression⟩ ::=
case E of C1 [] ... [] Cn   →   case E of C1 [] ... [] Cn
end                             else
                                   raise error(kernel(noElse ...) ...) end
                                end
    where the omitted parts of the exception are implementation-
    dependent.
```

Note: Missing: expansion of `case` statement/expression to `cond`

Locks

```
⟨statement⟩ ::=
⟨expression⟩ ::=
lock E then SE end   →   local X in
                            X = E
                            lock X then SE end
                         end
    if E is no variable.
```

```
⟨statement⟩ ::=
x = lock E1 then E2 end   →   lock E1 then x = E2 end
```

Threads

```
⟨statement⟩ ::=
x = thread E end   →   thread x = E end
```

Exception Handling

```
⟨statement⟩ ::=
⟨expression⟩ ::=
try SE1                   →   try SE1
catch C1 [] ... [] Cn         catch X then
[ finally S ]                    case X of C1 [] ... [] Cn
end                              else
                                    raise X end
                                 end
                              [ finally S ]
                              end
    if C1 [] ... [] Cn does not have the form x then SE2.
```

In the following rule, the intermediate variable X ensures that *x* is only bound iff evaluation of *E* does not raise an exception.

```
⟨statement⟩ ::=
x = try E                →  try X in
    [ catch y then E ]         X = E
    [ finally S ]              x = X
    end                      [ catch y then x = E ]
                             [ finally S ]
                             end
```

```
⟨statement⟩ ::=
try ...       →  local X in
finally S          X = try
end                       try ... end
                          unit
                        catch Y then ex(Y)
                        end
                   S
                   case X of ex(Z) then
                      raise Z end
                   else skip
                   end
                 end
```

```
⟨statement⟩ ::=
try SE end   →   SE
```

Exception Raising

```
⟨statement⟩ ::=
raise E end   →   {‘Exception.raise‘ E}
```

```
⟨statement⟩ ::=
x = raise E end   →   raise E end
```

Equations

```
⟨statement⟩ ::=
E1 = E2   →   local X in
                X = E1
                X = E2
              end
    if E1 is no variable.
```

Operators

⟨expression⟩ ::=
$o\ E \quad \longrightarrow \quad \{x\ E\}$
where $o \in \{$`!!`, `~`$\}$ and $x = CV(o)$.

⟨expression⟩ ::=
$E1\ o\ E2 \quad \longrightarrow \quad \{x\ E1\ E2\}$
where $o \in \{$`.`, `^`, `*`, `/`, `div`, `mod`, `+`, `-`, `==`, `\=`, `<`, `=<`, `>`, `>=`$\}$ and $x = CV(o)$.

$CV(o)$ denotes the Core variable to which operation o is bound. The following table summarizes in which module from *"The Oz Base Environment"* each operator is available, e.g., `+` is available as `Number.'+'`, which means that $CV(o)$ = `'Number.'+''`.

Operators	Located in Module
`!! . == \= < =< > >=`	`Value`
`~ * + -`	`Number`
`div mod`	`Int`
`/`	`Float`
`^`	`Record`

⟨expression⟩ ::=
E1 **andthen** *E2* $\longrightarrow$ **if** *E1* **then** *E2* **else false** **end**

⟨expression⟩ ::=
E1 **orelse** *E2* $\longrightarrow$ **if** *E1* **then true** **else** *E2* **end**

Records

⟨expression⟩ ::=
⟨pattern⟩ ::=
[*EP1* … *EPn*] $\longrightarrow$ *EP1*|…|*EPn*|nil

⟨expression⟩ ::=
⟨pattern⟩ ::=
EP1|*EP2* $\longrightarrow$ '|'(*EP1* *EP2*)

⟨expression⟩ ::=
⟨pattern⟩ ::=
EP1#…#*EPn* $\longrightarrow$ '#'(*EP1* … *EPn*)

Note: Missing: dots, omitted features

Uniform State

⟨statement⟩ ::=
x = @E → **local** X **in** X = E x = @X **end**
if E is no variable.

⟨statement⟩ ::=
$E1$.$E2$:= $E3$ → $E1$#$E2$:= $E3$

⟨statement⟩ ::=
x = $E1$.$E2$:= $E3$ → x = $E1$#$E2$:= $E3$

⟨statement⟩ ::=
$E1$:= $E2$ → **local** X **in** X = $E1$ X := $E2$ **end**
if $E1$ is no variable.

⟨statement⟩ ::=
x := E → **local** X **in** X = E x := X **end**
if E is no variable.

⟨statement⟩ ::=
x = $E1$:= $E2$ → **local** X **in** X = $E1$ x = X := $E2$ **end**
if $E1$ is no variable.

⟨statement⟩ ::=
x = y := E → **local** X **in** X = E x = y := X **end**
if E is no variable.

Wildcard

⟨expression⟩ ::=
_ → **local** X **in** X **end**

⟨pattern⟩ ::=
_ → X

Named Constants

⟨expression⟩ ::=
⟨label⟩ ::=
⟨feature⟩ ::=
unit → ‘Unit.’unit’‘

⟨pattern⟩ ::=
unit → **!**‘Unit.’unit’‘

⟨expression⟩ ::=
⟨label⟩ ::=
⟨feature⟩ ::=
true → ‘Bool.’true’‘

⟨pattern⟩ ::=
true → **!**‘Bool.’true’‘

⟨expression⟩ ::=
⟨label⟩ ::=
⟨feature⟩ ::=
false → ‘Bool.’false’‘

⟨pattern⟩ ::=
false → **!**‘Bool.’false’‘

5.2 Constraint Extensions and Combinators

Operators

Note: Missing: fd compare

⟨statement⟩ ::=
E1 **::** *E2* → {‘FD.int‘ *E2* *E1*}

⟨statement⟩ ::=
E1 **:::** *E2* → {‘FD.dom‘ *E2* *E1*}

⟨expression⟩ ::=
E1 **::** *E2* → {‘FD.reified.int‘ *E2* *E1*}

⟨expression⟩ ::=
E1 **:::** *E2* → {‘FD.reified.dom‘ *E2* *E1*}

Failure

⟨statement⟩ ::=
⟨expression⟩ ::=
fail → **raise** failure(…) **end**
where the omitted parts of the exception are implementation-dependent.

Combinators

⟨statement⟩ ::=
not *S* **end** → {‘Combinator.’not’‘ **proc** {$} *S* **end**}

⟨statement⟩ ::=
⟨expression⟩ ::=
cond *L1* **[]** … **[]** *Ln* **end** → **cond** *L1* **[]** … **[]** *Ln* **else** **raise** error(kernel(noElse …) …) **end** **end**
where the omitted parts of the exception are implementation-dependent.

⟨cond statement clause⟩ ::=
⟨dis statement clause⟩ ::=
D **in** *S1* [**then** *S2*] → *x1* … *xn* **in** *D’ S1* [**then** *S2*]
if *D* is not a sequence of distinct variables and where {*x1*, …, *xn*} = *PV*(*D*) and *D’* is *D* with singleton variables and escapes removed.

⟨cond expression clause⟩ ::=
D **in** *S* **then** *E* → *x1* … *xn* **in** *D’ S* **then** *E*
if *D* is not a sequence of distinct variables and where {*x1*, …, *xn*} = *PV*(*D*) and *D’* is *D* with singleton variables and escapes removed.

Note: Missing: translation of `cond`/`or`/`dis`/`choice` expression into statement The following rewrite rules make use of an auxiliary function *Proc*, defined as follows:

L	*Proc*(*L*)
`S1 in S2`	`proc {$} S1 in S2 end`
`S1 in S2 then S3`	`fun {$} S1 in S2 proc {$} S3 end end`

⟨statement⟩ ::=
cond *L1* **[]** … **[]** *Ln* **else** *S* **end** → {‘Combinator.’cond’‘ ’#’(*Proc*(*L1*) … *Proc*(*Ln*)) **proc** {$} *S* **end**}

⟨statement⟩ ::=
or *L1* **[]** ... **[]** *Ln* → {'Combinator.'or'' '#'(*Proc*(*L1*) ... *Proc*(*Ln*))} **end**

⟨statement⟩ ::=
dis *L1* **[]** ... **[]** *Ln* → {'Combinator.'dis'' '#'(*Proc*(*L1*) ... *Proc*(*Ln*))} **end**

⟨statement⟩ ::=
choice *S1* **[]** ... **[]** *Sn* → **case** {'Space.choose' *n*} **of** 1 **then** *S1* **[]** ... **[]** *n* **then** *Sn* **end**

5.3 Class Extensions

Classes

⟨statement⟩ ::=
x = **class** [$] ... **end** → **class** *x* ... **end**

Method Names

⟨method head⟩ ::=
⟨method head label⟩ ::=
unit → **!**'Unit.'unit''

⟨method head⟩ ::=
⟨method head label⟩ ::=
true → **!**'Bool.'true''

⟨method head⟩ ::=
⟨method head label⟩ ::=
false → **!**'Bool.'false''

Locks

⟨statement⟩ ::=
x = **lock** *E* **end** → **lock** *x* = *E* **end**

Operators

⟨statement⟩ ::=
E1 <- *E2* → **local** X **in** X = *E1* X <- *E2* **end**
if *E1* is no variable.

⟨statement⟩ ::=
x <- *E* → **local** X **in** X = *E* x <- X **end**
if *E* is no variable.

⟨statement⟩ ::=
x = *E1* <- *E2* → **local** X **in** X = *E1* *x* = X <- *E2* **end**
if *E1* is no variable.

⟨statement⟩ ::=
x = *y* <- *E* → **local** X **in** X = *E* *x* = *y* <- X **end**
if *E* is no variable.

⟨statement⟩ ::=
x = *E1*, *E2* → *E1*, *E2'*
if exactly one $ occurs in *E2* in pattern position. *E2'* is the result of replacing this $ in *E2* by *x*.

⟨statement⟩ ::=
E1, *E2* → **local** X **in** X = *E1* X, *E2* **end**
if *E1* is no variable.

⟨statement⟩ ::=
x, *E* → **local** X **in** X = *E* x, X **end**
if *E* is no variable.

5.4 Functor Extensions

Bibliography

[1] Information processing – 8-bit single-byte coded graphic character sets – part 1: Latin, alphabet no. 1. Technical Report ISO 8859-1:1987, Technical committee: JTC 1/SC 2, International Organization for Standardization, 1987.