The Mozart Compiler

Leif Kornstaedt

Version 1.3.2
June 15, 2006

Abstract

This document describes how to use the Oz Compiler in Mozart. This includes the descriptions of the batch compiler and compiler panel as well as of the application programmer's interface to the compiler.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1 Introduction

The Mozart Compiler is, in principle, only a special kind of evaluator. In general, an evaluator implements the mapping:

$$\text{source\_text} \times \text{environment} \rightarrow \text{value}$$

Compiling Programs Performing evaluation of Oz programs with a compiler has some advantages:

- Programs with statically discoverable errors are rejected. Apart from syntax errors and undeclared variables, this also includes discouraged uses of the language that are not–strictly speaking–necessarily errors. For instance, applying a procedure with the wrong number of arguments does raise a catchable exception, but may be reported as an error.
- Programs may be translated into a more efficient representation. In the case of the Mozart Compiler, this consists of bytecode for the Mozart VM.

For correct programs, both these steps are transparent to the user, but due to them the transformation has actually more parameters and output than the general evaluator illustrated above.

The Manual's Structure The remainder of this chapter will describe what the state of the Mozart Compiler consists of, and what additional parameters compilation takes into account. Chapter 2 describes what programs the compiler accepts as input. Chapter 3 describes two applications of the compiler, namely the batch compiler, which is invoked from the command line, and the compiler panel, which is a graphical interface to the compiler's state. Both of these are implemented using the compiler's Application Programmer's Interface, which is described in detail in Chapter 4.

Note that there is another widely used application of the compiler, namely the Oz Programming Interface. See Section *Seeing the OPI from Mozart, (The Oz Programming Interface)* for a description.

The Compiler's State

This section describes the components of the compiler's internal state as well as their initial values.

Macro Definitions The compiler stores a set of so-called macro names, used to control conditional compilation via macro directives such as `\ifdef` (see Section 2.2).

Initially, the set of macro names consists of entries describing the system's version number, for instance, when running Mozart 1.1.0, all of `Mozart_1`, `Mozart_1_1`, and `Mozart_1_1_0` are macro names.

Switches and Options The compiler's behaviour is influenced by a number of boolean switches and non-boolean options. While the switch settings can conveniently be changed by several methods as described later, the options are only accessible to users of the application programmer's interface.

The active switch settings are given by a mapping from switch names to boolean values. The compiler manages a stack of switch states, the top element of which is taken as the active state. This allows to temporarily escape into a different mode of compiler operation.

The available switches, their initial settings and their effects on the compilation process are described in detail in Appendix A.

The Environment The Oz compiler does not take the environment as input for each evaluation as was illustrated above, but stores its active environment. This may be extended as a side-effect of the compilation, and it may be replaced by other environments at will.

An environment is a mapping from variable print names to arbitrary Oz values. Initially, the environment consists of the variables defined by the Oz Base Environment *"The Oz Base Environment"*.

The Query Queue Since the Oz compiler has internal state, it is not implemented as a function as described above, but as an active object that sequentializes all requests made to it via a query queue.

2 Directives

Directive Syntax The Mozart compiler understands the full syntax of Oz programs. Additionally, it also accepts directives in its input. A directive can start anywhere on a line and is introduced by a backslash; it always extends until the end of the line.

Directives come in two flavors. So-called compiler directives provide a way to change the compiler's switches, whereas macro directives can be used for inserting files or performing compilation conditionally. While macro directives may appear between any two tokens in the input, the use of compiler directives underlies restrictions as described below.

Compilation Units A single input file is split into a number of compilation units as follows:

- Each compiler directive forms a compilation unit.
- Each `declare` statement starts a compilation unit, which includes all following non-`declare` statements.
- A sequence of statements without `declare` forms a compilation unit.

Note that this implies that compiler directives can only appear between top-level Oz statements.

All compilation units in a file are processed sequentially.

2.1 Compiler Directives

`\switch` **('+'** ⟨switchname⟩ | **'-'** ⟨switchname⟩ **)***

Set ('+') resp. reset ('-') the specified switches. The values for ⟨switchname⟩ the compiler understands are described in Appendix A. Case matters.

`\pushSwitches`

Save the current settings of all switches onto the internal switch state stack.

`\popSwitches`

Restore all switch settings from the topmost element of the internal switch state stack, provided it is not empty, else do nothing.

`\localSwitches`

Save the current settings of all switches as well as the internal switch state stack if no `\localSwitches` has occurred in the current input before, else do nothing. They are automatically restored after the last compilation unit of the current input has been processed.

2.2 Macro Directives

`\line` ⟨filename⟩
`\line` ⟨int⟩ ⟨filename⟩

sets the internal file name (and the line number of the subsequent line) to the given values. This data will appear in the error messages and debug information generated by the compiler.

`\insert` ⟨filename⟩

is substituted by the contents of the referenced file. The `~user` syntax is supported under Unix for absolute file names. If the file name is relative, it is first resolved against the directory in which the file containing the `\insert` resides, then against all directories named in the `OZPATH` environment variable (which has standard `PATH` syntax).

Macro Names Note that although macro names have the same syntax as Oz variables, there is absolutely no connection between macro names and Oz variables.

`\define` ⟨variable⟩

adjoins ⟨variable⟩ to the set of macro names.

`\undef` ⟨variable⟩

removes ⟨variable⟩ from the set of macro names.

`\ifdef` ⟨variable⟩

causes the text until the next balanced `\else` or `\endif` only to be compiled if ⟨variable⟩ is a macro name.

`\ifndef` ⟨variable⟩

caused the text until the next balanced `\else` or `\endif` to be ignored if ⟨variable⟩ is a macro name.

`\else`

causes the text until the next balanced `\endif` to be

- ignored, if the `\else` occurs after a text region that was being compiled;
- compiled, if the `\else` occurs after a text region that was being ignored.

`\endif`

terminates the active `\ifdef` or `\ifndef`. Every `\ifdef` and every `\ifndef` must be terminated by a corresponding `\endif`.

3

Standard Applications

This chapter defines the standard applications of the compiler available with the Mozart system: the batch compiler, the compiler panel, and its relation to the OPI.

3.1 The Batch Compiler

The batch compiler `ozc` provides a command line interface to the compiler for batch compilation. Batch compilation means that expressions (typically functor definitions) are compiled and evaluated, and the resulting value is pickled.

Parameterization The command line options of `ozc` allow to customize the state of the compiler to use for compilation. Options allow for configuration of the set of defined macro names, to set all compiler switches, and to configure the compilation environment. Pickling can be parameterized to using compression or headers that make the resulting pickle executable. Further options influence the compilation mode: The default, evaluating an expression and pickling the result, can be replaced by evaluation of a statement, or intermediate results of compilation can be dumped. Finally, a special mode allows to compute dependencies (in the form of included source files).

Reference The exact specification of command line options can be found in Chapter *The Oz Compiler:* `ozc`*, (Oz Shell Utilities)*.

3.2 The Compiler Panel

The Compiler Panel is another application of the compiler. The compiler defines the notion of a compiler interface to communicate with the outside world. The compiler panel is one such interface.

Interfaces A compiler interface received notification about the state of the compiler. This includes its compilation state, but also its state of execution (how it manages its query queue, what errors it reports, and whether a compilation succeeds). Chapter 4 describes how the API allows to define interfaces.

The Panel The compiler panel is a compiler interface that provides a graphical display of the compiler's state. Its toplevel window has a menu that allows to enqueue further compilation tasks and to configure how the the compiler's state is displayed. Furthermore, it features a set of notebook tabs for the different parts of its state.

Tabs The `Messages` tab provides access to a text display of compilation phase tracking, and the warnings and errors reported. The `Environment` tab displays all variables in the compilation environment. Each identifier can be clicked to execute an action on its value; for instance, values can be shown or inspected. The set of actions is configurable. The `Switches` tab displays the settings of every compiler switch and option. Finally, the `Query Queue` tab displays the list of queries the compiler currently executes.

The Compiler Panel API

Like all tools, the compiler panel has an API to enable applications to control it. The module `CompilerPanel`, located at `x-oz://system/CompilerPanel`, exports a class `CompilerPanel.'class'` implementing the `Listener` interface, with the following methods.

`init`

```
init(+CompilerObject +IconifiedB <= false)
```

initializes a new instance of a compiler panel, associated with the compiler `CompilerObject`. If `IconifiedB` is **`true`**, it is started with its toplevel window retracted.

`close`

```
close()
```

closes a compiler panel instance, also closing its window.

`enqueue`

```
enqueue(+Message)
```

enqueues a query to the compiler associated with the compiler panel instance. See Chapter 4 for a specification of the values `Message` can take.

`addAction`

```
addAction(+NameV +ActionP)
```

adds an action to the menu of actions that can be performed on values in the environment. `NameV` is the name of the action as displayed in the menu while `ActionP` is a unary procedure which will be passed the value when invoked.

3.3 The Oz Programming Interface

The OPI (see *"The Oz Programming Interface"*) runs a dedicated compiler instance which receives all queries generated by feeding code in the OPI by default. It is available as `OPI.compiler`.

4

Application Programmer’s Interface

The compiler is available to Mozart applications through the module `Compiler`. This chapter describes the functionality provided by that module and its classes.

First, a number of additional secondary type names used in this description is introduced in Section 4.1, then the `Compiler` module is described in Section 4.2. The material in that section should prove sufficient for most cases. The remainder of the chapter is intended for advanced uses.

An arbitrary number of compilers may be instantiated, each with its own internal state, and used concurrently. We distinguish between compiler engines, described in Section 4.3, which store the state of a compiler and perform the compilation proper, and compiler interfaces, described in Section 4.4, which allow to observe the activities of compiler engines and to react to them. Both of these use the narrator/listener mechanism described in Appendix B; familiarity with this is assumed.

Finally, examples are presented in Section 4.5; in particular, the provided abstractions are explained in terms of compiler engines and interfaces.

4.1 Additional Secondary Types

This section describes additional secondary types used in the descriptions in this chapter. The conventions defined in Section *Description Format, (The Oz Base Environment)* will be respected.

`Coord`	stands for information about source coordinates. This is either **`unit`** if no information is known or a tuple `pos(FileName Line Column)`, where `FileName` is represented as an atom (`''` meaning ‘unknown’) and `Line` and `Column` are integers. Line numbering begins at `1` and column numbering at `0`; a column number of `~1` means ‘unknown’.
`SwitchName`	is an atom which must be a valid switch name (see Appendix A).
`PrintName`	is an atom which must be a valid variable print name.
`Env`	represents an environment, represented as a record whose features are valid print names.

4.2 The `Compiler` Module

`evalExpression`

`{Compiler.evalExpression +V +Env ?KillP X}`

evaluates an expression, given as a virtual string `V`, in a base environment enriched by the bindings given by `Env`, either returning the result `X` of the evaluation or raising an exception. Furthermore, the variable `KillP` is bound to a nullary procedure which, when applied, interrupts compilation.

`virtualStringToValue`

`{Compiler.virtualStringToValue +V X}`

is a replacement for `System.virtualStringToValue`, which was available in Mozart's predecessor DFKI Oz.

Note that you are discouraged from using this for large data structures: Because it is much more powerful than `System.virtualStringToValue`, it can also be much less efficient. Rather, you should use pickling and unpickling of data structures (see Chapter *Persistent Values:* `Pickle`, *(System Modules)*).

`engine`

`Compiler.engine`

is the final class from which compiler engines can be instantiated. This is described in detail in Section 4.3.

`interface`

`Compiler.interface`

is a class providing a simple mechanism to create compiler interfaces. It is described in detail in Section 4.4.

`parseOzFile`

`{Compiler.parseOzFile +V +O +P +Dictionary ?T}`

parses the Oz source file named `V`, returning an abstract syntax tree as defined in Appendix C in `T`. `O` is an instance of the `PrivateNarrator` class described in Appendix B; its methods are invoked for example to report compilation errors. `P` is a unary procedure expecting a switch name as described in Appendix A and returning a boolean value indicating the switch's state; in the current implementation, only the settings of `gump`, `allowdeprecated` and `showinsert` are requested. Finally, `Dictionary` is the set of macro names: The keys are defined macro names; its items should always be **`true`**. As a side-effect, `Dictionary` is modified according to `\define` and `\undef` macro directives.

`parseOzVirtualString`

`{Compiler.parseOzVirtualString +V +O +P +Dictionary ?T}`

is similar to `parseOzFile`, except that `V` denotes the source text itself instead of a source file name.

`assemble`

`{Compiler.assemble +Ts +Xs +SwitchR ?P ?V}` takes a list of bytecode instructions `Ts` for the Mozart virtual machine (see Appendix D), assembles them and returns the result in `P`, a nullary procedure which causes the code to be executed when applied. `Xs` is a list of global variables (the closure of `P`), the first element corresponding to register `g(0)`. `SwitchR` is a record whose features are switch names and whose values are booleans. In the current implementation, the switches `profile`, `controlflowinfo`, `verify`, and `peephole` are used. All features of `SwitchR` are optional (default values are substituted). `V` is a lazily computed virtual string containing an external representation of the assembled code after peephole optimization.

4.3 Compiler Engines

Instances of the `Compiler.engine` class are active objects called compiler engines. Each object's thread processes all queries inside its query queue sequentially.

The final class `Compiler.engine` inherits from `Narrator.'class'`, described in Appendix B.

4.3.1 Methods of the `Compiler.engine` Class

`enqueue`

```
enqueue(+T ?I <= _)
enqueue(+Ts ?Is <= _)
```

appends a new query `T` to the query queue. If `T` is an unknown query, an exception is raised immediately. All of the query's input arguments (the subtrees of `T`) are type-checked before it is enqueued.

Internally, each enqueued query is assigned a unique identification number `I`. This may be used later to remove the query from the queue (unless its execution has already begun or terminated).

The argument to enqueue may also be a *list* of queries: These are guaranteed to be executed in sequence without other queries interleaving. The second argument then returns a list of identification numbers.

`dequeue`

```
dequeue(+I)
```

dequeues the query with identification number `I`, if that query is still waiting in the query queue for execution, else does nothing.

`interrupt`

```
interrupt()
```

interrupts execution of the current query. Does not affect the remaining queue.

`clearQueue`

```
clearQueue()
```

flushes the whole remaining queue. Does not affect the currently processed query (if any).

4.3.2 Queries

This chapter documents the queries understood by the Mozart Compiler.

Some queries request state information from the compiler engine. The following description annotates the corresponding output variables with a question mark, although they only become bound when the query is actually executed. If binding an output variable raises an exception, an error is reported through the registered listeners (see Appendix B).

Macro Definitions

`macroDefine(+V)`

Add V to the set of defined macro names.

`macroUndef(+V)`

Remove V from the set of defined macro names.

`getDefines(?PrintNames)`

Return all currently defined macro names as a list, in no particular order.

Compiler Switches

`setSwitch(+SwitchName +B)`

Set the state of the given switch to either 'on', if `B == true`, or to 'off', if `B == false`.

`getSwitch(+SwitchName ?B)`

Return the state of the given switch.

`pushSwitches()`

Save the current settings of all switches onto the internal switch state stack.

`popSwitches()`

Restore all switch settings from the topmost element of the internal switch state stack, provided it is not empty, else do nothing.

Compiler Options

`setMaxNumberOfErrors(+I)`

Set the maximal number of errors to report for any one compilation before aborting it to `I`. A negative value means never to abort.

`getMaxNumberOfErrors(?I)`

Return the maximal number of errors to report for any one compilation before aborting it.

setBaseURL(+VU**)**

Set the base URL relative to which the **require** clause of computed functors is resolved. A value of **unit** means to resolve the imports relative to the location of the file in which the **functor** keyword appeared.

getBaseURL(?AU**)**

Return the base URL relative to which the **require** clause of computed functors is resolved.

setGumpDirectory(+VU**)**

Set the directory in which Gump output files are created. Can be relative. **unit** means the current working directory.

getGumpDirectory(?VU**)**

Return the directory in which Gump output files are created.

The Environment

addToEnv(+PrintName X**)**

Add a binding for a variable with the given print name, and bound to X, to the environment.

lookupInEnv(+PrintName X**)**

Look up the binding for the variable with the given print name in the environment and bind X to its value. If it does not exist, report an error.

removeFromEnv(+PrintName**)**

Remove the binding for the variable with the given print name from the environment if it exists.

putEnv(+Env**)**

Replace the current environment by the one given by Env.

mergeEnv(+Env**)**

Adjoin Env to the current environment, overwriting already existing bindings.

getEnv(?Env**)**

Return the current environment.

Feeding Source Text

feedVirtualString(+V**)**

Evaluate the Oz source code given by the virtual string V.

feedVirtualString(+V +R**)**

Evaluate the Oz source code given by the virtual string V, returning the resulting value in R.result (if the expression switch is set and R has the feature result).

feedFile(+V)

Evaluate the Oz source code contained in the file with name V.

feedFile(+V +R)

Evaluate the Oz source code contained in the file with name V, returning the resulting value in `R.result` (if the `expression` switch is set and R has the feature `result`).

Synchronization

ping(?U)

Bind the variable U to **unit** on execution of this query. This allows to synchronize on the work of the compiler, e.g., to be informed when a compilation is finished.

ping(?U X)

Works like the `ping(_)` query, except gives a value which will reappear in the response notification sent to interfaces. This allows to identify the `ping` query with its `pong` notification.

Custom Front-Ends

setFrontEnd(+ParseFileP +ParseVirtualStringP)

Replace the front-end used by the compiler by a custom front-end implemented by procedures `ParseFileP` and `ParseVirtualStringP`. These procedures have the same signature as `Compiler.parseOzFile` and `Compiler.parseOzVirtualString`, documented above. Indeed, these procedures implement the default Oz front-end.

4.4 Compiler Interfaces

As said above, compiler engines are narrators. The term 'compiler interface' simply denotes a standard listener attached to a compiler engine. This section presents what is required to implement a compiler interface.

First the notifications sent by compiler engines are documented. These include normal compiler output and information about compiler state changes. Then a specific compiler interface is described that makes many compilation tasks easy to control.

4.4.1 Sent Notifications

Query Queue

newQuery(I T)

A new query T with identification I has been enqueued.

runQuery(I T)

The query T with identification I is now being executed.

removeQuery(I)

The query with identification I has been removed from the query queue, either because it finished executing or because it was dequeued by a user program.

Compiler Activity

`busy()`

The compiler is currently busy (i.e., executing a query).

`idle()`

The compiler is currently idle (i.e., waiting for a query to be enqueued).

State Change

`switch(SwitchName B)`

The given switch has been set to `B`.

`switches(R)`

The settings of all switches is transmitted as a record mapping each switch name to its setting.

`maxNumberOfErrors(I)`

The maximum number of errors after which to abort compilation has been set to `I`.

`baseURL(AU)`

The base URL relative to which the `require` clause of computed functors is resolved has been set to `AU`.

`env(Env)`

The environment has been set to `Env`.

Output

`info(V)`

An information message `V` is to be printed out.

`info(V Coord)`

An information message `V`, related to the source coordinates `Coord`, is to be printed out.

`message(R Coord)`

An error or warning message `R`, related to the source coordinates `Coord`, is to be printed out. `R` has the standard error message format, described in Chapter *Error Formatting:* `Error`, *(System Modules)*.

`insert(V Coord)`

During parsing, the file named `V` has been read. The corresponding `\insert` directive (if any) was at source coordinates `Coord`.

`displaySource(TitleV ExtV V)`

A source text `V` with title `TitleV` is to be displayed; its format is the one for which the file extension `ExtV` is typically used (such as `oz` or `ozm`).

`attention()`

The error output buffer should be raised with the cursor at the current output coordinates (an error message should follow).

Synchronization

`pong(X)`

This is sent in response to a `ping(_)` or `ping(_ X)` query (see Section 4.3). In the first case, **`unit`** is returned in X.

4.4.2 The `Compiler.interface` Class

The `Compiler.interface` class is a subclass of the error listener class described in Appendix B. Its purpose is to provide a standard listener powerful enough to server many purposes, to spare the user of defining an own listener.

Methods In addition to the standard error listener interface, it supports the following methods.

`init(+EngineO +VerboseL <= false)`

initializes a new compiler interface, attaching it to the compiler engine `EngineO`. `VerboseL` can be one of **`true`**, **`false`**, or `auto`: If **`true`**, all messages, including the compiler's banner, will be output. If **`false`**, no messages will be output. If `auto`, the interface will remain silent unless an error of warning message arrives, in which case it will become verbose.

`sync()`

waits until the compiler engine becomes idle.

`getInsertedFiles(?Vs)`

returns a list of the file names that compilation has caused the inclusion of so far, in order of appearance.

`getSource(?V)`

returns the source that has last been displayed by the compiler (typically some intermediate representation), or the empty string if none.

`reset()`

clears the internal lists of inserted files and the displayed source.

`clear()`

is the same as `reset`.

4.5 Examples

4.5.1 Inspecting the Parse Tree

As a first example, here is an application that expects as single argument the name of an Oz source file. It parses the file, accepting Gump syntax, and displays the parse tree in the Inspector (or `parseError` if parsing failed). If there was a parse error, it extracts an error messages and prints that to standard output.

```
functor
import
   Narrator('class')
   ErrorListener('class')
   Application(getArgs)
   Compiler(parseOzFile)
   Inspector(inspect)
   System(printInfo)
define
   PrivateNarratorO
   NarratorO = {New Narrator.'class' init(?PrivateNarratorO)}
   ListenerO = {New ErrorListener.'class' init(NarratorO)}

   case {Application.getArgs plain} of [FileName] then
      {Inspector.inspect
       {Compiler.parseOzFile FileName PrivateNarratorO
        fun {$ Switch} Switch == gump end {NewDictionary}}}
      if {ListenerO hasErrors($)} then
         {System.printInfo {ListenerO getVS($)}}
         {ListenerO reset()}
      end
   end
end
```

4.5.2 A Look into the Provided Abstractions

The implementation of the `Compiler.evalExpression` procedure is a good example of how to use compiler engines and interfaces. `evalExpression` causes compilation of an expression within a specified environment. It is synchronous, i.e., only returns after the compilation has finished. Compiler error messages are raised as exceptions, and the compilation may be interrupted using the nullary procedure returned in `Kill`.

Startup Since we both want to control a compilation (done by a new compiler engine) and to observe the compilation process (to synchronize and to determine whether it produced errors), we first instantiate both an engine and an interface which we register with the engine. A number of queries are enqueued to the engine: We need to set the environment and appropriate compiler switches for compilation of an expression and to cause synchronous execution of the compiled program. When we're done configuring the compiler, we can start compilation of the source proper, expecting a result to be returned in variable `Result`.

Killing We then define the `Kill` procedure. The rest of the observation is performed in a new thread, because we want to kill the observation as well when `Kill` is invoked. `Kill` will clear any non-processed queries from the queue and interrupt the current one, then kill the observation thread (unless it had been already dead).

Observing Next we'll observe the running compiler, and for this we need to make use of the interface we created earlier. When the compiler becomes idle, we check whether it has output any error messages, in which case we record the faulty condition, else we report success. The main thread waits until the condition becomes known and reacts upon it.

```
proc {Compiler.evalExpression VS Env ?Kill ?Result} E I S in
   E = {New Compiler.engine init()}
   I = {New Compiler.interface init(E)}
   {E enqueue(mergeEnv(Env))}
   {E enqueue(setSwitch(expression true))}
   {E enqueue(setSwitch(threadedqueries false))}
   {E enqueue(feedVirtualString(VS return(result: ?Result)))}
   thread T in
      T = {Thread.this}
      proc {Kill}
         {E clearQueue()}
         {E interrupt()}
         try
            {Thread.terminate T}
            S = killed
         catch _ then skip   % already dead
         end
      end
      {I sync()}
      if {I hasErrors($)} then Ms in
         {I getMessages(?Ms)}
         S = error(compiler(evalExpression VS Ms))
      else
         S = success
      end
   end
   case S of error(M) then
      {Exception.raiseError M}
   [] success then skip
   [] killed then skip
   end
end
```

virtualStringToValue The `Compiler.virtualStringToValue` is trivial to implement on top of the functionality provided by `evalExpression`.

```
fun {Compiler.virtualStringToValue VS}
```

```
   {Compiler.evalExpression VS env() _}
end
```

A

Compiler Switches

This appendix describes the available boolean switches, giving their name, their default setting, and their effects on the compilation process if they are set.

Global Configuration

`compilerpasses` **(default:** `false`**)**

Output tracing information about the different phases the execution of each query proceeds through.

`showinsert` **(default:** `false`**)**

Show the names of files as they get inserted by the `\insert` macro directive, after their name has been resolved by means of the environment variable `OZPATH` (see Section 2.2).

`echoqueries` **(default:** `true`**)**

Output each (compilation) query verbatim.

`showdeclares` **(default:** `true`**)**

Summarize the variables declared by each query and thus added to the environment (provided the compilation succeeds and the compiled code is actually executed).

`watchdog` **(default:** `true`**)**

Terminate the current compilation if one of the compiler's threads blocks unexpectedly. This is useful for debugging the compiler.

Warnings

`warnredecl` **(default:** `false`**)**

Output warnings about redeclarations of top-level variables, either by `declare` or locally.

`warnshadow` **(default:** `false`**)**

Output warnings about redeclarations of all variables, either by `declare` or locally (subsumes `warnredecl`).

`warnunused` **(default: `true`)**

Output warnings about local variables never used or used only once (that is, initialized but never used again).

`warnunusedformals` **(default: `false`)**

Output the above warnings also for formal parameters. If `warnunused` is not set, this switch is ignored.

`warnforward` **(default: `false`)**

Warn about uses of features, attributes, or methods which are not known to be declared in the enclosing class.

`warnopt` **(default: `false`)**

Warn if a **`case`** conditional cannot be translated into optimized code. Also warn if a **`cond`** or disjunction's guard is translated with an explicit thread creation.

I. Parsing and Expanding

`unnest` **(default: `true`)**

Enable unnesting. If unnesting is disabled, none of the following passes are executed either.

`expression` **(default: `false`)**

Compile expressions, not statements. The result of an evaluated expression can be obtained through an output argument of the enqueued query (see Section 4.3) or the variable `‘result‘`.

`allowdeprecated` **(default: `true`)**

Allow the use of deprecated syntax, i.e., allow to use **`case`** for boolean conditionals and to mix **`if`** with **`elsecase`** and **`case`** with **`elseif`**.

`gump` **(default: `false`)**

Allow Gump definitions. If this switch is off, Gump keywords are parsed as ordinary atoms.

The remaining Gump-related switches are described in *“Gump–A Front-End Generator for Oz”*.

II. Static Analysis

`staticanalysis` **(default: `true`)**

Perform static analysis. Switching this off has severe impacts on optimization and error reporting.

III. Core Output

`core` **(default: `false`)**

Output the result of the core expansion of a query.

`realcore` **(default: `false`)**

Output the core expansion as it really is (and not beautified by, e.g., the use of operator symbols).

`debugvalue` **(default: `false`)**

Include annotations about the values propagated by static analysis.

`debugtype` **(default: `false`)**

Include annotations about the types inferred by static analysis.

IV. Code Generation

`codegen` **(default: `true`)**

Generate code.

`outputcode` **(default: `false`)**

Output the generated code as human-readable assembly code.

`recordhoist` **(default: `true`)**

Perform the record hoisting optimization.

V. Feeding to the Emulator

`feedtoemulator` **(default: `true`)**

Load the generated code into the emulator and execute it.

`threadedqueries` **(default: `true`)**

Execute each piece of generated code in a separate thread without waiting for it to terminate before proceeding to the next query.

`profile` **(default: `false`)**

Include profiling information in the generated code.

VI. Debugging

Most of these switches are used by the source level debugger, described in *“The Mozart Debugger”*.

`runwithdebugger` **(default: `false`)**

Place a static breakpoint on the first statement of each compiled query.

Note that the following switches severely impact code size and run-time efficiency of the generated code.

`controlflowinfo` **(default:** `false`**)**

Include program flow information in the generated code.

`staticvarnames` **(default:** `false`**)**

Include environment information in the generated code.

`dynamicvarnames` **(default:** `false`**)**

All created local variables are annotated with their print name at run time, such that `System.printName` applied to the variable's value returns this name.

B

Narrators and Listeners

This appendix documents some supporting classes required to observe a running compiler. These classes implement a general producer–consumer pattern for the status of a running computation and the error messages generated by it. They can be used independently of the compiler (actually, they are also used by the `ozdoc` tool that generated this documentation).

A narrator, presented in Section B.1, is a producer of status and error messages. Conversely, a Listener, presented in Section B.2, is a consumer of such messages. Each listener is connected to exactly one narrator; every narrator may have a number of listeners (you can observe this if you open compiler panels in the Oz Programming Interface).

Typically, the result (from, say, a compilation) one will want to programmatically check for is success or failure, and the list of error messages generated. An error listener provides easy access to precisely that information, and is presented in Section B.3.

B.1 The `Narrator` Module

The `Narrator` module, located at `x-oz://system/Narrator`, exports a class `Narrator.'class'` which is the superclass of all narrators. It manages a number of connected listeners (which are simply ports for the purpose of narrators) and provides for broadcasting messages to all registered listeners. The constructor returns a private narrator instance, used by the running computation to send well-defined status messages. Both of these classes are described in the following.

The class `Narrator.'class'`

Methods

`init(?PrivateNarratorO)`

initializes a narrator that initially has no listeners attached, and returns the corresponding private narrator.

`register(+Port)`

registers a new listener *Port*. All subsequently told messages will be forwarded to *Port*.

`newListener(+Port)`

is invoked on every new listener *Port* that is registered. The default action is a no-op; subclasses may override this for example to provide the new Listener with a summary of the current state (as a batch of messages).

`unregister(+Port)`

unregisters a listener *Port*. No more messages will subsequently be forwarded to *Port*.

`tell(X)`

broadcasts the message *X* to all registered listeners.

Private Narrators

Methods

`setLogPhases(+B)`

determines whether messages about phases of the underlying computations should be broadcasted to listeners (if *B* is `true`) or not (if *B* is `false`). The default is `false`.

`setMaxNumberOfErrors(+I)`

sets the number of error messages to report before interrupting the underlying computation with a `tooManyErrors` exception. The default is `17`. If *I* is less than or equal to zero, infinitely many errors are acceptable.

`tell(X)`

broadcasts the message *X* to all registered listeners.

`startBatch()`

resets internal state (in particular, the error count).

`startPhase(+V)`

broadcasts an `info(...)` message to all registered listeners, to the effect that phase `V` has started.

`startSubPhase(+V)`

broadcasts an `info(...)` message to all registered listeners, to the effect that subphase `V` has started.

`endBatch(+A)`

broadcasts messages to all registered listeners about the outcome of the underlying computation. *A* can be one of `accepted`, `rejected`, `aborted`, `crashed`, or `interrupted`. For any but `accepted` or `interrupted`, an `attention` message is broadcast. Causes an `info(...)` message to be broadcast.

`error(coord: +Coord <= unit    kind: +KindV <= unit    msg: +MsgV <= unit`

broadcasts a `message(error(...) ...)`. Raises `tooManyErrors` if the maximum allowed error count is exceeded.

`warn(coord: +Coord <= unit kind: +KindV <= unit msg: +MsgV <= unit ite`

broadcasts a `message(warn(...) ...)`.

`hasSeenError(?B)`

returns **true** iff an error message has been broadcast.

B.2 The `Listener` Module

The `Listener` module, located at `x-oz://system/Listener`, exports the class `Listener.'class'` with the following methods.

Methods

`init(+NarratorO +ServeL)`

initializes a listener with a narrator and the label of a unary method. The listener creates a port, registers this with `NarratorO`, and creates a thread in which the `ServeL` method is applied to the port's stream.

`close()`

undoes all effects of the `init` method: The server thread is terminated and the listener's port is unregistered.

`getNarrator(?NarratorO)`

returns the narrator with which the listener's port is currently registered.

`getPort(?Port)`

returns the associated port.

B.3 The `ErrorListener` Module

The `ErrorListener` module, located at `x-oz://system/ErrorListener`, exports the class `ErrorListener.'class'` with the following methods.

Methods

`init(+NarratorO +ServeOneL <= unit ?VerboseL <= false)`

initializes an error listener. If *ServeOneL* is different from **unit**, it should be the label of a method of a subclass. If an unrecognized message is received from the narrator, the method is invoked with the message as its single parameter. *VerboseL* can be one of **true** (print all received messages to standard error), **false** (print none of the messages), and `auto` (print all messages if some error message is among them).

`reset()`

is a no-op, intended to be overriden by derived classes. Invoked when a `close()` message is received.

`setVerbosity(+L)`

sets the verbosity to *L*, which can be one of **`true`** (print all received messages to standard error), **`false`** (print none of the messages), and `auto` (print all messages if some error message is among them).

`hasErrors(?B)`

returns **`true`** iff some error message was received since the last `close()` message.

`isActive(?B)`

returns **`true`** iff some error or warning message has been received since the last `close()` message.

`getVS(?V)`

returns the history of messages as a virtual string.

`getMessages(?Xs)`

returns the history of messages as a list.

`formatMessages(+Xs ?V)`

takes a list of messages *Xs* and returns them as a virtual string.

C

Syntax Tree Format

This appendix documents the syntax tree data structure used by the compiler. This information is only needed by implementors of custom front-ends. Most nodes are self-explanatory; if in doubt, it is recommended that you refer to the Gump sample implementing an Oz parser (installed at `examples/gump/OzParser.ozg`).

Input

```
⟨input⟩  ::=  parseError
           |  [⟨compilation unit⟩]
```

Compilation Units

```
⟨compilation unit⟩  ::=  ⟨phrase⟩
                      |  ⟨directive⟩
                      |  fDeclare(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)

⟨directive⟩  ::=  dirSwitch([⟨switch⟩])
               |  dirPushSwitches
               |  dirPopSwitches
               |  dirLocalSwitches

⟨switch⟩  ::=  on(⟨switch name⟩ ⟨coord⟩)
            |  off(⟨switch name⟩ ⟨coord⟩)

⟨switch name⟩  ::=  ⟨atom⟩
```

C.1 The Base Language

Phrases At the syntactical level, statements are not distinguished from expressions. Both are subsumed by ⟨phrase⟩. In a top-down analysis of the tree, it can be determined which phrases need to be statements and which need to be expressions. The `fStepPoint` form is only required if you want to provide support for source-level debugging: It wraps the contained phrase into a step point (see *"The Mozart Debugger"*); the atom can be freely chosen to indicate its kind (`call`, `conditional`, etc.).

```
⟨phrase⟩   ::=  fStepPoint(⟨phrase⟩ ⟨atom⟩ ⟨coord⟩)
             |  fAnd(⟨phrase⟩ ⟨phrase⟩)
             |  fEq(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fAssign(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fOrElse(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fAndThen(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fOpApply(⟨atom⟩ [⟨phrase⟩] ⟨coord⟩)
             |  fOpApplyStatement(⟨atom⟩ [⟨phrase⟩]
                                  ⟨coord⟩)
             |  fDotAssign(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fObjApply(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fAt(⟨phrase⟩ ⟨coord⟩)
             |  ⟨atom literal⟩
             |  ⟨escapable variable⟩
             |  ⟨wildcard⟩
             |  fSelf(⟨coord⟩)
             |  fDollar(⟨coord⟩)
             |  ⟨int literal⟩
             |  fFloat(⟨float⟩ ⟨coord⟩)
             |  fRecord(⟨label⟩ [⟨record argument⟩])
             |  fOpenRecord(⟨label⟩ [⟨record argument⟩])
             |  fApply(⟨phrase⟩ [⟨phrase⟩] ⟨coord⟩)
             |  fProc(⟨phrase⟩ [⟨phrase⟩] ⟨phrase⟩
                      [⟨proc flag⟩] ⟨coord⟩)
             |  fFun(⟨phrase⟩ [⟨phrase⟩] ⟨phrase⟩
                      [⟨proc flag⟩] ⟨coord⟩)
             |  fFunctor(⟨phrase⟩ [⟨functor descriptor⟩] ⟨coord⟩)
             |  fClass(⟨phrase⟩ [⟨class descriptor⟩]
                       [⟨meth⟩] ⟨coord⟩)
             |  fLocal(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fBoolCase(⟨phrase⟩ ⟨phrase⟩ ⟨opt else⟩ ⟨coord⟩)
             |  fCase(⟨phrase⟩ [⟨case clause⟩]
                      ⟨opt else⟩ ⟨coord⟩)
             |  fFOR([⟨for decl⟩] ⟨phrase⟩ ⟨coord⟩)
             |  fLockThen(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
             |  fLock(⟨phrase⟩ ⟨coord⟩)
             |  fThread(⟨phrase⟩ ⟨coord⟩)
             |  fTry(⟨phrase⟩ ⟨catch⟩ ⟨finally⟩ ⟨coord⟩)
             |  fRaise(⟨phrase⟩ ⟨coord⟩)
             |  fSkip(⟨coord⟩)

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

⟨atom literal⟩   ::=  fAtom(⟨literal⟩ ⟨coord⟩)

⟨variable⟩   ::=  fVar(⟨atom⟩ ⟨coord⟩)
```

```
⟨escapable variable⟩  ::=  ⟨variable⟩
                        |  fEscape(⟨variable⟩ ⟨coord⟩)

⟨wildcard⟩  ::=  fWildcard(⟨coord⟩)

⟨int literal⟩  ::=  fInt(⟨int⟩ ⟨coord⟩)

⟨record argument⟩  ::=  ⟨phrase⟩
                     |  fColon(⟨feature⟩ ⟨phrase⟩)
```

Procedures can carry flags (atoms following the **proc** or **fun** keyword). For the moment, the only recognized flags are `instantiate` (the body's code is copied upon application), `lazy` (the body has by-need semantics), `dynamic` (disable static-call optimization of this procedure), and `sited` (cannot be pickled). Other atoms are silently ignored.

```
⟨proc flag⟩  ::=  ⟨atom literal⟩
```

Functors

```
⟨functor descriptor⟩  ::=  fRequire([⟨import decl⟩] ⟨coord⟩)
                        |  fPrepare(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
                        |  fImport([⟨import decl⟩] ⟨coord⟩)
                        |  fExport([⟨export decl⟩] ⟨coord⟩)
                        |  fDefine(⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)

⟨import decl⟩  ::=  fImportItem(⟨variable⟩ [⟨aliased feature⟩]
                                ⟨opt import at⟩)

⟨aliased feature⟩  ::=  ⟨feature no var⟩
                     |  ⟨variable⟩#⟨feature no var⟩

⟨opt import at⟩  ::=  fNoImportAt
                   |  fImportAt(⟨atom literal⟩)

⟨export decl⟩  ::=  fExportItem(⟨export item⟩)

⟨export item⟩  ::=  ⟨variable⟩
                 |  fColon(⟨feature no var⟩ ⟨variable⟩)
```

Classes

```
⟨class descriptor⟩ ::= fFrom([⟨phrase⟩] ⟨coord⟩)
                     | fProp([⟨phrase⟩] ⟨coord⟩)
                     | fAttr([⟨attr or feat⟩] ⟨coord⟩)
                     | fFeat([⟨attr or feat⟩] ⟨coord⟩)

⟨attr or feat⟩ ::= ⟨escaped feature⟩
                 | ⟨escaped feature⟩#⟨phrase⟩

⟨meth⟩ ::= fMeth(⟨meth head⟩ ⟨phrase⟩ ⟨coord⟩)

⟨meth head⟩ ::= ⟨meth head 1⟩
              | fEq(⟨meth head 1⟩ ⟨variable⟩ ⟨coord⟩)

⟨meth head 1⟩ ::= ⟨atom literal⟩
                | ⟨escapable variable⟩
                | fRecord(⟨meth head label⟩ [⟨meth argument⟩])
                | fOpenRecord(⟨meth head label⟩ [⟨meth argument⟩])

⟨meth head label⟩ ::= ⟨atom literal⟩
                    | ⟨escapable variable⟩

⟨meth argument⟩ ::= fMethArg(⟨meth arg term⟩ ⟨default⟩)
                  | fMethColonArg(⟨feature⟩ ⟨meth arg term⟩ ⟨default⟩)

⟨meth arg term⟩ ::= ⟨variable⟩
                  | ⟨wildcard⟩
                  | fDollar(⟨coord⟩)

⟨default⟩ ::= fNoDefault
            | fDefault(⟨phrase⟩ ⟨coord⟩)
```

Features

```
⟨feature no var⟩ ::= ⟨atom literal⟩
                   | ⟨int literal⟩

⟨feature⟩ ::= ⟨feature no var⟩
            | ⟨variable⟩

⟨escaped feature⟩ ::= ⟨feature no var⟩
                    | ⟨escapable variable⟩
```

Other

```
⟨case clause⟩  ::=  fCaseClause(⟨pattern⟩ ⟨phrase⟩)

⟨pattern⟩  ::=  ⟨phrase⟩
            |   fSideCondition(⟨phrase⟩ ⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)

⟨catch⟩  ::=  fNoCatch
          |   fCatch([⟨case clause⟩] ⟨coord⟩)

⟨finally⟩  ::=  fNoFinally
            |   ⟨phrase⟩

⟨opt else⟩  ::=  fNoElse(⟨coord⟩)
             |   ⟨phrase⟩

⟨for decl⟩  ::=  forFeature(⟨atom literal⟩ ⟨phrase⟩)
             |   forPattern(⟨phrase⟩ ⟨for gen⟩)

⟨for gen⟩  ::=  forGeneratorList(⟨phrase⟩)
            |   forGeneratorInt(⟨phrase⟩ ⟨phrase⟩ ⟨opt phrase⟩)
            |   forGeneratorC(⟨phrase⟩ ⟨phrase⟩ ⟨opt phrase⟩)

⟨opt phrase⟩  ::=  ⟨phrase⟩
               |   unit
```

Coordinates Each triple consisting of an ⟨atom⟩ and two ⟨int⟩s denotes a file name ('' if none known), a line number (starting at `1`; required) and a column number (starting at `0`; `~1` if none known). If two triples are given, then they denote the starting and ending coordinates of a construct. A `pos` may be turned into a `fineStep` or a `coarseStep`, denoting a step point for debugging. `unit` is an unknown coordinate.

```
⟨coord⟩  ::=  pos(⟨atom⟩ ⟨int⟩ ⟨int⟩)
          |   pos(⟨atom⟩ ⟨int⟩ ⟨int⟩ ⟨atom⟩ ⟨int⟩ ⟨int⟩)
          |   fineStep(⟨atom⟩ ⟨int⟩ ⟨int⟩)
          |   fineStep(⟨atom⟩ ⟨int⟩ ⟨int⟩ ⟨atom⟩ ⟨int⟩ ⟨int⟩)
          |   coarseStep(⟨atom⟩ ⟨int⟩ ⟨int⟩)
          |   coarseStep(⟨atom⟩ ⟨int⟩ ⟨int⟩ ⟨atom⟩ ⟨int⟩ ⟨int⟩)
          |   unit
```

C.2 Finite Domain Extensions and Combinators

```
⟨phrase⟩  +=  ⟨fd expression⟩
           |  fFail(⟨coord⟩)
           |  fNot(⟨phrase⟩ ⟨coord⟩)
           |  fCond([⟨clause⟩] ⟨opt else⟩ ⟨coord⟩)
           |  fOr([⟨clause opt then⟩] ⟨coord⟩)
           |  fDis([⟨clause opt then⟩] ⟨coord⟩)
           |  fChoice([⟨phrase⟩] ⟨coord⟩)

⟨fd expression⟩  ::=  fFdCompare(⟨atom⟩ ⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)
                  |  fFdIn(⟨atom⟩ ⟨phrase⟩ ⟨phrase⟩ ⟨coord⟩)

⟨clause⟩  ::=  fClause(⟨phrase⟩ ⟨phrase⟩ ⟨phrase⟩)

⟨clause opt then⟩  ::=  fClause(⟨phrase⟩ ⟨phrase⟩ ⟨opt then⟩)

⟨opt then⟩  ::=  fNoThen(⟨coord⟩)
             |  ⟨phrase⟩
```

C.3 Gump Extensions

```
⟨compilation unit⟩  +=  fSynTopLevelProductionTemplates([⟨prod clause⟩])

⟨phrase⟩  +=  fScanner(⟨variable⟩
                       [⟨class descriptor⟩] [⟨meth⟩]
                       [⟨scanner rule⟩] ⟨atom⟩ ⟨coord⟩)
           |  fParser(⟨variable⟩
                      [⟨class descriptor⟩] [⟨meth⟩]
                      ⟨token clause⟩ [⟨parser descriptor⟩] ⟨int⟩
                      ⟨coord⟩)

⟨grammar symbol⟩  ::=  ⟨atom literal⟩
                   |  ⟨variable⟩
```

Scanners

```
⟨scanner rule⟩  ::=  fMode(⟨variable⟩ [⟨mode descriptor⟩])
                 |  ⟨lex clause⟩

⟨mode descriptor⟩  ::=  fInheritedModes([⟨variable⟩])
                    |  ⟨lex clause⟩

⟨lex clause⟩  ::=  fLexicalAbbreviation(⟨grammar symbol⟩ ⟨regex⟩)
               |  fLexicalRule(⟨regex⟩ ⟨phrase⟩)

⟨regex⟩  ::=  ⟨string⟩
```

Parsers

```
⟨token clause⟩   ::=  fToken([⟨token decl⟩])

⟨token decl⟩   ::=  ⟨atom literal⟩
                 |  ⟨atom literal⟩#⟨phrase⟩

⟨parser descriptor⟩   ::=  ⟨prod clause⟩
                        |  ⟨syntax rule⟩

⟨prod clause⟩   ::=  fProductionTemplate(⟨prod key⟩ [⟨prod param⟩]
                                         [⟨syntax rule⟩] [⟨syn expression⟩]
                                         [⟨prod ret⟩])

⟨prod param⟩   ::=  ⟨variable⟩
                 |  ⟨wildcard⟩

⟨prod key⟩   ::=  none#⟨string⟩
               |  ⟨atom⟩#⟨string⟩

⟨prod ret⟩   ::=  none
              |  ⟨variable⟩
              |  fDollar(⟨coord⟩)

⟨syntax rule⟩   ::=  fSyntaxRule(⟨grammar symbol⟩ [⟨syn formal⟩]
                                 ⟨syn expression⟩)

⟨syn formal⟩   ::=  ⟨variable⟩
                 |  ⟨wildcard⟩
                 |  fDollar(⟨coord⟩)

⟨syn expression⟩   ::=  fSynApplication(⟨grammar symbol⟩ [⟨phrase⟩])
                     |  fSynAction(⟨phrase⟩)
                     |  fSynSequence([⟨variable⟩] [⟨syn expression⟩])
                     |  fSynAlternative([⟨syn expression⟩])
                     |  fSynAssignment(⟨escapable variable⟩ ⟨syn expression⟩)
                     |  fSynTemplateInstantiation(⟨prod key⟩ [⟨syn expression⟩]
                                                  ⟨coord⟩)
```

D

Mozart Virtual Machine Bytecode

Index