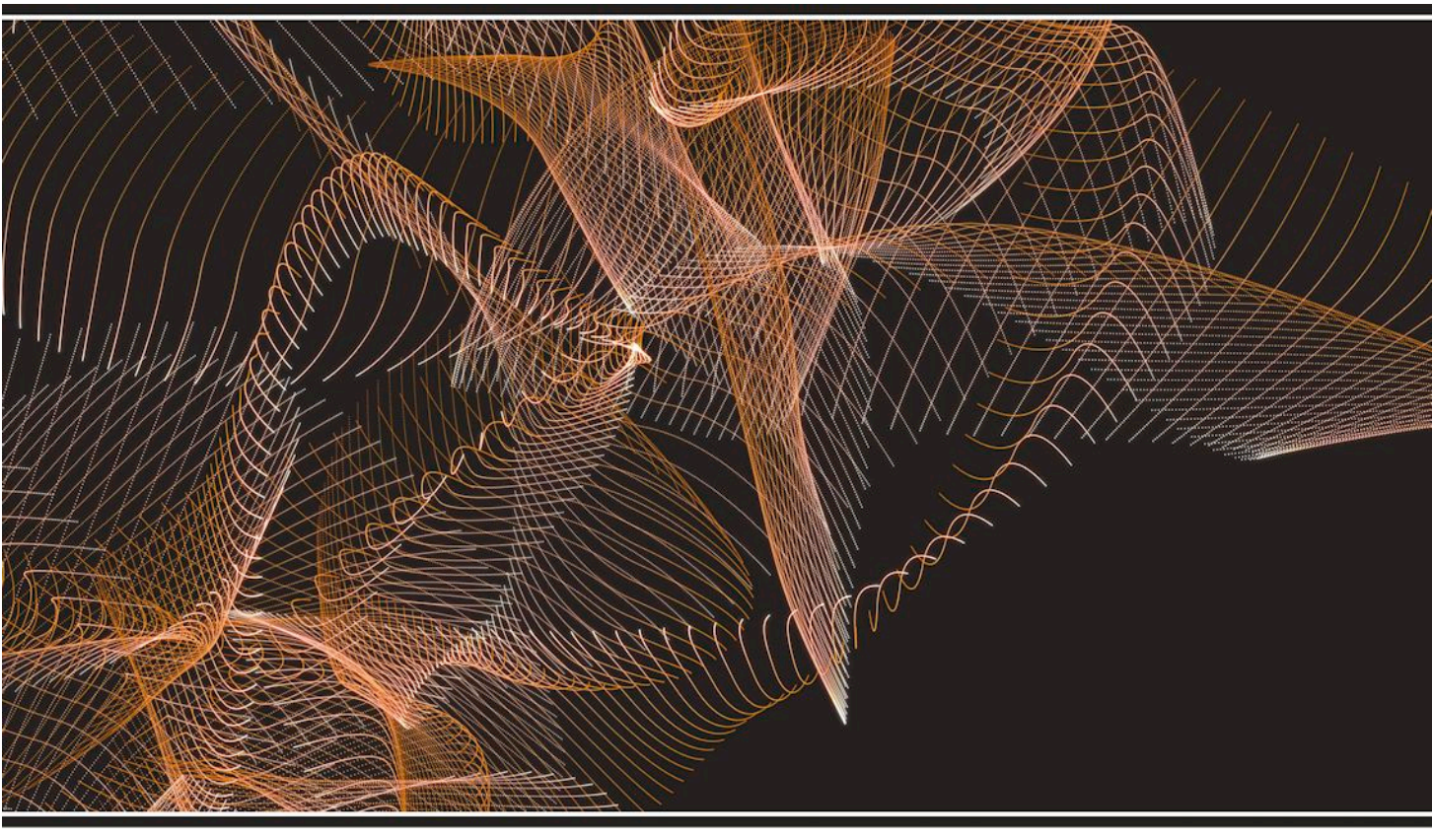
Compiler User Guide

Dyalog version 15.0



DYALOG

The tool of thought for software solutions

Dyalog is a trademark of Dyalog Limited
Copyright © 1982-2016 by Dyalog Limited
All rights reserved.

Compiler User Guide

Dyalog version 15.0
Document Revision: 20160608_150

No part of this publication may be reproduced in any form by any means without the prior written permission of Dyalog Limited.

Dyalog Limited makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Dyalog Limited reserves the right to revise this publication without notification.

email: support@dyalog.com
http://www.dyalog.com

TRADEMARKS:

SQAPL is copyright of Insight Systems ApS.
Array Editor is copyright of davidliebtag.com
Raspberry Pi is a trademark of the Raspberry Pi Foundation.
Oracle® and Java™ are registered trademarks of Oracle and/or its affiliates.
UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.
Windows® is a registered trademark of Microsoft Corporation in the United States and other countries.
Mac OS® and OS X® (operating system software) are trademarks of Apple Inc., registered in the U.S. and other countries.

All other trademarks and copyrights are acknowledged.

Contents

1 About This Document

This document is intended as an introduction to the compiler that was introduced with Dyalog version 14.0 for the purpose of improving the performance of defined functions and operators.

1.1 Audience

It is assumed that the reader has a reasonable understanding of Dyalog.

For information on the resources available to help develop your Dyalog knowledge, see http://www.dyalog.com/introduction.htm.

1.2 Conventions

Unless explicitly stated otherwise, all examples in Dyalog documentation assume that ⎕IO and ⎕ML are both 1.

Various icons are used in this document to emphasise specific material.

General note icons, and the type of material that they are used to emphasise, include:


Hints, tips, best practice and recommendations from Dyalog Ltd.


Information note highlighting material of particular significance or relevance.


Legacy information pertaining to behaviour in earlier releases of Dyalog or to functionality that still exists but has been superseded and is no longer recommended.


Warnings about actions that can impact the behaviour of Dyalog or have unforeseen consequences.

Although the Dyalog programming language is identical on all platforms, differences do exist in the way some functionality is implemented and in the tools and interfaces that are available. A full list of the platforms on which Dyalog version 15.0 is supported is available at www.dyalog.com/dyalog/current-platforms.htm. Within this document, differences in behaviour between operating systems are identified with the following icons (representing Mac OS, Linux, UNIX and Microsoft Windows respectively):

2 Introduction

When the APL interpreter executes a user-defined function, it spends most of its time performing two separate actions:

- Parsing the APL syntax ("interpreter overhead")
- Executing individual primitive functions

The compiler is designed to reduce the time spent on the first of these two actions, the interpreter overhead, by converting the APL source code into a bytecode form that is more efficient to execute.

The biggest performance gains are achieved when the compiler is used on functions that are applied to simple scalars or small array arguments. If the arguments are large arrays, then the interpreter spends most of its time executing the primitive functions; this means that the benefit of reducing the interpreter overhead is less significant.

2.1 Optimisations

In addition to reducing interpreter overhead, the compiler can also perform certain optimisations on the APL code. These include:

- constant folding (see *Section 2.1.1*)
- eliminating local names (see *Section 2.1.2*)
- flexible idiom recognition (see *Section 2.1.3*)

2.1.1 Constant Folding

When a primitive function is applied to constant arguments, the compiler attempts to evaluate the entire expression at compile time, thereby saving time when the function is executed. For example:

```
encode←{(⎕A,⎕D)⍳⍵}    ⍝ ⎕A,⎕D is evaluated at compile time
```

However, the compiler cannot always successfully evaluate every expression. For example, the compiler cannot evaluate primitive functions that depend on system variables:

```
numbers←{⍳7} ⍝ compiler cannot evaluate ⍳7 as it does not
             ⍝ know what value ⎕IO will have
```

Constants will only be retained if they are reasonably small. The limit is typically around 1,000 items for a simple array.

2.1.2 Eliminating Local Names

Every assignment to a local name incurs a measurable overhead, especially within a dfn. The compiler discards all local names as part of its normal operation, so this overhead is eliminated in compiled code.

2.1.3 Flexible Idiom Recognition

The Dyalog interpreter recognises idioms as specific sequences of characters, for example, `0=⍴⍴x`. The compiler recognises the same idioms but in a more flexible way, enabling it to cope with syntactic variations. This means that expressions can be identified as idioms (and processed as such) even if:

- parts of the expression are named (as long as there are no other uses of the same name), for example, `s←⍴x ⋄ 0=⍴s`
- redundant parentheses are added, for example, `0=(⍴⍴x)`
- the arguments to commutative functions are swapped, for example, `(⍴⍴x)=0`

In these situations, the compiler's optimisations transform the expression into one that matches an idiom. For example, `(≠⍵)=⍴⍴x` is recognised as being the same as the idiom `0=⍴⍴x` because the expression `≠⍵` is evaluated to 0 at compile time.

2.2 Future Work

Over the next several releases, the compiler will be enhanced to include additional optimisations. Planned extensions include:

- common sub-expression elimination (see *Section 2.2.1*)
- loop folding (see *Section 2.2.2*)

2.2.1 Common Sub-expression Elimination

The compiler should be able to combine duplicated expressions. For example:

```
foo←{(⍵+1)+÷⍵+1}
```

should be combined into the equivalent of this:

```
foo←{
    t←⍵+1
    t+÷t
}
```

except without any overhead for introducing the local name (see *Section 2.1.2*).

2.2.2 Loop Folding

The compiler should allow more efficient execution of expressions such as `A+B×C` where the arguments are large arrays. Currently the interpreter has to evaluate `B×C` first, generating a large temporary result `T`, then evaluate `A+T` and finally discard `T`. The compiler should recognise the whole of the expression `A+B×C` before it starts evaluating it, which will enable a more efficient execution that does not have to generate a large temporary result.

2.3 Changes to Behaviour of Functions when Compiled

The same run-time engine is used by both compiled functions and interpreted functions when executing primitive functions. However, a small number of behavioural changes occur when functions are compiled.

2.3.1 Thread Switching

Thread switching will not occur between lines of code after a function has been compiled. However, it can still occur at the start of the function before the first line is executed.

2.3.2 Error Trapping

Compiled functions cannot be suspended. Errors occurring within compiled functions are signalled back to the calling environment (in the same way as if `⎕SIGNAL` had been used inside the function).

Similarly, when an error in a compiled function is handled by an Execute trap, the APL expression specified in the trap will be executed in the calling environment and will not be able to see any of the compiled function's local names.

2.3.3 Visible Names

When a user-defined operator is compiled, its local names are eliminated. As a result, the names are no longer visible to any sub-functions that it calls

2.4 Upgrading from Previous Versions of Dyalog

The format of the bytecode used for compiled functions in Dyalog version 15.0 is not compatible with the bytecode used for compiled functions in previous versions of Dyalog. When loading a workspace containing compiled functions that was saved by a previous version of Dyalog, all bytecode for compiled functions will be discarded and you must use `400⌶` to recompile them (see *Chapter 7*).

3 Basic Usage

Theoretically (see *Chapter 6*), every defined function in a workspace can have a compiled bytecode form. This bytecode is saved and loaded as part of the workspace, and will be copied along with the function on ⎕CY or)COPY or the ⎕OR of a compiled function.

To query whether a function `foo` has been successfully compiled, enter:

```
1(400⌶)'foo'
```

This returns a Boolean value of 1 if the compilation has been performed.

To compile a function foo, enter:

```
2(400⌶)'foo'
```

This returns a matrix of diagnostic information. If the matrix has zero rows then the function was compiled successfully. Otherwise, each row of the matrix describes a problem that prevented the compiler from compiling the function.

When a function is executed, Dyalog automatically executes any compiled code for the function. If none is available, then the function is executed using the traditional APL parser.

In summary:

```
⎕FX'r←foo y' ...  ⍝ define foo
foo 99            ⍝ execute the uncompiled code
2(400⌶)'foo'      ⍝ compile it
foo 99            ⍝ execute the compiled code
```

For full details on the syntax of `400⌶`, see *Chapter 7*.

3.1 User Interface

This only applies to Dyalog on the Microsoft Windows operating system.

Not every defined function or operator will compile successfully; this could be due to restrictions inherent within the compiler (see *Chapter 6*). Some of these might be unavoidable, but some could be avoided by recoding using a different approach. To make an informed decision, it is necessary to identify the lines of code within the function that cannot be compiled.

To identify lines of code that cannot be compiled

1. In the **Edit** window's menu bar, select **View** > **Compiler Errors**.

   An additional column is displayed to the left of the source code. If a line of code cannot be compiled, a vertical red bar will be present in this column next to the that line of code.
2. Move the cursor over a red bar.

   A pop-up message is displayed explaining why the line cannot be compiled.

Once the reason for not being able to compile a function is known, a decision can be made as to whether it should be amended to meet the compiler's requirements.

4 Compiling With Global Names

When compiling a defined function or operator, the compiler needs to know the nameclass of every name that is used. It is useful to distinguish between *local* names (those that are defined in the function or operator being compiled) and non-local or *global* names (everything else).

```
foo←{
    n←⍵×2              ⍝ define local name n
    bar n              ⍝ use global name bar and local name n
}
```

If this fuction is compiled with `2(400I) 'foo'`, the compiler will determine the nameclass of global names by looking at the names that are currently defined in the workspace:

- If `bar` is undefined, then the compiler will not compile `foo`.
- If `bar` is defined, then the compiler will use its nameclass to determine whether it is an array, function or operator, and parse the body of `foo` accordingly.

In the latter case, the nameclass of `bar` is recorded in the compiled form of `foo` as a checked assumption; when `foo` is executed, if `bar` no longer exists or has a different nameclass, an error will be reported.

In more complex applications there could be a requirement to restrict the set of global functions and variables that compiled code can refer to, or it might be known in advance exactly which global variables and functions will exist when the application is run. In these situations, the application can be compiled with `N(400I)`, where `N` is a namespace containing callback functions that the compiler can use to determine the nameclass of any global names it encounters.

For example, to mimic the behaviour of `2(400I)`, the following callback functions can be defined in `#` (the root namespace):

```
quadNC←⎕NC ⋄ quadAT←⎕AT      ⍝ define callback fns in #
#(400I)'foo'                 ⍝ pass in # as the namespace
```

More complicated definitions of the callback functions grant finer control over exactly which global names a compiled function is allowed to refer to (see *Chapter 7*).

5 Compiling Operators

When compiling a defined function or operator, the compiler needs to know the nameclass of every name that is used. This presents a problem for defined operators, because the nameclass of the operands is not known:

```
op←{
    ⍺⍺ / ⍵        ⍝ ⍺⍺ could be an array or a function
}
```

When an operator is compiled using `2(400⌶)Y` (see *Section 7.3*), the compiler assumes that the operands are functions. If the compiled operator is subsequently called with an array operand, then the compiled version is not used and the interpreter uses the parser instead (this restriction is expected to be removed in the future, when it should be possible to declare the types of names used by a function or operator).

To work around this, the compiler can be run in a mode where it will attempt to compile a defined operator the first time it is applied to some arguments; at this point the compiler can see exactly what the operands are. If the compilation is successful, then the compiler will record the nameclass of the operands along with the compiled bytecode. When the operator is applied again, the compiled bytecode will only be executed if the operands still have the same nameclass as they did the first time the operator was applied (if the nameclass of an operand has changed, then the compiled bytecode will not be used and the operator will be interpreted).

Continuing the example above:

```
      400⌶2       ⍝ enable auto compilation of operators

      +op 1 2 3 4 ⍝ op is compiled assuming fn operand
10

      400⌶0       ⍝ disable auto compilation
      ×op 1 2 3 4 ⍝ execute compiled bytecode again
24

      1 2 3 4 op 1 2 3 4 ⍝ operand is array; revert to interpreter
1 2 2 3 3 3 4 4 4 4
```

6 Restrictions

There are several restrictions when using the compiler, some of which will be removed in later versions.

RESTRICTION 1
A function that uses semi-global names cannot be compiled.

To compile a function, the compiler needs to be able to determine the nameclass of every name used in that function so that it knows whether it refers to an array, a function or an operator.

For local names, the compiler can identify the nameclass because it can see the definition of the name:

```
sum←{
    f←+             ⍝ compiler sees this definition...
    f/⍵             ⍝ ... so knows that f is a function here
}
```

For global names, callbacks from `400⌶` enable the compiler to identify the nameclass (see *Section 7.6*).

However, for semi-global names (that is, names that are local to the function that calls the function to be compiled) the compiler cannot determine the nameclass:

```
∇ r←sum y           ⍝ if f is defined in sum's caller, then...
  r←f/y             ⍝ ...this could be +/y, or 2/y, etc.
∇
```

RESTRICTION 2
A function that calls system functions which refer to values by name or create new named values cannot be compiled.

Compiled functions can use local names but, as part of the compilation process, the compiler discards these names, so they do not appear in the compiled bytecode. For this reason, system functions that refer to values by name, or create new named values, are prohibited:

```
foo←{
    a←⍺+⍵           ⍝ local name 'a' is discarded by compiler
    r←⎕NL 2         ⍝ ⎕NL is prohibited as it needs to see 'a'
    'a'⎕NS''        ⍝ ⎕NS is prohibited as it redefines 'a'
}
```

RESTRICTION 3

A function that includes the Execute function (⍎) cannot be compiled.

The compiler prohibits the use of Execute (⍎) because it could have arbitrary side effects unknown to the compiler.

RESTRICTION 4

A function that uses the dot syntax for namespace references cannot be compiled.

The compiler cannot currently determine the nameclass of a name when the dot syntax is used to refer to names inside arbitrary namespaces:

```
sum←{
    ⍺.f / ⍵         ⍝ ⍺.f could be an array or a function
}

prod←{
    #.util.prod ⍵   ⍝ nameclass of util and prod unknown
}
```

RESTRICTION 5

A function that includes certain control structures cannot be compiled.

The following control structures prevent a function from being compiled:

- `:Trap`
- `:Hold`
- `:With`
- `:Disposable`

RESTRICTION 6
A function cannot be compiled if it includes certain language features.

The following language features prevent a function from being compiled:

- dfn error guards
- localised ⎕TRAP
- function trains

6.1 Summary

A function cannot be compiled if it:

- uses semi-global names
- uses the dot syntax between user-defined names
- calls a system function that refers to values by name or creates new named values
- includes the Execute function (⍎)
- includes the control structures `Trap`, `:Hold`, `:With` or `:Disposable`
- includes dfn error guards or localises ⎕TRAP
- includes function trains

7 Language Reference

The compiler is controlled with ⍎400⌶. The syntax for this I-beam is:

```
R←{X}(400⌶)Y
```

In this syntax, X must be one of the following:

- 0 – Set automatic compilation options (default)
- 1 – Determine whether the function/operator Y has been successfully compiled
- 2 – Compile the function/operator Y
- 3 – Uncompile the function/operator Y
- 4 – Show bytecode for the compiled function/operator Y
- A namespace – Compile the function/operator Y using callbacks to provide information about global names

The nature of Y and R depend on the value of X.

7.1 Control Automatic Compilation (X = 0)

```
R←0(400⌶)Y ⍝ control automatic compilation of fns
```

When X is 0, Y must be one of the following values:

- 0 – Disable automatic compilation (initial setting)
- 1 – Compile functions when they are fixed (with ⎕FX or from the function editor)
- 2 – Compile operators the first time they are executed
- 3 – Compile functions when they are fixed (with ⎕FX or from the function editor) and compile operators the first time they are executed

The result R is the previous value of Y.

The automatic compilation setting is maintained within the workspace, and is saved and loaded with the workspace.

7.2 Query Compilation State (X = 1)

```
R←1(400⌶)Y ⍝ query compilation of function/operator Y
```

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names.

The result R is a Boolean scalar or vector, with a value of 1 if the function/operator has been successfully compiled and a value of 0 if it has not.

7.3 Compile (X = 2)

```
R←2(400⌶)Y ⍝ compile function/operator(s) Y
```

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names that should be compiled.

The result R is a matrix of diagnostic information or, if Y was either a matrix or a vector of vectors, a vector of such matrices. Each row of the matrix describes a problem that caused the compilation to fail, with four columns corresponding to:

- the APL error number
- the line number in the function/operator
- the column number (currently always 0)
- the error message

If the matrix has zero rows then the compilation was successful.

If this mechanism is used to compile operators, then the compiled bytecode will assume that the operator's operands are functions rather than arrays. At run time, the operands will be checked – if they are functions then the compiled bytecode will be used, otherwise the operator will be interpreted.

7.4 Discard Compiled Form (X = 3)

```
R←3(400⌶)Y ⍝ uncompile function/operator(s) Y
```

If Y is empty, then discard any compiled bytecode for all functions and operators in the workspace.

If Y is a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names, then discard any compiled bytecode for them.

R is always 0.

7.5 Show Bytecode (X = 4)

```
R←4(400⌶)Y ⍝ show bytecode for function/operator(s) Y
```

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names.

The result R is a multi-line string (that is, a character vector with embedded newlines) or, if Y was either a matrix or a vector of vectors, a vector of such strings. Each string is a human-readable representation of the bytecode of a compiled function or operator.

This functionality is provided for information and diagnostic purposes only. The human-readable form of the bytecode is subject to change at any time.

7.6 Compile with Callbacks (X is a Namespace)

```
R←N(400⌶)Y ⍝ compile function/operator(s) Y
```

Y must be a character vector, matrix or vector of vectors specifying the name of a function or operator or a list of such names. The specified functions or operators are compiled in the same way as when X = 2 (see *Section 7.4*) except that the compiler uses the namespace N to obtain information about global names.

The namespace N can contain any or all of following callback functions:

- `N.quadNC` – analogous to the system function ⎕NC. When applied monadically to an enclosed character vector it returns the detailed nameclass of that name. For example, given the name of a global dfn it returns the value 3.2.
- `N.quadAT` – analogous to the system function ⎕AT. When applied monadically to an enclosed character vector it returns a 1 by 4 matrix whose first item is a vector of 3 integers describing (respectively) the result, function valence and operator valence of the name.
- `N.getValue` – used to obtain the name of a global constant. When applied monadically to a character vector that is a global constant it returns the enclose of the constant value, otherwise it returns ⍬.

Each of these callback functions returns information about names that should be guaranteed to exist when the compiled functions are executed. The compiler assumes that the information returned by the callbacks is correct, and generates bytecode accordingly. In the case of `quadNC` and `quadAT`, if the information returned by the callbacks turns out not to be correct when the compiled function is executed, then a runtime error is generated.

The result R is a matrix of diagnostic information or, if Y was either a matrix or a vector of vectors, a vector of such matrices. Each row of the matrix describes a problem that caused the compilation to fail, with four columns corresponding to:

- the APL error number
- the line number in the function/operator
- the column number (currently always 0)
- the error message

If the matrix has zero rows then the compilation was successful.

Index