DYALOG

The tool of thought for expert programming

Dyalog
Experimental Functionality
Compiler

Version 14.0

Dyalog Limited

Minchens Court, Minchens Lane
Bramley, Hampshire
RG26 5BH
United Kingdom

tel: +44(0)1256 830030
fax: +44 (0)1256 830031
email: support@dyalog.com
http://www.dyalog.com

Version 14.0

Revision: 20140624_140

Contents

1 About This Document

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

The functionality and behaviour of the compiler has not yet been finalised and is subject to change by Dyalog Ltd. at any time.

The compiler should be considered experimental; users are encouraged to experiment with it and provide feedback to Dyalog Ltd. regarding the benefits and shortcomings of the current features to help prioritise further work. Users should be aware that the features of the compiler may change significantly based on feedback.

1.1 Audience

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

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.

At this release, the compiler is restricted to working only on functions and operators that are written in a purely functional style; that is, functions and operators that:

- do not rely on any global or semi-global variables
- do not have any side effects.
- apply some calculations to their arguments and return a result

While dfns encourage this style of programming, it is also possible to write purely functional traditional functions.

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. In Dyalog version 14.0 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
```

NOTE: 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

In Dyalog version 14.0, 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.

3 Basic Usage

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 *6*.

4 Restrictions

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

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

To compile a function, the compiler needs to be able to determine the name class 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 work this out, because it can see the definition of the name:

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

However, for global names the compiler does not look beyond the definition of the function currently being compiled, so it cannot determine their name class:

```
sum←{
    f/⍵      ⍝ could be +/⍵, or 2/⍵, etc.
}
```

RESTRICTION 2:
A function that uses the dot syntax for namespace references cannot be compiled.

The compiler cannot currently determine the name class 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
}
```

RESTRICTION 3:
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 4:
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 5:
A function that includes control structures cannot be compiled.

The initial release of the compiler is targeted towards dfns, so support for tradfns is limited. In particular, control structures cannot be compiled:

```
∇ r←foo y
  r←3
  :If y=0     ⍝ prohibited
      r←9
  :End
∇
```

RESTRICTION 6:
A function cannot be compiled if it includes any of the following language features:

- *indexed assignment*
- *selective assignment*
- *dfn error guards*
- *function trains*

4.1 Summary

A function cannot be compiled if it:

- uses global or 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 control structures
- includes indexed/selective assignment
- includes dfn error guards or localises ⎕TRAP
- includes function trains

5 Compiling Operators

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

```
op←{
    αα / ω        ⍝ αα could be an array or a function
}
```

When an operator is compiled using `2(400⌶)Y`, 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 name class 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 name class as they did the first time the operator was applied (if the name class 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; fall back
                                        to interpreter
1 2 2 3 3 3 4 4 4 4
```

6 Language Reference – 400⌶

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 values:

- 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

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

6.1 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.

6.2 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.

6.3 Compile (X = 2)

```
R←2(400I)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.

6.4 Discard Compiled Form (X = 3)

```
R←3(400I)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.

Appendix A Examples

```
      iscompiled←1⊃(400⌶)  ⍝ name some useful I-beams
        compile ←2⊃(400⌶)
      uncompile ←3⊃(400⌶)
```

A.1 Basic Usage

```
      mean←{(+⌿⍵)÷≢⍵}      ⍝ define a simple function
      mean 1 2 3 4         ⍝   and test it
2.5

      iscompiled'mean'     ⍝ it's not compiled yet
0

      compile'mean'        ⍝ compile it
      iscompiled'mean'     ⍝ (yes, that worked)
1

      mean 1 2 3 4         ⍝ check it still works
2.5
```

A.2 Speed Tests

```
      )copy dfns           ⍝ load some useful functions
dfns saved Mon Apr 28 08:55:24 2014

      ⎕CR'easter'
easter←{                   ⍝ Easter Sunday in year ⍵
    G←1+19|⍵               ⍝ year "golden number"
    C←1+⌊⍵÷100             ⍝ Century: eg 1984 → 20th

    X←¯12+⌊C×3÷4           ⍝ yrs in which leap yr omitted
    Z←¯5+⌊(5+8×C)÷25       ⍝ synch Easter & moon's orbit

    S←(⌊(5×⍵)÷4)-X+10      ⍝ find Sunday
    E←30|(11×G)+20+Z-X     ⍝ Epact
    F←E+(E=24)∨(E=25)∧G>11 ⍝    (when full moon occurs)

    N←(30×F>23)+44-F       ⍝ find full moon
    N←N+7-7|S+N            ⍝ advance to Sunday

    M←3+N>31               ⍝ month: March or April
    D←N-31×N>31            ⍝ day within month
    ↑10000 100 1+.×⍵ M D   ⍝ yyyymmdd

}

      cmpx'easter 2014'    ⍝ run time of 6.6 µs...
6.6E¯6
```

```
      compile'easter'      ⍝ ...is reduced by compiler...
      cmpx'easter 2014'    ⍝ ...to 4.3 µs
4.3E¯6
```