# ESPRIT PROJECT 6062

## Prototype Dylan Producer


**Written by:** Tony Mann, Rod Moyse,
Simon Green, Robert Mathews
Harlequin Ltd. _____

**Issued by:** Jorgen Bundgaard _____
DDC International

**Delivered by:** Gianluigi Castelli _____
Project Director


*The status of this document is "WORKING" if signed only by its author(s); it is "ISSUED" if signed by the Workpackage Manager; it is "DELIVERED" if signed by the Project Director.*


**Project deliverable id:** TR4.2.4-01

**Document code:** Harlequin GLUE4.2.4 - 01

**Date of first issue:** 1995-10-31
**Date of last issue:** 1995-10-31

**Availability:** Confidential




**omi/glue**

## CHANGE HISTORY

**This is the first version.**

# Contents

# 1. Purpose

This deliverable and its associated software demonstration represent the culmination of Harlequin's efforts in Workpackage 4 of the GLUE project. The original goal was to establish the efficacy of ANDF's support for the compilation of advanced languages by developing a Dylan producer. This includes the development of the appropriate load-time and runtime support for advanced languages based on ANDF hooks. Dylan is an advanced, modern, dynamically typed, object-oriented programming language which contains features not found in most other languages, such as automatic memory management and an exception mechanism. The language is still under development, although most of the design work that remains concerns the standard language libraries rather than the semantics of the language itself.

This task was tackled in four stages. In the first stage Harlequin used their background in existing implementations of development environments to identify the features which require support from ANDF to allow both the efficient compilation of Dylan and the efficient implementation of its runtime system. The second stage took the results of this work and began the incremental development of a Dylan producer. In the first instance, a small subset of Dylan was implemented, leading to an initial evaluation of ANDF's ability to support the relevant language features and runtime system.

The third stage focused on inter-language working. Software components written in ANDF-supported languages were combined with Dylan producer output, with the aim of ensuring that no performance penalty was imposed on the other ANDF-based components.

This deliverable reports the work of the fourth stage, the incremental extension of the implementation towards a prototype Dylan system. The incremental approach allowed the impact of various language features on the runtime to be studied, and also allowed the controlled introduction of a range of optimisation techniques used by Harlequin.

This work was sponsored by the Commission of the European Communities.

# 2. Executive Summary

## 2.1 Overview of the Dylan Producer

An important design goal for Dylan has been to provide a degree of algorithmic interchangeability in certain important subsystems of the implementation such as the garbage collector, the function calling mechanism, and the manipulation of tagged data. The abstraction of these important subsystems allows rapid experimentation with different implementations for these systems. ANDF link and install times are significantly lower than compilation times for real programs such as the Dylan compiler itself. The development and evaluation of new language implementation strategies can be hampered by the time taken to recompile the compiler, language test suites, and bench-marking programs. By moving as many of the details of the implementation as possible into the Dylan token library, the "edit-compile-evaluate" cycle for experimentation with implementation techniques has been significantly shortened. Changes to token definitions become effective as soon as the producer output is re-installed, and the previously-produced test-suite and benchmark programs need only be re-installed with the modified token library before the new implementation strategy can be evaluated.

The Dylan token library has been carefully designed to support the varying combinations of runtime activity described above, while keeping the externally visible set of tokens simple. This is desirable as a simpler token library helps to keep the compiler back-end relatively simple: complexity is moved out of the compiler and into the token library. A manageable token library also provides the possibility of future extensibility without the need for extensive modification to the compiler. In the ideal case, only the definitions in the token library would need to be changed.

The following six chapters give the details of this work with relevant conclusions.

## 2.2 Chapter Outline

### 2.2.1 Chapter 3. The Dylan Implementation Model

This chapter explains the techniques which are used by the producer to implement the dynamic features of Dylan.

### 2.2.2 Chapter 4. The Dylan Producer Representation

This chapter gives the details of the representations used in the Dylan producer, and relates them to the requirements of the language.

It contains sections on:

- Parsing and macro expansion
- Converting
- Optimising
- Code generation: the ‘emit’ protocols

### 2.2.3 Chapter 5. Tokens and the ICR

This gives details of the structure of the Dylan Token Library, with examples of how the tokens map onto the Implicit Continuation Representation (ICR) described in Chapter 4.

### 2.2.4 Chapter 6. The Dylan Runtime

This chapter describes the details of the Dylan runtime.

It contains sections on:

- ANDF token expansions
- Primitive functions in C

### 2.2.5 Chapter 7. Dylan Features Mapped to ANDF

This chapter takes a range of Dylan features and describes how they map onto ANDF.

#### 2.2.6 Chapter 8. Conclusions

This chapter presents conclusions and contains a bibliography.

# 3. The Dylan Implementation Model

## 3.1 Object Representation

Dylan objects are dynamically typed. The language permits a program to determine the class of any object at runtime. Harlequin's implementation achieves this by using a standard convention for associating the type with an object based on tagging.

In many circumstances, the Dylan compiler can statically determine the type of an object. This knowledge can be used to select an alternative representation which is more efficient than the canonical representation. For example, the canonical representation of a double float object in Dylan is as a pointer to heap-allocated storage which contains the IEEE bit pattern of the double float in addition to a reference to the Dylan class object <**double-float**>. The compiler may choose to represent the value as a direct bit pattern, wherever this does not violate the semantics of the program.

### 3.1.1 Tagging Scheme

All Dylan values are represented as data of the same size, or SHAPE. The size is the size of a pointer. The bit pattern of these values contains tag bits which indicate whether the value is actually a pointer, or whether it is a direct value.

To some extent, the details of the tagging scheme are abstracted by means of ANDF tokens. However, the compiler is aware of the groups of types which are similarly tagged. Since there are three major groups (integers, characters and everything else), the anticipated representation for all platforms is to use two bits.

Table 1. Tagging Scheme

| Tag Bits | Type |
|---|---|
| 00 | Heap Allocated |
| 01 | Integers |
| 10 | Characters |
| 11 | Unused |

### 3.1.2 Integers and Characters

Integers and characters are represented as direct values, using the tag bits as the only indication of type. The tagging scheme would normally use the least significant two bits (although this is only finally determined by the token library at install time). With the normal scheme, a character or integer is converted to its untagged representation by arithmetic right shifting by two bits. Similarly the conversion from an untagged to a tagged representation is to shift left and add in the tag bits.

Operations on these values (e.g., addition, or other arithmetic operations) are always performed on the untagged representation. This is sub-optimal, because it is possible to perform arithmetic operations directly on the tagged values. It is planned to improve this mechanism at a later date, along with a revision of the tagging scheme.

### 3.1.3 Boxed Objects

Apart from integers and characters, all Dylan objects are indirectly represented as *boxed* values (that is, they are pointers to heap allocated boxes). The runtime system is responsible for ensuring that these boxed values are appropriately tagged, because the runtime system provides the allocation service, and must ensure appropriate alignment.

Boxed objects are dynamically identified by their first slot, which is an identification wrapper. This identification wrapper (itself a boxed Dylan object) contains a pointer to the class of the object it is wrapping, as well as some encoded information for the garbage collector about which slots should be traced.

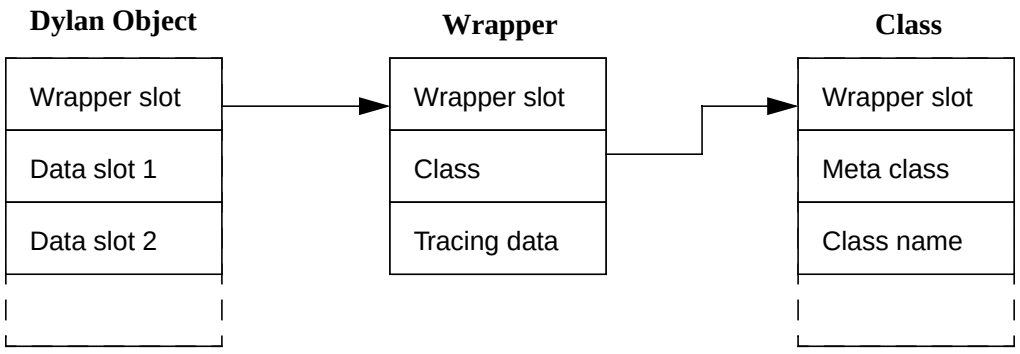


Figure 1. Boxed Objects

Note that two indirections are necessary to find the class of an object. In practice, this is a rare operation, because almost all dynamic class testing within Dylan is implicit, and the implementation can use the wrapper for these implicit tests. Note that there is potentially a many-to-one correspondence between wrapper objects and class objects.

The Dylan compiler builds literal boxed objects statically whenever it can. In practice, this will include most function objects apart from closures, virtually all wrappers, and most class objects, as well as strings, symbols, and literal vectors and lists.

### 3.1.4 Variably Sized Objects

Variably sized objects, such as strings, vectors and arrays, are boxed objects which contain a *repeated slot*. The repeated slot is implemented as a variably sized data area preceded by a normal slot containing the size of the variably sized data represented as a tagged integer. The size slot is used at the Dylan language level to determine the size of the array. There is also a special encoding for it in the tracing data of the

wrapper so that the memory manager knows how to trace the repeated data. For example, an instance of the **<byte-string> "foo"** is represented as in Figure 2 on page 5.

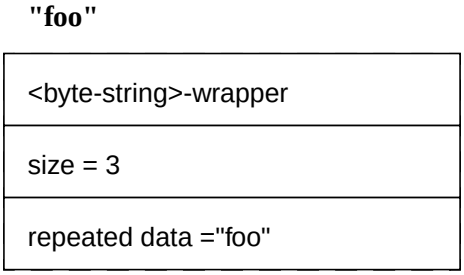


Figure 2. An Instance of **<byte-string>**

### 3.1.5 Function Objects

Dylan provides two built-in classes of functions: **<generic-function>** and **<method>**. These both obey the same general purpose calling convention, but also support specialised calling conventions (described below) which the compiler may use depending on the detail of its knowledge about the function being called, and the circumstances. Slots in the function object point to the code which implements each convention.

All functions also have a slot which encodes the number of required parameters the function accepts, and whether the function accepts optional or keyword parameters. Another slot in each function object contains a vector of the types which are acceptable for each required parameter. These slots are used for consistency checking of the arguments.

Generic functions have further slots which support the method dispatching process — including a slot which contains a vector of all the methods belonging to the generic function, and a slot which contains a cache of sorted applicable methods for combinations of arguments which have been processed before.

Methods may be closures, in which case a slot in the method object contains the environment for the method, which is represented as a vector of closed-over variables. If the variable is known actually to be constant, then the constant value is stored directly in the vector. Alternatively, if there is any possibility of an assignment to the variable, then the value is stored with an extra indirection to a *value cell*, which may be shared between many closures with related environments.

## 3.2 Calling Convention

Dylan permits any number of arguments of any Dylan class to be passed to a function. The implementation is obliged to check the arguments to ensure that there are an appropriate number, with appropriate types, and a Dylan condition must be signalled if the parameters are inappropriate. This general calling convention, supported by all function objects, is implemented by a slot called **XEP**, which points to the *eXternal Entry Point (XEP)* code for that function. The XEP checks all the arguments for consistency, and signals an error if necessary. If the function accepts keyword or optional arguments, then the supplied arguments are

processed, and appropriate values are calculated for the formal parameters of the function. If the parameters are not in error, the XEP code calls the *Internal Entry Point (IEP)* of the function.

Every Dylan function also has an internal entry point, which is represented as code stored in a slot called **IEP**. All argument checking and parameter processing must have been performed before calling the IEP. The compiler may choose to call the IEP of a function directly if it can verify and process the arguments of the call statically, at compile time.

The compiler is responsible for generating the IEP code for each Dylan function; however, it does not generate the XEP code. Functions with *congruent* parameters (i.e., same number of required parameters, and similar usage of optional and keyword parameters) share the same XEP code, which is provided by the runtime system. There are approximately 40 separate XEP functions in the runtime system, which are appropriate for all the variations of Dylan parameters. The fact that the XEP code is a shared resource leads to a significant saving in overall code size, compared with a system where the compiler generates argument checking code for each function. By implementing the XEP code in the runtime system, it is possible for it to be written with detailed knowledge of the calling convention for that platform, to an extent which is not describable in ANDF. This permits greater efficiency during the argument processing stage — for instance, when grouping together the dynamically sized number of optional parameters to produce a stack-allocated Dylan vector of them.

### 3.2.1 External Entry Point Convention

The XEP convention is the general calling convention for all Dylan functions. It involves passing the arguments of the function (the *language* parameters), as well two extra arguments (the *implementation* parameters). The implementation parameters are:

- *argcount*, the number of language parameters, expressed as an integer
- *function*, the Dylan function object being called

The implementation parameters are necessary for the implementation of the XEP code, so that the language arguments can be checked, and so that the IEP code can be located.

For example, consider the following Dylan code:

```
define method func1 (a, b, #rest optionals, #key key1, key2)
end method;

func1(1, 2, key2: 99);
```

For the call to **func1**, above, the parameters are described in Table 2 on page 6.

Table 2. XEP Parameters for the Call to **func1**

| **XEP Parameters** | **Values** |
|---|---|
| language parameters | **1**, **2**, **#"key2"**, **99** |
| *argcount* | 4 |
| *function* | generic function **func1** |

### 3.2.2 Internal Entry Point Convention

The IEP convention uses a fixed number of language parameters, corresponding to each of the parameters of the function (5 in the case of **func1**, above). In addition, there are two implementation parameters:

- *mlist*, a list of the next applicable methods to call if the function is a method called from a generic function (this parameter is used to support calls to **next-method**). If the function is not being called from a generic function, the value is **#f** (false).
- *function*, the Dylan function object being called (as for the XEP).

The implementation parameters are not obligatory for all IEP code. It is only necessary to pass *mlist* if the function contains a call to **next-method**. It is only necessary to pass *function* if the function is a closure (because the value is used by the IEP code to locate the environment of the closure). If the IEP is called from the XEP code, both the implementation parameters will always be set, even though they may not be necessary. For the same call to **func1**, above, the parameters are described in Table 3 on page 7.

Table 3. IEP Parameters for the Call to **func1**

| **IEP Parameters** | **Values** |
|---|---|
| language parameters | **1**, **2**, *optionals*, **#f**, **99** |
| *mlist* | **#f** |
| *function* | generic function **func1** |

Note that the language parameters now correspond to the formal parameters of the function, whereas, for the XEP, they corresponded to the supplied arguments.

The value of *optionals* in the set of language parameters is the Dylan vector **#[#"key2", 99]** which corresponds to all the optional arguments. The language parameter corresponding to **key1** is **#f**, because the keyword **#"key1"** was not supplied. However, the language parameter corresponding to **key2** is **99**, because **#"key2"** was supplied with that value.

### 3.2.3 Argument Passing Mechanisms

The XEP and IEP calling conventions are currently implemented using the standard C argument passing mechanism, corresponding to *caller parameters* in ANDF. For the XEP convention, the *function* and *argcount* parameters are passed first, respectively, followed by the language parameters. For the IEP convention, only the language parameters are passed in this way, and the *mlist* and *function* parameters are passed in global variables if required.

The usage of caller parameters is sub-optimal for Dylan, because it implies that no tail call optimisation is possible between the XEP and the IEP. This has a cost in terms of environment (stack) space and also because those parameters which are passed on the stack must be copied onto a new stack frame.

It is planned to use *callee* parameters for the Dylan language parameters, using the caller parameters for implementation parameters only. This will permit the XEP code to rearrange the language parameters as appropriate and tail-call the IEP with modified callee parameters (and overwritten callee parameters corresponding to the different set of implementation parameters).

Although this change is expected to relieve one of the last bottlenecks to implementation speed which is related to the use of ANDF, this has not yet been attempted. The cost of making the change is large, because it requires a complete rewrite of most of the runtime system code. This rewrite would also require a change of source language (the runtime system is currently written in C, but would probably be rewritten directly in TNC because no C compiler is compatible with ANDF's callee parameter mechanism). This calling convention change will be a necessary part of the evolution of the producer from *prototype* to *product* status.

## 3.3 Special Features

### 3.3.1 unwind-protect

**unwind-protect** (or **cleanup** as it has been renamed in infix Dylan) is used to protect a body of code so that a cleanup action will always be taken whenever the dynamic scope of the body is left, either because the body has finished normally, or because there has been a *non-local exit*, or *NLX* (see Section 3.3.2 on page 8).

The protection mechanism is implemented by maintaining a chain of *unwind protect frames (UPFs)*, which correspond to the currently active cleanups in the dynamic environment. The head of the chain is a global variable. When the dynamic scope of a new protected body is entered, the producer arranges for a new UPF structure to be created, as a local variable. The fields of this structure are then initialised. They include the next entry in the chain (the current contents of the global variable), the destination of the cleanup code for this frame (which is the result of a call to **setjmp**), and the ultimate destination for the NLX (which is initialised to 0 to indicate that there is no NLX occurring). The global variable is then updated to point to the new frame.

If the protected body exits normally, then the cleanup code is entered normally. The cleanup code is first evaluated, and then a function in the runtime system is called to mark the end of the cleanup. The runtime function first unlinks the UPF from the global chain, and then checks whether the cleanup was invoked because of a NLX (which is true if the ultimate destination field is not 0). In this case, as it is a normal exit, the runtime function just returns to the Dylan function, and the unwind-protect has finished.

If the cleanup code is invoked because of an NLX, then the runtime function finds the ultimate destination *bind exit frame* (*BEF*) from the UPF. The runtime function then passes this BEF to another runtime function (as for **bind-exit)** to test whether there are any further intervening cleanups, or to transfer control to the ultimate destination if not.

### 3.3.2 bind-exit

**bind-exit** is used to transfer control flow in Dylan to a destination point in the same lexical and dynamic scope. In many cases, the destination point is in the same function as the source, and the compiler is able to optimise such a case into a simple jump. The destination is not necessarily in the same function, however, because transfer may be from an inner closure to an outer one. Such a transfer of control is called a *non-local exit* (or *NLX*), and requires a more complex mechanism.

If a bind-exit destination may be reached by an NLX, then producer creates a new *bind exit frame (BEF)* structure, as a local variable. The fields of this structure are then initialised. They include the current cleanup environment (the contents of the global dynamic environment chain), and the destination of the

bind-exit (which is the result of a call to **setjmp**). The inner closure must store a pointer to the BEF in its environment, so that it can reach the destination.

When an NLX occurs, the transfer of control is implemented by a call into the runtime system, passing the pointer to the BEF as a parameter. The runtime function first checks whether there is an intervening cleanup, by testing whether the target dynamic environment in the BEF matches the current global dynamic environment. If there is no intervening cleanup, then the transfer to the destination is achieved with a **longjmp**. Alternatively, if there is an intervening cleanup, then the ultimate destination field of the current UPF is set to the destination BEF, and the cleanup code is invoked with a **longjmp**.

### 3.3.3 apply

The **apply** function is used to call a second function with a dynamically sized number of arguments. The dynamic arguments are supplied as a Dylan sequence. There is no direct support for this feature in ANDF if the caller parameter passing mechanism is used, so the final call to the second function is performed by code inside the runtime system. The part of the implementation which is managed by the producer copies the data of the Dylan sequence into a buffer, before calling the primitive function in the runtime system. The primitive function is then responsible for copying the data out of the buffer and arranging it in registers or on the stack for the call. The implementation is permitted to limit the number of values which may be passed in this way. Given a static upper bound of the number which may be passed, it is possible to implement the primitive function portably (e.g., in C) with every argument count case implemented with a separate static call.

In practice, it is very common for **apply** to be used as a mechanism for supplying all the optional parameters to a function. The compiler can often detect this case, and arrange to call the second function directly via its IEP, bypassing the primitive mechanism. In this case, the need to spread out the dynamic sequence of parameters is removed completely, as the IEP calling convention expects a formal optional parameter to be a dynamically sized vector.

### 3.3.4 Multiple Values

Dylan functions may return multiple values. Most callers require only a single value as a result of the function call, but callers may optionally determine exactly how many values were returned. Functions which return or use multiple values are the exception rather than the rule, in practice.

Harlequin's implementation of Dylan uses the normal ANDF function returning mechanism to return a single Dylan value, as this is the only value that is used by almost all callers. In addition, each function sets a count of the number of values being returned, in a global variable. This count can be examined by the caller, if required, to determine how many values were returned. If a function is returning more than one value, the additional values are stored in a global area, where the caller may retrieve then, if desired.

The need to set a global variable at every return for Dylan is a source of inefficiency for the implementation. However, it has not been a priority to find a more efficient solution, as the semantics of multiple-value returning in Dylan have recently been a subject of discussion. A language change has now been accepted which makes it possible to determine the number of return values statically in a large number of cases. We have not yet implemented this language change, but when we do it will only be necessary to set a multiple value return count in those rare circumstances when functions are defined to return a dynamically sized number of values.

## 3.4 Garbage Collection

Harlequin's Dylan implementation is designed to be used with various alternative types of garbage collector. This is achieved by using ANDF tokens to provide an abstract interface between the compiled code and the garbage collector. The features that this interface comprises are described in [Mann93a].

The garbage collector and the allocator are both provided by the Dylan runtime system, which can be implemented in a platform-specific manner for efficiency. The details of how the allocator and the collector interface with each other are localised within the runtime system, and the Dylan producer need not be aware of the design.

Support for incremental and/or generational collection is provided via a token abstraction for read and write barriers. The tokens for this are used to implement the readers and writers of slots in Dylan objects. In practice, it is anticipated that any read and write barriers will be implemented in hardware, rather than software, because that is potentially much more efficient. Many modern platforms provide access to the virtual memory page protection scheme, which may be used for this purpose. Harlequin's Dylan has only been tested with hardware read and write barriers — but the support is there for software barriers as and when required.

The Dylan heap may be traced accurately by the garbage collector, because each heap-allocated Dylan object refers to a wrapper object which contains tracing information for the garbage collector. However, no support is currently provided for accurately tracing the stack (although techniques were discussed in [Mann93a]).

It is currently anticipated that a conservative stack scan will always be used, as this offers many advantages, particularly when Dylan code is used with foreign code. The conservative stack scan places some limitations on the usage of optimisations involving derived pointers which may be performed by the compiler. These limitations are discussed in [BW88]. The Dylan producer is aware of these limitations, and always ensures that for each derived pointer, there is a variable in scope which contains a base pointer. Since the installer is also a part of the compiler, there is a requirement that the installer maintains certain invariants too. In practice a very aggressively optimising installer would be required to break these invariants, and it is believed that no current installer will cause problems. It would be desirable for installers to offer switches to guarantee safe behaviour for conservative garbage collection (not just for the benefit of Dylan, because many C and C++ applications use conservative garbage collection). This desire has been informally discussed with technical representatives of DRA and Etnoteam.

The combination of conservative stack scan but total tracing information for the heap makes it possible to implement a *mostly copying collector*, which is expected to be a typical algorithm for any given platform. Harlequin are in the process of implementing such a collector, but in the meantime, all testing of Dylan with ANDF has been with Boehm's fully conservative collector.

## 3.5 Name Mangling

In Dylan, unlike C, identifier names are case insensitive. Dylan also permits additional characters to appear in names. As a further complication, Dylan provides multiple namespaces, and the namespaces are controlled within a two-tier hierarchy of modules and libraries.

In order to make it possible to link Dylan code with tools designed to support more traditional languages, the Dylan producer transforms the names which appear in Dylan programs to C compatible names, via a process called *mangling*.

The library, module and identifier names are each processed, according to the following rules:

1. All uppercase characters are converted to lowercase.
2. Any character which appears on the left-hand side of Table 4 on page 11 is mapped to the new character sequence accordingly.

Table 4. Mangler Mapping Table

| Old | New | Comment |
|---|---|---|
| **-** | **_** | dash |
| **!** | **_E_** | exclamation |
| **$** | **_D_** | dollar |
| ***** | **_T_** | times |
| / | **_S_** | slash |
| < | **_L_** | less |
| > | **_G_** | greater |
| **?** | **_Q_** | question mark |
| + | **_PL_** | plus |
| **&** | **_AP_** | ampersand |
| ^ | **_CR_** | caret |
| **_** | **_UB_** | underbar |
| ~ | **_SG_** | squiggle |
| | **_SP_** | space |

Finally, the fully mangled name is created by concatenating the processed library, module, and identifier names respectively, separated by **X**.

For example, the Dylan identifier **add-new!** in module **internal** of library **dylan** would be mangled as **dylanXinternalXadd_new_E_**.

# 4. The Dylan Producer Representation

## 4.1 Introduction

Traditionally, compilers use one or two intermediate representations for code. The first of these is usually a parse tree which is a representation of the derivation of the source program, structured according to productions in the grammar of the language in which it is written. This tree carries both syntactic and semantic information, although it is the latter which is important for code generation. Code can be generated directly from the parse tree by a variety of techniques, or an intermediate form such as 3-address code can be generated before finally emitting the executable code.

Each representation is amenable to different kinds of analyses and optimisations. Higher-level structured representations being good for *global* optimisations involving mutation and / or movement of code. Lower-level or linear representations are good for local or *peephole* optimisations such as 'move', 'chain' or dead code elimination. ANDF itself falls into the latter category.

Harlequin's Dylan producer uses a single 'medium-level' intermediate representation which to some extent allows both global and local optimisations. This takes the form of a directed acyclic graph (DAG) composed of several layers, in which lower layers describe individual operations, and higher layers group the lower layers into structural units. This representation carries more semantic information, but there is no direct correspondence between its structure and the syntactic structure of the original source. A mapping between each node of the graph and the source that generated it is maintained, however.

Much of the grammatical structure of Dylan is built with *macros*. A macro is an extension to the core language that can be defined by the user, by the implementation, or as part of the Dylan language specification. A macro defines the meaning of one construct in terms of another construct. The compiler substitutes the new construct for the original. The purpose of macros is to allow programmers to extend the Dylan language, for example by creating new control structures or new definitions. Unlike C, Dylan does not intend macros to be used to optimise code by in-lining. Other parts of the language, such as sealing and **define constant**, address that need.

Compilation consists of a sequence of conceptual phases:

- Parsing a stream of characters into tokens, and parsing a stream of tokens into a program
- Macro expansion, which translates the program to a core language
- Converting, which recognises special and built-in definitions and builds a compile-time model of the static structure of the program
- Optimisation, which rewrites the program for improved performance
- Code generation, which translates the program to ANDF
- Installation, which translates ANDF to executable form

## 4.2 Parsing and Macro Expansion

An important feature of Dylan is a very powerful 'macro' system which adds great flexibility to the language. It allows new language constructs to be defined, e.g. a new 'for' construct, which will have the same status as pre-existing constructs.

In dealing with these the parsing and macro expansion phases of compilation are interleaved, not sequential. The parsing phase of the compiler parses a macro call just enough to find its end. This process of parsing a macro call also parses any macro calls nested inside it. The result is a macro call fragment.

Macro expansion replaces a macro call with another construct, which can itself be a macro call or contain macro calls. This expansion process repeats until there are no macro calls remaining in the program, thus macros have no space or speed cost at run time. Of course, expanding macros affects the speed and space cost of the compiler.

A macro definition describes both the syntax of a macro call and the process for creating a new construct to replace the macro call. Typically the new construct contains portions of the old one, which can be regarded as arguments to the macro. A macro definition consists of a sequence of rewrite rules. The left-hand side of

each rule is a pattern that matches a macro call. The right-hand side is a template for the expansion of a matching call. Pattern variables appearing in the left-hand side act as names for macro arguments. Pattern variables appearing in the right-hand side substitute arguments into the expansion. Macro arguments can be constrained to match specified elements of the Dylan grammar. Auxiliary rule sets enhance the rewrite rule notation with named sub-rules.

The input to, and output from, macro expansion is a fragment, which is a sequence of elementary fragments. The 'core' result of the inter-leaved parsing and macro expansion is passed to the compiler for processing in the subsequent stages.

## 4.3 Converting

This stage takes the 'core' language from the parsing and macro expansion phase and re-generates it as the intermediate representation that the compiler uses. In this process input syntax is matched to the intermediate representation. This representation is known as the *Implicit Continuation Representation* or ICR. Dylan has a rich set of language forms, but these can be reduced to a fairly small set of core forms in the ICR such as: **method**, **let**, **begin**, **if**. This conversion from source into ICR throws away large amounts of syntactic information although we still do the equivalent of 'beta transformation' on the ICR rather than on syntax. The initial conversion from source to ICR takes care of any name conflict problems by assigning unique names appropriately. The full Dylan method is implemented with a simple fixed-argument lambda, which greatly simplifies the later compiler phases.

The ICR representation is a flow-graph which directly supports flow analysis. It carries information describing both control and data flow. In relation to control flow, the progress of the program from point to point, (e.g. from 'if' to 'then' or 'else') is represented as arcs in the graph, while actual computations are represented as nodes. The number of different types of nodes is kept to a minimum, with effort being concentrated on producing high-quality optimisations for that small set.

Data flow to different storage locations are also represented on the arcs. The integrated representations of both control and data flow are known as *continuations* (see Table 5 on page 14). For each ICR node, these are resolved into a set of instructions related where necessary to specific data. For example, an 'assignment' node would be described in terms of a register transfer where the source would be the incoming continuation, and the destination would be the outgoing continuation. There are seven basic node types, as follows: a 'leaf' reference **<reference>**; assignment to variables **<assign>**; conditional execution **<if>**; a function call **<combination>**; function return **<return>**; start of dynamic extent **<entry>**; non-local exit **<exit>**.

Other elements of the ICR are leaves, blocks, and components. *Leaves* represent the computational actors, (i.e. variables, functions, and literals) in a Dylan program, and are classified into three main types: **<constant-leaf>**, **<variable-leaf>**, and **<function-leaf>**. A *block* is a contiguous set of nodes, or a basic block in the control sense within which there are no basic forks or joins of computational flow. A *component* is a connected piece of the flow graph. Most of the passes in the compiler operate on components rather than the entire flow graph.

Table 5. Elements in the ICR

| ICR Element | Types | Description |
|---|---|---|
| Node | **<reference>**<br>**<assign>**<br>**<if>**<br>**<combination>**<br>**<return>**<br>**<entry>**<br>**<exit>** | Nodes represent the program's actual computation. They are joined by arcs in the flow graph. |
| Continuation | | A continuation represents a place in the code, or, alternatively, the destination of an expression result and a transfer of control. It is an integrated representation of both control and data flow. |
| Leaf | **<constant-leaf>**<br>**<variable-leaf>**<br>**<function-leaf>** | Leaves represent the computational actors (variables, functions, and literals) in a Dylan program. |
| Block | | A block is a contiguous set of nodes, or a basic block in the control sense. |
| Component | | A component is a connected piece of the flow graph, computed as the equivalence class of functions' lexical environments. |

## 4.4 Optimisation

### 4.4.1 Introduction

Languages which are 'close to the machine' have generally concentrated on lower level or 'back-end' optimisations such as minimising jumps or eliminating unnecessary machine instructions. Dylan still applies these optimisations, but goes through a significant simplification step beforehand.

Analysis is used, for example, to determine when full function calls can be eliminated, and jumps used instead. Dylan does not have a 'GOTO', but only has methods and function calls. A simple loop such as 'for' loop inside a function is represented as a local function which is called recursively. In order to achieve good performance it is thus important for Dylan to recognise such cases where the function is not passed elsewhere, and is only used as a jump. The compiler must a) avoid allocating the method object and b) compile the code in-line in the enclosing function, and c) instead of making the full function call, jump as necessary.

Other optimisations include:

- dead code elimination
- constant folding
- partial evaluation based on constant folding and in-lining
- values

- block
- apply

These optimisations (described in more detail below) are ‘opportunistic’ in the sense that they happen automatically until no further opportunities for optimisation are available. The application of one technique may reveal conditions favouring the applications of another. Conceptually, these techniques are based on manipulations of the flow-graph. The essential point is that while Dylan is a powerful, high-level language, this generality needs to be stripped away from the executable code where it is not used or not necessary, in order to gain maximum efficiency. As much processing as possible is pushed into compile time, rather than run time. This includes type checking, keyword processing, and method selection. Another way to express this idea is to say that Dylan is a dynamic language which can be used statically. Where it is used in a static manner, the compiler should be able to optimise as appropriate.

A powerful technique in this context is representation analysis. Dylan objects are tagged with information about what type they have. This takes up extra space and in certain circumstances can mean that memory has to be allocated in order to make an object that can identify itself. An example is a double float. Dylan’s calling convention requires that anything can be passed anywhere, but double floats give a problem as they do not fit into a pointer space. If no analysis was done, and a new allocation was made on the heap for every operation involving a floating point, then problems would quickly arise.

### 4.4.2 Optimisation Examples

One of the techniques mentioned above was *constant folding*. This refers to the use of a known function with constants, where the function can actually be executed at compile time. The resulting value is then ‘folded in’ in place of the function call.

Constant folding is used at compile time in conjunction with in-lining and compile-time method dispatching to give a form of partial evaluation. This essentially involves doing as much of the execution as possible at compile-time. Where, for example, a function involving recursion, such as calculating a factorial, is given a known literal, the computation is actually carried out at compile time. This includes the check to see if the argument is less than 1, and the disposal of the ‘else’ part where it is not.

Some special optimisers exist for specific features of the language, such as ‘multiple values’. Where, for example, a multiple-value bind is carried out using **let**, we wish to optimise for the case where the binding involves a call to **values** as the form that we are binding from. This optimisation allows us to do assignment directly to the variables from the arguments. This is also important in other contexts, such as where the forward-iteration protocol is used. This protocol actually uses a call to **values**, so that when it is inlined, the optimisation applying to **values** can be triggered. This allows the much less expensive operation of assignment to be used.

Another optimisation applies to lexically scoped exits, i.e. a break from a loop, which may be achieved using **block**. The compiler notices these cases and optimises them into ‘jumps’.

In the case of **apply**, used with a known function and known arguments, the compiler will in-line the code as a straightforward function call. This may give opportunities for other optimisations to be applied.

## 4.5 Code Generation: The ‘Emit’ Protocols

For a given back-end, the emit protocols are defined as methods on a set of generic functions, all named by the prefix **emit**.

Signatures:

**emit** [Generic Function]

*Signature*

**(stream, back-end, object)**

*Description*

This walks the given object emitting ANDF for the containing ICR.

**emit** [Method]

*Signature*

**(stream , back-end, component :: <component>)**

*Description*

This emits all the lambdas in the given component.

**emit** [Method]

*Signature*

**(stream , back-end, lambda :: <lambda-leaf)**

*Description*

This emits a lambda-prolog, all the blocks, and then a lambda-epilog.

**emit@offset** [Generic Function]

*Signature*

**(stream, back-end, object, offset)**

*Description*

This emits the given object starting at the given offset. There are **emit@offset** methods for **<block>** and all **<node>**s.

**emit-name** [Generic Function]

*Signature*

**(stream, back-end, object)**

*Description*

This emits a reference to the given object. There are **emit-name** methods for <**continuation**> objects and <**leaf**> objects. It automatically handles closure variable references.

These functions and methods take a lambda leaf node from the ICR and turn it into a function definition. In the emitter body calls are made to the generic function **emit@offset** which specialises on such constructs as <**if**> for conditional execution, or <**assign**> to bind a value to a local variable. Other types of nodes may be specialised on <**reference**>, which is reference to a leaf in a compiler representing the value of a variable, or <**exit**> which supports unwind-protect and non-local exits. <**combination**>s are calls to functions which return values. <**primitive-combination**>s are calls to system primitives.

### 4.5.1 emit@offset on a <lambda-leaf>

This typically calls several functions. **emit-lambda-head** will output the first part of the function definition, i.e. the return type, the name, and any parameter information that is available. We emit **makeid_tagdef** which creates a tagged definition in ANDF. The name of the tag is the name of the function that we're creating. The definition of the tag is a **make_proc** to introduce the function definition itself. The first argument to the **make_proc** is the return type, and then we iterate across the parameters that the function takes, emitting them based on the type information contained in the back end. To close **emit-lambda-head** we output the start of a ‘sequence’ which tells ANDF to execute the following blocks.

### 4.5.2 emit-lambda-body-prolog

The next call is to **emit-lambda-body-prolog** which outputs local variable definitions which are used within the scope of the function. These represent the bindings in the Dylan code, along with any local variables that the compiler needs to implement that code. There are two special cases: if the ‘function’ object is required out of the current environment then we define a variable called **environment**, and use the token SLOT_READ to access the function element of the current environment; if a call to **next-method** is to be supported, we need to create a value to hold the **next-method** function object. All the local variables are created with the ANDF ‘variable’ construct. The lambda leaf itself which is passed into **emit-lambda-body-prolog** has an information slot which contains the local variables which are associated with this lambda. We iterate across these, and if necessary generate local variables which are tagged Dylan objects. Typing information is available in the ICR associated with each of the local variables.

It is possible that there are also local stack vector variables. These are a little more complicated to emit. To define the initial value for such a vector, we use **make_nof** to create elements containing the class header (for a SOV), and the vector size, and **in_copies** to reserve space for **size** data elements. These two component parts are wrapped with **concat_nof** to tie them together.

If there is any non-local-exit potential with this function we create an ‘unwind-protect frame’ or a ‘bind-exit’ frame to control it.

### 4.5.3 emit-lambda-body

This outputs the part of the lambda which actually creates effects. The lambda leaf is described as a series of <**block**>s. (A <**block**> is section of code which is entered at the top, and exited at the bottom, without any exits in-between). A <**block**> is modelled as being contained by a 'sequence construction' and this set

of sequences is contained within a 'labelled' declaration, which introduces a label name per **<block>**, and allows transfer of control at the end of each block with a GOTO to the appropriately named label.

#### 4.5.4 emit-lambda-body-epilog

This performs the necessary cleanups for the ‘emit’ functions. It also outputs a token called ENDPROC, which could contain any necessary cleanup for the end of a method call.

#### 4.5.5 emit@offset for the ANDF Back End

There are several definitions of this specialised on different ‘node’ types. One example is that specialised on the **<return>** node type, which is used to return a value from a method. It calls **emit-begin-return** (the generic function which will be specialised on a specific back end), to emit the appropriate code to return a value. It then emits the name of the return value, the value itself, and calls **emit-end-return** to close the return statement. For the ANDF back end, this means that we emit **return**, the value and the matching close parenthesis.

The emitter-offset for an **<if>** node generates a branch based on a condition. There are three possibilities here. We could branch if a value is ‘true, ‘false’, or either ‘true’ or ‘false’. This makes use of some ANDF tokens called GOTO_IF_TRUE, GOTO_IF_FALSE, and GOTO_IF_TRUE_FALSE. The token library defines the appropriate behaviour of these tokens. They are given the result of a test as the condition to branch on, and a destination label (see above). The definition of a **<block>** means that such a branch is the last action of a given **<block>**.

Emitter-offset for a combination is the method used output function calls. (A **<combination>** represents a function or generic function call). There are two main cases here. This offset will either call **emit-local-call@offset** if the compiler has been able to determine which function will be called a compile-time, or else call **emit-regular-call-offset**, if it has insufficient information for this. The body of the **local-call** method is based on **apply_proc** and outputs the ANDF for the call. If the results of the call are being used elsewhere, the results of the call are assigned to the appropriate destination. The **regular-call** method is similar, except that it relates to the use of generic functions. This means that we need to inspect the slots of the generic function to determine the first parameter for the call to **apply_proc**. This is achieved through the use of the SLOT_READ token. The next parameter is the number of arguments being passed to the generic function, with the third being the arguments themselves.

# 5. Tokens and the ICR

A range of tokens are used with the Dylan Producer. Some of these support Dylan itself, while others are imported from the C API that comes with ANDF. This chapter gives an overview of Dylan tokens, describes their use in relation to the ICR, and lists the signature for each token used.

## 5.1 Overview of Dylan Tokens

Tokens are available for each of the following categories:

### 5.1.1 Coercions

These perform a function equivalent to type casting in C. This allows such types as pointers to void to be ‘cast’ to other types such as integers and floats.

### 5.1.2 Constants

There are some constant definitions used to simplify work in the compiler. An example would be where we wish to output '8' as a raw integer value, rather than as a tagged Dylan value. In this instance the compiler would emit the token _8, one of a handful of tokens defined as a short-form representation of common low valued integers, which avoids cluttering the output with numerous constructs using **make_int**.

### 5.1.3 Primitive Operations

These include conditionals, tests for equality, less-than, greater-than, arithmetic operations, shifting, and logical operations. These are defined for integers and floats.

### 5.1.4 String Routines

These support the declaration of string literals.

### 5.1.5 Runtime Structures

There are structure definitions for generic function objects, for non-local exit frames, and unwind-protect frames. These support the definition and slot access for the relevant objects.

### 5.1.6 Thread Primitives

These call the C implementations for the Dylan thread library.

### 5.1.7 Frequently Used Operations

These cover a range of actions and common abbreviations, such as using OT for **obtain_tag**, taking the raw value of a Dylan integer, or tagging a raw value into a Dylan integer. Other tokens manipulate characters; access the value of variables; assign values to variables; read and write to vectors; define the GOTO_ conditional operations.

### 5.1.8 Shape Definitions

These define such types as pz, which is Dylan's equivalent of a pointer to **void**, along with a range of integer types that are used by the compiler.

## 5.2 Token Use in Relation to the ICR

Tokens do not map directly onto the ICR. They are generated by the 'emit' code, and typically are used to abstract-away from the details of the ANDF in order to make the generated code more readable and compact. Much generated TDF is given in the form of tokens, with little 'bare' TDF left. Tokens may refer to

other tokens. The use of tokens can also make the construction of the compiler simpler, as the definition of the token can be changed without the need to change or re-compile the compiler.

### 5.2.1 ICR Node <entry>

This node indicates the start of an 'unwind-protect' or 'bind-exit' section. This is emitted as a sequence of tokens. When and unwind-protect section is entered, a set of information has to be stored in an unwind-protect frame. This requires multiple SLOT_WRITEs. A call is also made to the ANSI C **setjmp** function which stores the current state of the program. One of the parameters to **setjmp** is itself a token which does a SLOT_READ to determine where **setjmp** should store its current state.

### 5.2.2 ICR Node <exit>

This node represents the condition where unwind-protect or bind-exit performs a non-local exit. It generates a call to a fixed function which is a Dylan non-local-exit handler along with the appropriate 'nlx' information and any arguments that the function is passed.

### 5.2.3 ICR Node <if>

This node makes use of tokens called GOTO_IF_TRUE, GOTO_IF_FALSE, and GOTO_IF_TRUE_FALSE. These represent branches for 'true, 'false', or either 'true' or 'false'. There are also 'primitive' versions of these tokens which are invoked when the code is manipulating real, rather than 'Dylan' values.

### 5.2.4 ICR Node <reference>

The node refers to a lambda variable leaf, or a global variable leaf. The node can have a <**continuations**> object, which indicates that is should potentially have its value assigned to another variable. This could use such tokens as ASSIGN, WRITE_ATOMIC, or WRITE_THREAD.

### 5.2.5 ICR Node <return>

This calls the TDF construct 'return' and a value. The value itself could well use a token such as RVAL or LVAL to obtain the correct representation of the entity to be returned. The precise token used depends on the type of the information to be returned.

### 5.2.6 ICR Node <combination>

A <**combination**> represents a function or generic function call. There are two main cases here:

- Where the compiler has been able to determine which function will be called at compile time
- Where the call represents a call to generic dispatch via a generic function

In the first case, where the function is known, an **apply_proc** is emitted to the appropriate method. In the second, **apply_proc** is called on the appropriate entry-point of the generic function object.

It is not usually possible to use a single token to represent the entirety of one of these calls, as tokens cannot take a variable number of parameters, whereas **apply_proc** may well do so. An example of the tokens involved here is that which supports a slot read in the generic dispatch to extract the appropriate entry-

point from the generic function object. Others would be used to access the arguments to be passed in various ways, as when and array element has to be read. Tokens are also often used here as a shorthand for integer constants, as the number of arguments being passed has to be specified.

### 5.2.7 ICR Node <funny-function-combination>

A 'funny function' is a special function known by the compiler that has a special emitting procedure (and thus does not necessarily obey normal calling conventions). A <**funny-function-combination**> is a node representing a call to a funny function.

Examples:

The following call creates a raw value:

```
% immediate(1);
```

The following call invokes a method, also specifying the next-methods:

```
%method-call(function, next-methods, a1, …, an);
```

### 5.2.8 ICR Node <multiple-value-combination>

This represents a special call to a function of n parameters with the multiple values returned from evaluating its single argument. this call is then optimised down to assignments if its argument is a call to **values**, or is emitting by fetching the arguments from the multiple values buffer.

Example:

```
begin
    let (a, b) = truncate(1.5);
    a - b
end
```

gives:

```
multiple-value-combination(method (a, b) a - b end, truncate(1.5))
```

### 5.2.9 ICR Node <primitive-combination-leaf>

This implements *primitives* in the compiler. These look like regular function calls but are treated specially by the back end. A primitive has an associated emitter-function which is called to handle it, and emit the correct output. Typically primitives are used to generate calls to the runtime system or to special ANDF tokens rather than calls to generic functions, so they will generate calls that do not have the supporting structures associated with a call to a generic function. Tokens which might be used are OT on the name of the primitive itself, or one of a series of RVALs or LVALs on the arguments being passed.

## 5.3 Primitives and Token Mappings

This section details the primitives used. Unless otherwise specified, these map to calls into the runtime system, which are described in chapter 6.

### 5.3.1 General Tokens

pz [Token]

*Signature*

→ SHAPE

*Description*

Returns the same shape as a ~ptr_void from the ANSI API. This is the generic type used extensively throughout the compiler.

OT_TAG [Token]

*Signature*

*x*: EXP
→ EXP *x*

*Description*

Convenience shorthand for **obtain_tag**.

### 5.3.2 General Primitives

**primitive-make-box** [Primitive]

*Signature*

**(object :: <object>) => <object>**

**primitive-allocate** [Primitive]

*Signature*

**(size :: <raw-small-integer>) => <object>)**

**primitive-byte-allocate** [Primitive]

*Signature*

**(word-size :: <raw-small-integer>, byte-size :: <raw-small-integer>) => <object>)**

**primitive-make-environment** [Primitive]

*Signature*

**(size :: <raw-small-integer>) => <object>**

**primitive-copy-vector** [Primitive]

*Signature*

**(vector :: <object>) => <object>**

**primitive-make-string** [Primitive]

*Signature*

**(vector :: <raw-c-char*>) => <raw-c-char*>**

**primitive-function-code** [Primitive]

*Signature*

**(function :: <object>) => <object>**

**primitive-function-environment** [Primitive]

*Signature*

**(function :: <object>) => <object>**

### 5.3.3 Low-Level Apply Primitives

**primitive-xep-apply** [Primitive]

*Signature*

**(function :: <object>, buffer-size :: <raw-small-integer>, buffer :: <object>) => :: <object>**

**primitive-iep-apply** [Primitive]

*Signature*

**(function :: <object>, buffer-size :: <raw-small-integer>, buffer :: <object>) => <object>)**

**primitive-true?** [Primitive]

*Signature*

**(value :: <raw-small-integer>) => <object>**

*Description*

This primitive returns Dylan true if *value* is non-zero, and false if *value* is zero. The token ~i_to_tf is emitted to achieve this.

~i_to_tf [Token]

*Signature*

*i*: EXP pz
→ TAG pz

*Description*

Performs 'if *i* then **#t** else **#f**'.

**primitive-false?** [Primitive]

*Signature*

**(value :: <raw-small-integer>) => <object>**

*Description*

This is the complement of **primitive-true?**, returning **#t** if the value is 0, **#f** otherwise. The token ~i_to_ft is used for this.

~i_to_ft [Token]

*Signature*

*i*: EXP pz
→ TAG pz

*Description*

Performs 'if *i* then **#f** else **#t**'.

**primitive-equals?** [Primitive]

*Signature*

**(x :: <object>, y :: <object>) => <raw-c-int>**

This is an operator; it maps to the token EQUALS.

EQUALS [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* == *y* then 1 else 0.

**primitive-continue-unwind** [Primitive]

*Signature*

**() => <object>**

**primitive-nlx** [Primitive]

*Signature*

**(bind-exit-frame :: <raw-c-void*>, args :: <raw-c-void*>) => <raw-c-void>**

**primitive-inlined-nlx** [Primitive]

*Signature*

**(bind-exit-frame :: <raw-c-void*>, first-argument :: <raw-c-void*>) => <raw-c-void>**

**rimitive-variable-lookup** [Primitive]

*Signature*

**(variable-pointer :: <raw-c-void*>) => <raw-c-void*>**

**primitive-variable-lookup-setter** [Primitive]

*Signature*

**(value :: <raw-c-void*>, variable-pointer :: <raw-c-void*>) => <raw-c-void*>**

### 5.3.4 Integer Primitives

**primitive-int?** [Primitive]

*Signature*

**(x :: <object>) => <raw-small-integer>**

There are several sets of primitive operators defined on integers, which all make use of the following tokens:

INT_EQ [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* == *y* then 1 else 0.

INT_NEQ [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* != *y* then 1 else 0.

INT_LT [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* < *y* then 1 else 0.

INT_LE [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* <= *y* then 1 else 0.

INT_GT [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* > *y* then 1 else 0.

INT_GE [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* >= *y* then 1 else 0.

INT_NEG [Token]

*Signature*

*x*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

Negate *x*.

INT_ADD [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

Sum *x* and *y*.

INT_SUB [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

Perform *x* – *y*.

INT_MULT [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

*x * y*

INT_DIV [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

*x / y.*

INT_MOD [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

*x % y.*

INT_LSHIFT [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

*x << y.*

INT_RSHIFT [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

*x >> y.*

INT_NOT [Token]

*Signature*

*x*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

Bitwise not(*x*).

INT_AND [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

Bitwise *x* AND *y.*

INT_OR [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

Bitwise *x* or *y.*

INT_XOR [Token]

*Signature*

*x*: EXP INTEGER(iv)
*y*: EXP INTEGER(iv)
→ EXP INTEGER(iv)

*Description*

Bitwise *x* exclusive-or *y*.

**primitive-address-equals?** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

**primitive-address-add** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

**primitive-address-subtract** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

**primitive-address-multiply** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

**primitive-address-left-shift** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

**primitive-address-right-shift** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

**primitive-address-not** [Primitive]

*Signature*

**(x :: <raw-address>) => <raw-address>**

**primitive-address-and** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

**primitive-address-or** [Primitive]

*Signature*

**(x :: <raw-address>, y :: <raw-address>) => <raw-address>**

The following primitives map to the tokens listed above:

**primitive-small-integer-equals?** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-not-equals?** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-less-than?** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-greater-than?** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-greater-than-or-equal?** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-negate** [Primitive]

*Signature*

**(x :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-add** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-subtract** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-multiply** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-divide** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-modulo** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-left-shift** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-right-shift** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-not** [Primitive]

*Signature*

**(x :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-and** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-or** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

**primitive-small-integer-xor** [Primitive]

*Signature*

**(x :: <raw-small-integer>, y :: <raw-small-integer>) => <raw-small-integer>**

In addition to the small-integer operators above, there are also definitions for three other integer types, defined in the same manner. Table 6 on page 33 summarises the relationship between these types and Dylan primitives.

Table 6. Integer Types and Dylan Primitives

| **General Variety of Integer** | **Class of Primitive Parameters and Return Values** | **Value of *type* in Primitive Name primitive-*type*-*operator*** |
|---|---|---|
| Small Integer | **<raw-small-integer>** | **small-integer** |
| Big Integer | **<raw-big-integer>** | **big-integer** |
| Machine Integer | **<raw-machine-integer>** | **machine-integer** |
| Unsigned Machine Integer | **<raw-unsigned-machine-integer>** | **unsigned-machine-integer** |

### 5.3.5 Float Primitives

**primitive-decoded-bits-as-single-float** [Primitive}

*Signature*

**(sign :: <raw-small-integer>, exponent :: <raw-small-integer>,**
**significand :: <raw-small-integer>) => <raw-single-float>)**

**primitive-bits-as-single-float** [Primitive]

*Signature*

**(x :: <raw-small-integer>) => <raw-single-float>**

*Description*

Uses a custom emitter to map to a call to a function called **integer_to_single_float** in the runtime system.

**primitive-single-float-as-bits** [Primitive]

*Signature*

**(x :: <raw-single-float>) => <raw-small-integer>**

*Description*

Uses a custom emitter to map to a call to a function called **single_float_to_integer** in the runtime system.

**primitive-single-float-equals?** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-c-int>**

**primitive-single-float-not-equals?** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-c-int>**

**primitive-single-float-less-than?** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-c-int>**

**primitive-single-float-less-than-or-equal?** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-c-int>**

**primitive-single-float-greater-than?** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-c-int>**

**primitive-single-float-greater-than-or-equal?** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-c-int>**

The single-float predicate primitives above map onto the following tokens:

FLOAT_EQ [Token]

*Signature*

*x*: EXP FLOATING(fv)
*y*: EXP FLOATING(fv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* == *y* then 1 else 0.

FLOAT_NEQ [Token]

*Signature*

*x*: EXP FLOATING(fv)
*y*: EXP FLOATING(fv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* != *y* then 1 else 0.

FLOAT_LT [Token]

*Signature*

*x*: EXP FLOATING(fv)
*y*: EXP FLOATING(fv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* < *y* then 1 else 0.

FLOAT_LE [Token]

*Signature*

*x*: EXP FLOATING(fv)
*y*: EXP FLOATING(fv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* <= *y* then 1 else 0.

FLOAT_GT [Token]

*Signature*

*x*: EXP FLOATING(fv)
*y*: EXP FLOATING(fv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* > *y* then 1 else 0.

FLOAT_GE [Token]

*Signature*

*x*: EXP FLOATING(fv)
*y*: EXP FLOATING(fv)
→ EXP INTEGER(~signed_long)

*Description*

If *x* >= *y* then 1 else 0.

**primitive-single-float-negate** [Primitive]

*Signature*

**(x :: <raw-single-float>) => <raw-single-float>**

**primitive-single-float-add** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-single-float>**

**primitive-single-float-subtract** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-single-float>**

**primitive-single-float-multiply** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-single-float>**

**primitive-single-float-divide** [Primitive]

*Signature*

**(x :: <raw-single-float>, y :: <raw-single-float>) => <raw-single-float>**

**primitive-single-float-unary-divide** [Primitive]

*Signature*

**(x :: <raw-single-float>>) => <raw-single-float>**

The single-float operators above correspond to the following tokens:

FLOAT_NEG [Token]

*Signature*

*x*: EXP FLOATING(f)
→ EXP FLOATING(f)

*Description*

*-x*.

FLOAT_ADD [Token]

*Signature*

*x*: EXP FLOATING(f)
*y*: EXP FLOATING(f)
→ EXP FLOATING(f)

*Description*

*x + y*.

FLOAT_SUB [Token]

*Signature*

*x*: EXP FLOATING(f)
*y*: EXP FLOATING(f)
→ EXP FLOATING(f)

*Description*

*x – y*.

FLOAT_MULT [Token]

*Signature*

*x*: EXP FLOATING(f)
*y*: EXP FLOATING(f)
→ EXP FLOATING(f)

*Description*

x * *y*.

FLOAT_DIV [Token]

*Signature*

*x*: EXP FLOATING(f)
*y*: EXP FLOATING(f)
→ EXP FLOATING(f)

*Description*

*x* / *y*.

FLOAT_RECIPROCAL [Token]

*Signature*

*x*: EXP FLOATING(f)
→ EXP FLOATING(f)

*Description*

1 / *x*.

### 5.3.6 Accessor Primitives

**primitive-element** [Primitive]

*Signature*

**(array :: <object>, index :: <raw-small-integer>) => <object>**

*Description*

This is used for de-referencing slots in the middle of Dylan objects, and thus potentially invokes read-barrier code. It takes two parameters: a Dylan object, and an index which is the ‘word’ index into the object. It returns the Dylan value found in that corresponding slot.

Uses a custom emitter and the token READ_VECTOR_ELEMENT.

READ_VECTOR_ELEMENT [Token]

*Signature*

*vector*: EXP pz
*element*: EXP INTEGER(~signed_long)
→ EXP pz

*Description*

Return the contents of *vector*[*element*].

**primitive-element-setter** [Primitive]

*Signature*

**(new-value :: <object>, array :: <object>, index :: <raw-small-integer>) => <object>**

*Description*

This is the assignment operator corresponding to **primitive-element**, which is used to change the value of a Dylan slot. This takes an extra initial parameter which is the new value to put into the object. The new value is stored in the appropriate object at the given index.

Uses a custom emitter and the token WRITE_VECTOR_ELEMENT.

WRITE_VECTOR_ELEMENT [Token]

*Signature*

*vector*: EXP pz
*element*: EXP INTEGER(~signed_long)
*value*: EXP pz
→ EXP pz

*Description*

*vector*[*element*] := *value*.

**primitive-byte-element** [Primitive]

*Signature*

**(array <object>, base-index :: <raw-small-integer>, byte-offset :: <raw-small-integer>) => <raw-c-char>**

*Description*

This is similar to **primitive-element**, but deals with byte vectors. It takes a new value and a Dylan object, along with a base offset and a byte offset. The base offset, expressed in words, and the byte offset, expressed in bytes, are added, and the byte found at that location is returned.

**primitive-byte-element-setter** [Primitive]

*Signature*

**(new-value :: <raw-c-char>) array :: <object>, base-index :: <raw-small-integer>, byte-offset :: <raw-small-integer>) => <raw-c-char>**

*Description*

This is the corresponding setter for **primitive-byte-element**.

**primitive-fill!** [Primitive]

*Signature*

**(array :: <object>, size :: <raw-small-integer>, value :: <object>) => <object>**

**primitive-replace!** [Primitive]

*Signature*

**(new-array :: <object>, array :: <object>, size :: <raw-small-integer>) => <object>**

**primitive-replace-bytes!** [Primitive]

*Signature*

**(dst :: <raw-c-void*>, src :: <raw-c-void*>, size :: <raw-c-int>) => <raw-c-void>**

*Description*

Maps to the token ansi.string.memcpy in the ANSI API.

The following primitives, named **primitive-***type***-at** and **primitive-***type***-at-setter** load or store, respectively, a value of the designated *type* at the specified address. All of these use one of a pair of custom emitters.

**primitive-untyped-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-untyped>**

**primitive-untyped-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-untyped>, address :: <raw-pointer>) => <raw-untyped>**

**primitive-pointer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-pointer>**

**primitive-pointer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-pointer>, address :: <raw-pointer>) => <raw-pointer>**

**primitive-byte-character-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-byte-character>**

**primitive-byte-character-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-byte-character>, address :: <raw-pointer>) => <raw-byte-character>**

**primitive-small-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-small-integer>**

**primitive-small-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-small-integer>, address :: <raw-pointer>) => <raw-small-integer>**

**primitive-big-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-big-integer>**

**primitive-big-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-big-integer>, address :: <raw-pointer>) => <raw-big-integer>**

**primitive-machine-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-machine-integer>**

**primitive-machine-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-machine-integer>, address :: <raw-pointer>) => <raw-machine-integer>**

**primitive-unsigned-machine-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-unsigned-machine-integer>**

**primitive-unsigned-machine-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-unsigned-machine-integer>, address :: <raw-pointer>)**
**=> <raw-unsigned-machine-integer>**

**primitive-single-float-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-single-float>**

**primitive-single-float-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-single-float>, address :: <raw-pointer>) => <raw-single-float>**

**primitive-double-float-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-double-float>**

**primitive-double-float-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-double-float>, address :: <raw-pointer>) => <raw-double-float>**

**primitive-extended-float-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-extended-float>**

**primitive-extended-float-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-extended-float>, address :: <raw-pointer>) => <raw-extended-float>**

**primitive-signed-8-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-signed-8-bit-integer>**

**primitive-signed-8-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-signed-8-bit-integer>, address :: <raw-pointer>)**
**=> <raw-signed-8-bit-integer>**

**primitive-unsigned-8-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-unsigned-8-bit-integer>**

**primitive-unsigned-8-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-unsigned-8-bit-integer>, address :: <raw-pointer>)**
**=> <raw-unsigned-8-bit-integer>**

**primitive-signed-16-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-signed-16-bit-integer>**

**primitive-signed-16-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-signed-16-bit-integer>, address :: <raw-pointer>)**
**=> <raw-signed-16-bit-integer>**

**primitive-unsigned-16-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-unsigned-16-bit-integer>**

**primitive-unsigned-16-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-unsigned-16-bit-integer>, address :: <raw-pointer>)**
**=> <raw-unsigned-16-bit-integer>**

**primitive-signed-32-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-signed-32-bit-integer>**

**primitive-signed-32-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-signed-32-bit-integer>, address :: <raw-pointer>)**
**=> <raw-signed-32-bit-integer>**

**primitive-unsigned-32-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-unsigned-32-bit-integer>**

**primitive-unsigned-32-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-unsigned-32-bit-integer>, address :: <raw-pointer>)**
**=> <raw-unsigned-32-bit-integer>**

**primitive-signed-64-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-signed-64-bit-integer>**

**primitive-signed-64-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-signed-64-bit-integer>, address :: <raw-pointer>)**
**=> <raw-signed-64-bit-integer>**

**primitive-unsigned-64-bit-integer-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-unsigned-64-bit-integer>**

**primitive-unsigned-64-bit-integer-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-unsigned-64-bit-integer>, address :: <raw-pointer>)**
**=> <raw-unsigned-64-bit-integer>**

**primitive-ieee-single-float-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-ieee-single-float>**

**primitive-ieee-single-float-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-ieee-single-float>, address :: <raw-pointer>) => <raw-ieee-single-float>**

**primitive-ieee-double-float-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-ieee-double-float>**

**primitive-ieee-double-float-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-ieee-double-float>, address :: <raw-pointer>)**
**=> <raw-ieee-double-float>**

**primitive-ieee-extended-float-at** [Primitive]

*Signature*

**(address :: <raw-pointer>) => <raw-ieee-extended-float>**

**primitive-ieee-extended-float-at-setter** [Primitive]

*Signature*

**(new-value :: <raw-ieee-extended-float>, address :: <raw-pointer>)**
**=> <raw-ieee-extended-float>**

### 5.3.7 Other Tokens Produced When Emitting Nodes

There are two structure definitions used to hold data on bind-exit frames and unwind-protect frames, and the emit code for the **<entry>** node type makes uses of a couple of tokens to parameterise access to the slots of these structures.

SLOT_READ [Token]

*Signature*

| | |
|---|---|
| *result*: | SHAPE |
| *al_struct*: | AL_TAG |
| *struct_base*: | EXP pz |
| *field*: | EXP OFFSET(f) |
| → | EXP *result* |

*Description*

Reads a slot from a structure.

SLOT_WRITE [Token]

*Signature*

| | |
|---|---|
| *al_struct*: | AL_TAG |
| *struct_base*: | EXP pz |
| *field*: | EXP OFFSET(f) |
| → | EXP POINTER(ALIGNMENT) |

*Description*

Delivers a slot of a structure in a form that can be assigned to. That is, SLOT_WRITE does not assign to the structure, but returns a form suitable for use as the first argument to (**assign**...).

The <**if**> node potentially uses one of the following tokens:

GOTO_IF_TRUE [Token]

*Signature*

| | |
|---|---|
| *test*: | EXP pz |
| *truelab*: | LABEL |
| → | EXP TOP |

*Description*

If *test* != **#f** then goto *truelab.*

GOTO_IF_FALSE [Token]

*Signature*

| | |
|---|---|
| *test*: | EXP pz |
| *falselab*: | LABEL |
| → | EXP TOP |

*Description*

If *test* == **#f** then goto *falselab.*

GOTO_IF_TRUE_FALSE [Token]

*Signature*

*test*: EXP pz
*truelab*: LABEL
*falselab*: LABEL
→ EXP BOTTOM

*Description*

If *test* != **#f** then goto *truelab* else goto *falselab.*

PRIM_GOTO_IF_TRUE [Token]

*Signature*

*test*: EXP pz
*truelab*: LABEL
→ EXP TOP

*Description*

If *test* != 0 then goto *truelab.*

PRIM_GOTO_IF_FALSE [Token]

*Signature*

*test*: EXP pz
*falselab*: LABEL
→ EXP TOP

*Description*

If *test* == 0 then goto *falselab.*

PRIM_GOTO_IF_TRUE_FALSE [Token]

*Signature*

*test*: EXP pz
*truelab*: LABEL
*falselab*: LABEL
→ EXP BOTTOM

*Description*

If *test* != 0 then goto *truelab* else goto *falselab.*

### 5.3.8 Literal Dumping

Dylan integers are dumped using the following token:

INTEGER [Token]

*Signature*

*value*: SIGNED_NAT
→ EXP pv

*Description*

Takes a natural number and puts it in the tagged Dylan representation.

Characters use an equivalent token:

CHAR [Token]

*Signature*

*value*: SIGNED_NAT
→ EXP pv

*Description*

Take an ASCII value for a character and convert to the tagged representation.

Strings also make use of a special set of tokens to inline the string data. They are named STRING_*n*, where *n* is an integer from 0 through 4. STRING_*n* takes *n* ASCII values and packages them up into the space of a single integer to form a contiguous string representation that can be used inline, since TDF's **make_string** produces an out-of-line string.

STRING_0 [Token]

*Signature*

→ EXP pv

STRING_1 [Token]

*Signature*

*one*: SIGNED_NAT
→ EXP pv

STRING_2 [Token]

*Signature*

*one*: SIGNED_NAT
*two*: SIGNED_NAT
→ EXP pv

STRING_3 [Token]

*Signature*

*one*: SIGNED_NAT
*two*: SIGNED_NAT
*three*: SIGNED_NAT
→ EXP pv

STRING_4 [Token]

*Signature*

*one*: SIGNED_NAT
*two*: SIGNED_NAT
*three*: SIGNED_NAT
*four*: SIGNED_NAT
→ EXP pv

# 6. The Dylan Runtime

## 6.1 ANDF Token Expansions

The output of the Dylan producer is read by the installer for a specific platform which has access to a full set of token definitions. Wherever a reference to a token appears in the producer's output, the token's expansion is inserted by the installer, and runnable code is output. This allows the correct token library for a specified platform to be used.

### 6.1.1 Example Expansion 1: INTEGER

In addition to outputting integers in a 'raw' format, the compiler also outputs them in the tagged Dylan representation. ANDF achieves this using the token INTEGER. This utilises another token called TAG_INTEGER, which performs the actual tagging. This involves shifting the number left two places (using **shift_left**), with 0 1 being inserted as the tag on the two least significant bits (using **or**). INTEGER then takes the result of TAG_INTEGER and coerces it back to the generic shape pz that is used extensively throughout the compiler.

### 6.1.2 Example Expansion 2: CHARACTER

Dylan characters also have a tagged representation, with 1 0 as the least significant bits. The 'raw' value is again shifted left two places, with 1 0 being inserted as the tag on the two least significant bits. The shifting and tagging is done for ANDF by the CHAR token.

### 6.1.3 Example Expansion 3: GOTO_IF_TRUE

This token is one of the constituents of conditional execution. The token takes the values to test, and a destination to jump to if the test yields true.

The definition uses the ANDF **pointer_test** which takes two expressions to compare using a test. If the test succeeds it yields ‘top’ and carries on. If the test fails, it jumps to a 'label', (the target of a GOTO). In this token, the test to be used is hard-coded as the 'equal' test, and the Dylan value that we are checking is compared with the Dylan 'false' value.

Thus if 'value equal false' GOTO_IF_TRUE does not want to GOTO anywhere, and so yields ‘top’ and carries on. If the test fails, i.e., is not Dylan false and so is a 'true' value (in Dylan, anything other than the canonical false value is considered to be true), GOTO_IF_TRUE will jump to the target label specified.

### 6.1.4 Example Expansion 4: GOTO_IF_FALSE

This token is written in exactly the same way as GOTO_IF_TRUE, except that it uses the 'not_equal' instead of the 'equal' test.

### 6.1.5 Example Expansion 5: SLOT_READ, SLOT_WRITE, and STRUCT_ACCESS

This illustrates the tokens used in accessing structures. These are used frequently in function calls where the external or internal entry point slot of the function object is referenced. A 'utility' token STRUCT_ACCESS simplifies the core of this exercise and forms the basic primitive for accessing structures. It takes three arguments: an alignment tag describing the alignment of the structure that we are accessing; a pointer to the start of the structure; an offset describing the field that we are trying to access. The token thus takes the pointer to the base structure which is probably modelled as a pointer to void. This is then cast to give it the shape of the structure we are accessing. (This involves a call to the ANSI C API function **~ptr_to_ptr** to turn the void into the alignment tag of the structure that we wish to access). Once in the correct representation, the pointer is then passed to **add_to_pointer** which is a standard ANDF function which can add the field offset to the base address of the structure.

STRUCT_ACCESS is used by the token library in the definition of two tokens: SLOT_READ and SLOT_WRITE. These are generated as required by the compiler.

If we are reading a slot, then SLOT_READ takes the shape of the result in addition to the same three parameters as STRUCT_ACCESS. It calls STRUCT_ACCESS, and then calls CONTENTS on the result of this, telling CONTENTS the shape of the expected result. For a slot write, we do not need to call CONTENTS. SLOT_WRITE does not actually write the slot itself, but produces a value which can be the first parameter of a call to **assign**. (It is possible to assign to slots as well as to any other variable, given the generation of suitable tokens).

## 6.2 Dylan C Runtime Reference

### 6.2.1 The Runtime Primitives

**primitive_allocate** [Function]

*Signature:*

**Z primitive_allocate(int size)**

*Implementation:*

This is the interface to the memory allocator which might be dependent on the garbage collector. It takes a size in bytes as a parameter, and returns some freshly allocated memory which the run-time system knows how to memory-manage.

**primitive_byte_allocate** [Function]

*Signature:*

**Z primitive_byte_allocate(int word-size, int byte-size)**

*Implementation:*

This is built on the same mechanism as **primitive_allocate**, but it is specifically designed for allocating objects which have Dylan slots, but also have a repeated slot of byte-sized elements, such as a byte string, or a byte vector. It takes two parameters, a size in 'words' for the object slots (e.g., one for 'class' and a second for 'size'), followed by the number of bytes for the vector. The value returned from the primitive is the freshly allocated memory making up the string.

**primitive_fill_E_** [Function]

*Signature:*

**Z primitive_fill_E_ (Z storage[], int size, Z value)**

*Implementation:*

(The odd name is a result of name mangling from **primitive-fill!**). This takes a Dylan object (or a pointer to the middle of one), a size, and a value. It inserts the value into as many slots as are specified by *size*.

**primitive_replace_E_** [Function]

*Signature:*

**Z primitive_replace_E_ (Z dst[], Z src[], int size)**

*Implementation:*

(See **primitive_fill_E_** re. name). This copies from the source vector into the destination vector as many values as are specified in the *size* parameter.

**primitive_replace_vector_E_** [Function]

*Signature:*

**Z primitive_replace_vector_E_ (SOV* dest, SOV* source)**

*Implementation:*

This is related to **primitive_replace_E_**, except that the two arguments are guaranteed to be simple object vectors, and they are self-sizing. It takes two parameters, 'dest', and 'source', and the data from 'source' is copied into 'dest'. 'Dest' is returned.

**primitive_allocate_vector** [Function]

*Signature:*

**Z primitive_allocate_vector (int size)**

*Implementation:*

This is related to **primitive_allocate**, except that it takes a 'size' argument, which is the size of repeated slots in a simple object vector (SOV). An object which is big enough to hold the specified indices is allocated, and appropriately initialised, so that the 'class' field shows that it is an SOV, and the 'size' field shows how big it is.

**primitive_copy_vector** [Function]

*Signature:*

**Z primitive_copy_vector(Z vector)**

*Implementation:*

This takes a SOV as a parameter, and allocates a fresh SOV of the same size. It copies all the data that was supplied from the old one to the new one, and returns the new one.

**primitive_initialize_vector_from_buffer** [Function]

*Signature:*

**Z primitive_initialize_vector_from_buffer (SOV * vector, int size, Z* buffer)**

*Implementation:*

This primitive takes a pre-existing vector, and copies data into it from a buffer so as to initialise an SOV. The primitive takes a SOV to be updated, a 'size' parameter (the specified size of the SOV),

and a pointer to a buffer which will supply the necessary data. The class and size values for the new SOV are set, and the data written to the rest of the SOV. The SOV is returned.

**primitive_make_string** [Function]

*Signature:*

**Z primitive_make_string(char * string)**

*Implementation:*

This takes as a parameter a 'C' string with is zero-terminated, and returns a Dylan string with the same data inside it.

**primitive_continue_unwind** [Function]

*Signature:*

**Z primitive_continue_unwind ()**

*Implementation:*

This is used as the last thing to be done at the end of an unwind-protect cleanup. It is responsible for determining why the cleanup is being called, and thus taking appropriate action afterwards.

It handles 2 basic cases:

- a non-local exit
- a normal unwind-protect

In the first case we wish to transfer control back to some other location, but there is a cleanup that needs to be done first. In this case there will be an unwind-protect frame on the stack which contains a marker to identify the target of the non-local exit. Control can thus be transferred, possibly invoking another unwind-protect on the way.

Alternatively, no transfer of control may be required, and unwind-protect can proceed normally. As a result of evaluating our protected forms, the multiple values of these forms are stored in the unwind-protect frame. These values are put back in the multiple values area, and control is returned.

**primitive_nlx** [Function]

*Signature:*

**Z primitive_nlx (Bind_exit_frame* target, SOV* arguments)**

*Implementation:*

This takes two parameters: a bind-exit frame which is put on the stack whenever a bind-exit frame is bound, and an SOV of the multiple values that we wish to return to that bind-exit point. We then step to the bind-exit frame target, while checking to see if there are any intervening unwind-protect frames. If there are, we put the marker for our ultimate destination into the unwind-protect frame

that has been detected on the stack between us and our destination. The multiple values we wish to return are put into the unwind-protect frame. The relevant cleanup code is invoked, and at the end of this a primitive_continue_unwind should be called. This should detect that there is further to go, and insert the multiple values into any intervening frames.

**primitive_inlined_nlx** [Function]

*Signature:*

**Z primitive_inlined_nlx (Bind_exit_frame* target, Z first_argument)**

*Implementation:*

This is similar to **primitive_nlx**, except that it is used when the compiler has been able to gain more information about the circumstances in which the non-local-exit call is happening. In particular it is used when it is possible to in-line the call, so that the multiple values that are being passed are known to be in the multiple values area, rather than having been created as an SOV. An SOV has to be built up from these arguments.

**primitive_make_box** [Function]

*Signature:*

**Z* primitive_make_box(Z object)**

*Implementation:*

A box is a value-cell that is used for closed-over variables which are subject to assignment. The function takes a Dylan object, and returns a value-cell box which contains the object. The compiler deals with the extra level of indirection needed to get the value out of the box.

**primitive_make_environment** [Function]

*Signature:*

**Z* primitive_make_environment(int size, …)**

*Implementation:*

This is the function which makes the vector which is used in a closure. The arguments to this are either boxes, or normal Dylan objects. This takes an argument of ‘size’ for the initial arguments to be closed over, plus the arguments themselves. ‘Size’ arguments are built up into an SOV which is used as an environment.

**primitive_basic_iep_apply** [Function]

*Signature:*

**Z primitive_basic_iep_apply (FN* f, int argument_count, Z a[])**

*Implementation:*

This is used to call internal entry points. It takes three parameters: a Dylan function object (where the iep is stored in a slot), an argument count of the number of arguments that we are passing to the iep, and a vector of all of these arguments. This is a 'basic' IEP apply because is does no more than check the argument count, and call the IEP with the appropriate number of Dylan parameters. It does not bother to set any implementation parameters. Implementation parameters which could be set in by other primitives are 'function', and a 'mlist' (the list of next-methods) . Not all IEPs care about the 'function' or 'mlist' parameters, but when the compiler calls 'primitive_basic_iep_apply', it has to make sure that any necessary 'function' or 'mlist' parameters have been set up.

**primitive_iep_apply** [Function]

*Signature:*

**Z primitive_iep_apply (FN* f, int argument_count, Z a[])**

*Implementation:*

This is closely related to **primitive_basic_iep_apply**. It takes the same number of parameters, but it sets the explicit, implementation-dependent function parameter which is usually set to the first argument, and also sets the 'mlist' argument to 'false'. This is the normal case when a method object is being called directly, rather than as part of a generic function.

**primitive_xep_apply** [Function]

*Signature:*

**Z primitive_xep_apply (FN* f, int argument_count, Z a[])**

*Implementation:*

This is a more usual usage of apply, i.e., the standard Dylan calling convention being invoked by **apply**. It takes three parameters: the Dylan function to be called, the number of arguments being passed, and a vector containing all those arguments. This primitive relates to the external entry point for the function, and guarantees full type checking and argument count checking. This primitive does all that is necessary to conform with the xep calling convention of Dylan: i.e., it sets the 'function' parameter, it sets the argument count, and then calls the XEP for the function.

**xep_ 0 … xep_9** [Function]

*Signature:*

**Z xep_0 (FN* function, int argument_count)**

*Implementation:*

These are the XEP entry-point handlers for those Dylan functions which do not accept optional parameters. Each Dylan function has an external (safe) entry point with full checking. After checking, this calls the internal entry point, which is the most efficient available.

The compiler itself only ever generates code for the internal entry point. Any value put into the external entry point field of an object is a shared value provided by the runtime system. If the function takes no parameters, the value will be ‘xep0’; if it takes a single required parameter it will be ‘xep1’, and so on. There are values available for ‘xep0’ to ‘xep9’. For more than nine required parameters, the next function (below) is used.

**xep** [Function]

*Signature:*

**xep (FN* function, int argument_count, …)**

*Implementation:*

If the function takes more than nine required parameters, then the function will simply be called **xep**, the general function which will work in all such cases. The arguments are passed as ‘varargs’. This function will check the number of arguments, raising an error if it is wrong. It then sets the calling convention for calling the internal entry point. This basically means that the function register is appropriately set, and the implementation ‘mlist’ parameter is set to **#f**.

**optional_xep** [Function]

*Signature:*

**Z optional_xep (FN* function, int argument_count, …)**

*Implementation:*

This function is used as the XEP code for any Dylan function which has optional parameters. In this case, the external entry point conventions do not require the caller to have any knowledge of where the optionals start. The XEP code is thus responsible for separating the code into those which are required parameters, to be passed via the normal machine conventions, and those which are optionals. to be passed as a Dylan SOV. If the function object takes keywords, all the information about which keywords are accepted is stored in the function itself. The vector of optional parameters is scanned by the XEP code to see if any appropriate ones have been supplied. If one is found, then the associated value is taken and used as an implicit parameter to the internal entry point. If a value is not supplied, then a suitable default parameter which is stored inside the function object is passed instead.

**gf_xep_0 … gf_xep_9** [Function]

*Signature:*

**Z gf_xep_0(FN* function, int argument_count)**

*Implementation:*

These primitives are similar to **xep_0** through **xep_9**, but deal with the entry points for generic functions. Generic functions do not require the ‘mlist’ parameter to be set, so a special optimised entry point is provided. These versions are for 0 - 9 required parameters. These functions call the internal entry point.

**gf_xep** [Function]

*Signature:*

**Z gf_xep (FN* function, int argument_count, …)**

*Implementation:*

This primitive is similar to **xep**, but deals with the entry points for generic functions. Generic functions do not require the 'mlist' parameter to be set, so a special optimised entry point is provided. This is the general version for functions which do not take optional arguments. This function calls the internal entry point.

**gf_optional_xep** [Function]

*Signature:*

**Z gf_optional_xep (FN* function, int argument_count, …)**

*Implementation:*

This is used for all generic functions which take optional arguments. This function calls the internal entry point.

# 7. Dylan Features Mapped to ANDF

This chapter takes a range of Dylan features and shows by example how these are mapped onto ANDF. The features detailed here have been chosen as representing the core features of the language, and as such are the foundation upon which the higher-level features of the language were built.

All of the examples are defined in a module **dylan-test** of library **test**, and the ANDF output is name mangled accordingly.

## 7.1 The Sample of Dylan Features

- defining variables
- defining functions
- defining methods of GFs
- evaluating variables
- calling functions
- calling GFs
- if
- multiple-value generation
- multiple-value binding
- bind-exit
- unwind-protect
- closures

## 7.2 The Dylan Features Mapped to ANDF

### 7.2.1 Defining Variables

First, consider the a simple case of variable definition:

```
define variable *simple* = 1;
```

This generates the following ANDF, to declare and define the variable. Use is made of the token INTEGER, to provide the tagged Dylan integer representation of 1.

```
(make_var_tagdec dylan_testXtestX_T_simple_T_ - pz )
(make_var_tagdef dylan_testXtestX_T_simple_T_ (INTEGER 1))
```

A more complex case involves the variable's initial value being the result of a dynamic function call, such as the following to define a single element list.

```
define variable *dynamic* = make(<list>, size: 1);
```

Each compiled dylan library has a top level form function which performs any initialisation the library requires before any of it's facilities are utilised. The top level form function performs initialisation of any variables whose initial values are dynamically assigned. So for the above example, the following declaration and definition are generated at the top level:

```
(make_var_tagdec dylan_testXtestX_T_dynamic_T_ - pz )
(make_var_tagdef dylan_testXtestX_T_dynamic_T_ (OT dylanXinternalX_P_unbound))
```

And then the top level form initialisation function will contain the following:

```
(ASSIGN
 (OT dylan_testXtestX_T_dynamic_T_)
 (apply_proc pz (OT dylanXinternalXmake_9_IEP)
        (LVAL dylanXinternalX_L_list_G_)
        (INTEGER 1)
        (LVAL dylanXinternalX_P_false)
        (LVAL LIT_1) ))
```

This is assigning the result of the call to ***dynamic***. The function being called is the **make** method for **<list>**. The compiler selected the appropriate method and arranged to call it via its IEP. The first parameter is the class we wish to make an instance of, the second is the size of the list, the third specifies a value to fill the elements with (in this case false, the default, means don't fill), and the final parameter is the **#rest** parameters list. To see what this represents here, we need to see the definition of the statically built literal **LIT_1**:

```
(local make_var_tagdec LIT_1 - (nof 2 pz) )
(local make_var_tagdef LIT_1
 (make_nof
  (LVAL dylanXinternalX_L_simple_object_vector_G__wrapper)
  (INTEGER 0) | ) )
```

This represents a zero element list, as there are no **#rest** parameters passed in this case.

### 7.2.2 Defining Functions

This defines a single method not associated with any generic function.

```
define constant constant-function
 = method (a)
    a
  end;
```

The function is implemented by the following proc, which simply sets the global variable representing the number of values returned to 1 (using the **_1** shorthand for a raw integer) and then returns its only argument.

```
(make_id_tagdef dylan_testXtestXconstant_function_IEP
 (make_proc pz
  pz - a
 | -
 (sequence
 (labelled block_intro | (goto block_intro)
 (sequence (make_null_ptr (alignment top)) # block_intro:
  (ASSIGN (OT _P_number_values) _1)
  (return (RVAL a) )
 ))
ENDPROC)))
```

This is the statically built literal function object that represents the function:

```
(make_var_tagdec dylan_testXtestXconstant_function - (nof 8 pz) )
(make_var_tagdef dylan_testXtestXconstant_function
 (make_nof
  (LVAL dylanXinternalX_L_method_G__wrapper)
  (INTEGER 1)
  (FVAL xep_1)
  (FVAL dylan_testXtestXconstant_function_IEP)
  (LVAL dylanXinternalX_P_false)
  (LVAL LIT_7)
  (FVAL dylan_testXtestXconstant_function_IEP) | ) )
```

These slots represent the class of object (a method in this case), some function properties, the XEP, the IEP, the environment, method specialisers and the MEP.

The specialisers are represented by **LIT_7**, a single element vector indicating that the method expects a parameter of class <**object**>.

```
(local make_var_tagdec LIT_7 - (nof 3 pz) )
(local make_var_tagdef LIT_7
 (make_nof
  (LVAL dylanXinternalX_L_simple_object_vector_G__wrapper)
  (INTEGER 1)
  (LVAL dylanXinternalX_L_object_G_) | ) )
```

### 7.2.3 Defining Methods of Generic Functions

In Dylan, calling **define method** will automatically create the corresponding GF if it doesn't exist already. The two statements below will generate the **add-1** generic function, which takes a single argument, and two methods defined for that GF, one specialised on **<integer>**, the other on **<byte-string>**.

```
define method add-1 (a :: <integer>) => (res :: <integer>)
   a + 1;
end method;

define method add-1 (a :: <byte-string>) => (res :: <byte-string>)
  concatenate(a, "1");
end method;
```

From this, we get the GF object, 2 method objects and the 2 method bodies.

The GF object contains slots for such things as the XEP and IEP of the function. Slot number 7 here (**LIT_13**) is a pointer to a list of methods defined on the generic function. The code below shows the list implemented as a sequence of '**<pair>**s', which point to **dylan_testXtestXadd_1_1** and **dylan_testXtestXadd_1_0**, the method objects.

```
(make_var_tagdef dylan_testXtestXadd_1
  (make_nof
    (LVAL dylanXinternalX_L_generic_function_G__wrapper)
    (INTEGER 1)
    (LVAL gf_xep_1)
    (LVAL dylanXinternalXdiscriminator_1_IEP)
    (LVAL dylanXinternalX_P_false)
    (LVAL dylanXinternalX_P_false)
    (LVAL LIT_13)
    (LVAL dylanXinternalX_P_false) | ) )
(local make_var_tagdef LIT_13
  (make_nof
    (LVAL dylanXinternalX_L_pair_G__wrapper)
    (LVAL dylan_testXtestXadd_1_1)
    (LVAL LIT_14) | ) )
(local make_var_tagdef LIT_14
  (make_nof
    (LVAL dylanXinternalX_L_pair_G__wrapper)
    (LVAL dylan_testXtestXadd_1_0)
    (LVAL dylanXinternalX_P_empty_list) | ) )
```

The ANDF to build the '**dylan_testXtestXadd_1_1**' and '**dylan_testXtestXadd_1_0**' methods is similar to that for '**dylan_testXtestXconstant_function**' and so is not further documented here.

The method body of the **add_1** method specialised on **<integer>** includes a call to the generic function **PL** (i.e., +). The return value, **RET_0**, is assigned the result of calling the generic function. The function being called is the result of the (SLOT_READ... token. In this case it will expand into an access on the XEP field of the GF object for +. The XEP call takes the GF object itself and the number of parameters being passed, in addition to the arguments themselves.

```
(make_id_tagdef dylan_testXtestXadd_1_0_IEP
 (make_proc pz
  pz - a
 | -
 (sequence
 (variable - RET_0 (OT dylanXinternalX_P_unbound)
  labelled block_intro | (goto block_intro)
  (sequence (make_null_ptr (alignment top)) # block_intro:
   (ASSIGN
    (OT RET_0)
    (apply_proc pz (SLOT_READ proc AL_FN (OT dylanXinternalX_PL_) FN.xep)
     (OT dylanXinternalX_PL_)
     _2
     (RVAL a)
     (INTEGER 1) ))
  (return (RVAL RET_0))
 ))
)ENDPROC)))
```

### 7.2.4 Evaluating Variables and Calling Functions

```
format-out("The value of *simple* is %s\n", *simple*);
```

This evaluates to a call to the IEP of the function. **LIT_2** is a dylan **<byte-string>** object representing the text, **BUF_0** is a **<simple-object-vector>** containing the extra arguments — just ***simple*** in this case.

```
(apply_proc pz (OT simple_streamsXsimple_streamsXformat_out_0_IEP)
        (LVAL LIT_2)
        (LVAL BUF_0) )
```

### 7.2.5 Calling Generic Functions

```
add-1("98765432");
```

The compiler is able to optimise out the generic dispatch in this case, so the call looks very much like the above. **LIT_3** is the string **"98765432"** and **BUF_1** is a single element SOV holding the string **"1"**.

```
(apply_proc pz (OT dylanXinternalXconcatenate_0_IEP)
        (LVAL LIT_3)
        (LVAL BUF_1) )
```

### 7.2.6 if

```
define method if-test (a)
 if (a)
  do-something()
 else
  do-something-else()
```

```
  end
end method;
```

The functionality of **if** in the ANDF below is handled by the GOTO_IF_TRUE token, which branches to the **do_something** call if **a** is true. Otherwise control continues within **block_intro**, and makes the call to **do_something_else**.

```
(make_id_tagdef dylan_testXtestXif_test_0_IEP
 (make_proc pz
  pz - a
 | -
 (sequence
 (variable - RET_0 (OT dylanXinternalX_P_unbound)
 (labelled block_intro LBL_0 LBL_1 | (goto block_intro)
  (sequence (make_null_ptr (alignment top)) # block_intro:
   (GOTO_IF_TRUE (RVAL a)  LBL_1)
   (ASSIGN (OT RET_0)
    (apply_proc pz
     (SLOT_READ proc AL_FN (OT dylan_testXtestXdo_something_else) FN.xep)
     (OT dylan_testXtestXdo_something_else)
     _0))
   (goto LBL_0))
  (sequence (make_null_ptr (alignment top)) # LBL_0:
   (return (RVAL RET_0))
   (goto LBL_1))
  (sequence (make_null_ptr (alignment top)) # LBL_1:
   (ASSIGN (OT RET_0)
    (apply_proc pz
     (SLOT_READ proc AL_FN (OT dylan_testXtestXdo_something) FN.xep)
     (OT dylan_testXtestXdo_something)
      _0))
   (goto LBL_0)
  ))
 )ENDPROC)))
```

### 7.2.7 Multiple-Value Generation

To specify multiple return values, a Dylan method calls **values**. Thus, a method body ending with:

```
values(1, 2, 3);
```

is returning 3 integers. This doesn't compile into anything particularly special:

```
(apply_proc pz
 (SLOT_READ proc AL_FN (OT dylanXinternalXvalues) FN.xep)
 (OT dylanXinternalXvalues)
 _3
 (INTEGER 1)
 (INTEGER 2)
 (INTEGER 3))
```

Which is a call to the external entry point of the values GF, passed three parameters, which are the Dylan integers 1, 2 and 3.

### 7.2.8 Multiple-Value Binding

This form binds the results of a call to values to the lexical variable results, as a sequence. The implementation chooses to represent this sequence as a simple object vector.

```
begin
  let (#rest results) = values(1, 2, 3);
  format-out("The results are: %S\n", results);
end;
```

We have seen examples of calling **format-out** already, so let's just look at the code in the top level form that implements the let binding of multiple values.

There are references to some temporary local variables in the code below, these are defined by the **top_level_form** (or, more generally, the function that contains the let binding), as follows:

```
(variable - ARG_0 (OT dylanXinternalX_P_unbound)
(variable - o_0 (OT dylanXinternalX_P_unbound)
(variable - results_1M (OT dylanXinternalX_P_unbound)
```

There are three stages involved:

```
(ASSIGN (OT ARG_0)
  (apply_proc pz
    (SLOT_READ proc AL_FN (OT dylanXinternalXvalues) FN.xep)
    (OT dylanXinternalXvalues)
    _3
    (INTEGER 1)
    (INTEGER 2)
    (INTEGER 3)))
```

This simply makes the call to **values**, and assigns the result to a local temporary variable, **ARG_0**.

```
(ASSIGN (OT results_0M)
  (apply_proc pz (OT dylanXinternalXprimitive_stack_vector_remaining_values) _0))
```

**results_0M** then gets allocated as a SOV containing the values at the end of the values buffer. The argument to **primitive_stack_vector_remaining_values** tells it how many buffer elements to skip before copying the values into the new vector. In this case, all the values are being assigned to **results** so we don't want to skip any of them. If the Dylan statement read: **let (result #rest results) = values(1, 2, 3);** then the parameter would be 1, so we only copy 2 and 3 into the **results** vector.

### 7.2.9 bind-exit

This piece of Dylan will cause the block to call **inner-fn**, which might perform a non-local exit and return a value of 22.

```
block (return)
  inner-fn(return, 22);
end;
```

Most of the mechanics of this process are contained within the runtime library, so the compiler output for the above is mostly calls to other routines.

The code below first stores the current unwind-protect state in a field of **NLX_0**, which is a local variable used to hold information on the bind exit frame. Then a call is made to the ANSI API **setjmp** routine, to store the current execution state, in another field of the **NLX_0** structure. If this returns 0, we're calling **setjmp**, so control continues into the body of the block. If control is being returned to this point via a call to **longjmp**, we'll get a non-zero value, and a branch is made to **LBL_0**. The body of the block, in this case, is just the call to the **inner-fn**. The first parameter to **inner-fn**, is the result of calling **%make-nlx-closure(NLX_0)** and the other parameter is 22, the return value specified in the Dylan code fragment. In essence, the nlx-closure will copy the return values that it's given into the **return_values** slot of the **NLX_0** object, and then perform a **longjmp**, which will return control to the **setjmp** part of the code. This time, the branch will be taken, and all that remains is the call to **set-values!** to copy the return values from the bind exit frame into the **%values** buffer.

```
(variable - NLX_0 (make_value BEF_OFFSET) ...
(ASSIGN
  (SLOT_WRITE AL_BEF (OT NLX_0) BEF.present_unwind_protect_frame)
  (RVAL _P_current_unwind_protect_frame))
(PRIM_GOTO_IF_TRUE
  (ansi.setjmp.setjmp (add_to_ptr (OT NLX_0) BEF.destination))
  LBL_0)
(apply_proc pz (SLOT_READ proc AL_FN (OT dylan_testXtestXinner_fn) FN.xep)
  (OT dylan_testXtestXinner_fn)
  _2
  (apply_proc pz (OT dylanXinternalX_P_make_nlx_closure_IEP)
    (LVAL NLX_0))
    (INTEGER 22))
(goto LBL_1))
(sequence (make_null_ptr (alignment top)) # LBL_0:
  (apply_proc pz (OT dylanXinternalXset_values_E__IEP)
    (~to_ptr_void (alignment pz) (add_to_ptr (OT NLX_0) BEF.return_values)))
  (goto LBL_1))
    (sequence (make_null_ptr (alignment top)) # LBL_1:
```

### 7.2.10 unwind-protect

Dylan guarantees that the cleanup clause in the block statement below, will be executed, whether the block performs a non-local exit or not.

```
block (return)
  inner-fn(return, 22);
cleanup
  format-out("Cleanup happening\n");
end;
```

This is very similar to the previous case, but in addition to the bind exit frame for the block, there is also an unwind-protect frame. If no nlx is performed, control would transfer from then end of the body of the block straight to the cleanup clause (the block represented by **LBL_1** to the block represented by **LBL_2**). When a nlx does occur, the cleanup clause will be called by code in the runtime system, because the unwind-protect frame has been pushed onto the list of active bind exit frames. The call to **continue_unwind()** passes control back to the runtime system, which can then **longjmp** to the next active nlx frame, which in this case will be the bind exit frame representing the block.

```
(variable - NLX_1 (make_value BEF_OFFSET) ...
(variable - NLX_2 (make_value UPF_OFFSET) ...
(sequence (make_null_ptr (alignment top)) # LBL_1:
 (ASSIGN
  (SLOT_WRITE AL_BEF (OT NLX_1) BEF.present_unwind_protect_frame)
  (RVAL _P_current_unwind_protect_frame))
 (PRIM_GOTO_IF_TRUE
  (ansi.setjmp.setjmp (add_to_ptr (OT NLX_1) BEF.destination))
  LBL_4)
 (ASSIGN
  (SLOT_WRITE AL_UPF (OT NLX_2) UPF.ultimate_destination)
  _0)
 (ASSIGN
  (SLOT_WRITE AL_UPF (OT NLX_2) UPF.previous_unwind_protect_frame)
  (RVAL _P_current_unwind_protect_frame))
 (ASSIGN
  (OT _P_current_unwind_protect_frame)
  (LVAL NLX_2))
 (PRIM_GOTO_IF_TRUE
  (ansi.setjmp.setjmp (SLOT_READ jmp_buf_shape AL_UPF (OT NLX_2)
UPF.destination))
  LBL_5)
 (ASSIGN
  (OT ARG_1)
  (apply_proc pz (SLOT_READ proc AL_FN (OT dylan_testXtestXinner_fn) FN.xep)
  (OT dylan_testXtestXinner_fn)
  _2
  (apply_proc pz (OT dylanXinternalX_P_make_nlx_closure_IEP)
   (LVAL NLX_1))
   (INTEGER 22) ))
 (goto LBL_2))
(sequence (make_null_ptr (alignment top)) # LBL_2:
 (apply_proc pz (OT simple_formatXsimple_formatXformat_out_0_IEP)
  (LVAL LIT_4)
  (LVAL LIT_1) )
 (apply_proc pz (OT dylanXinternalXprimitive_continue_unwind))
 (goto LBL_3))
(sequence (make_null_ptr (alignment top)) # LBL_3:
```

The following is top-level code following the end of the block:

```
(sequence (make_null_ptr (alignment top)) # LBL_4:
 (apply_proc pz (OT dylanXinternalXset_values_E__IEP)
  (~to_ptr_void (alignment pz) (add_to_ptr (OT NLX_1) BEF.return_values)))
 (goto LBL_5))
(sequence (make_null_ptr (alignment top)) # LBL_5:
 (goto LBL_2))
```

### 7.2.11 Closures

```
define method make-add-n (n :: <integer>) => (closure :: <function>)
  method (x :: <integer>) => (res :: <integer>)
    x + n;
  end method;
end method;
```

Calling **make-add-n(2)** returns a method that adds two to its argument. Let's consider the ANDF in reverse order.

The proc below is the implementation of the inner method above. It calls the + generic function passing both its sole argument, and the first data item in its environment, which will correspond to **n**. This method is statically built as a *prototype*, from which the closure will be copied when its environment is known.

```
(local make_id_tagdef dylan_testXtestXanonymous_3_IEP_0
 (make_proc pz
  pz - x
 | -
 (sequence
 (variable - environment (SLOT_READ pz AL_FN (RVAL _P_function) FN.environment)
 (variable - RET_0 (OT dylanXinternalX_P_unbound)
 (labelled block_intro | (goto block_intro)
 (sequence (make_null_ptr (alignment top)) # block_intro:
  (ASSIGN
   (OT RET_0)
   (apply_proc pz (SLOT_READ proc AL_FN (OT dylanXinternalX_PL_) FN.xep)
    (OT dylanXinternalX_PL_)
    _2
    (RVAL x)
    (READ_VECTOR_ELEMENT (OT environment) (INT_ADD _0 _2))))
  (return (RVAL RET_0))
 ))
))ENDPROC)))
```

To generate a method that is closed over **n**, we create a new method object by copying the prototype for the closure, with its environment initialised to the appropriate value.

```
(make_id_tagdef dylan_testXtestXmake_add_n_0_IEP
 (make_proc pz
  pz - n
 | -
 (sequence
```

```
  (labelled block_intro | (goto block_intro)
  (sequence (make_null_ptr (alignment top)) # block_intro:
   (ASSIGN (OT _P_number_values) _1)
   (return
    (apply_proc pz (OT dylanXinternalX_P_copy_method_using_environment_IEP)
     (LVAL dylan_testXtestXanonymous_3_0)
     (apply_proc pz (OT dylanXinternalXprimitive_make_environment)
      _1
      (RVAL n))))
   ))
 ENDPROC)))
```

# 8. Conclusions

As noted in chapter 1, this deliverable and its associated software demonstration represent the culmination of Harlequin's efforts in Workpackage 4 of the GLUE project. The original goal was to establish the efficacy of ANDF's support for the compilation of advanced languages by developing a Dylan producer. This included the development of the appropriate load-time and runtime support for advanced languages based on ANDF hooks.

The work reported here represents the fourth and final stage of development, and so we shall again briefly describe the initial three stages. In the first stage Harlequin used their background in existing implementations of development environments to identify the features which require support from ANDF to allow both the efficient compilation of Dylan and the efficient implementation of its runtime system. The second stage took the results of this work and began the incremental development of a Dylan producer. In the first instance, a small subset of Dylan was implemented, leading to an initial evaluation of ANDF's ability to support the relevant language features and runtime system. The third stage focused on inter-language working. Software components written in ANDF-supported languages were combined with Dylan producer output, with the aim of ensuring that no performance penalty was imposed on the other ANDF-based components.

This deliverable reports the work of the fourth stage, the incremental extension of the implementation towards a prototype Dylan system. The incremental approach allowed the impact of various language features on the runtime to be studied, and also allowed the controlled introduction of a range of optimisation techniques used by Harlequin.

In completing this stage we wished to provide a degree of algorithmic interchangeability in certain important subsystems of the implementation such as the garbage collector, the function calling mechanism, and the manipulation of tagged data. The approach of abstracting these important subsystems was taken in order to allow rapid experimentation with different implementations. By moving as many of the details of the implementation as possible into the Dylan token library we were able to significantly shorten the "edit-compile-evaluate" cycle for experimentation with implementation techniques. Changes to token definitions became effective as soon as the modified token library was recompiled, and the previously produced test-suite and benchmark programs had only to be re-installed with the modified token library before the new implementation strategy could be evaluated.

The Dylan token library has been carefully designed to support the desired combinations of runtime activity, while keeping the externally visible set of tokens simple. This is desirable as a simpler token library

helps to keep the compiler back-end relatively simple: complexity is moved out of the compiler and into the token library. A manageable token library also provides the possibility of future extensibility without the need for extensive modification to the compiler. In the ideal case, only the definitions in the token library would need to be changed.

Our primary conclusion is that this approach has been successful: our Dylan producer for the Solaris platform can be compiled and run to demonstrate a complete implementation of Dylan with interface to C. We have thus shown that ANDF can be used to support dynamic languages along with the use of garbage collection techniques. The compiler and runtime structures that implement this are detailed in the preceding chapters. The techniques used to implement garbage collection have been described in an earlier deliverable, TR 4.2.2a, “Initial Evaluation of TDF support for Garbage Collection” [Mann93a].

A further conclusion is that Dylan's features can be implemented with minimal additional support from core ANDF. Support for the more complex features of the language is provided by tokens which are expanded at installation time. This has the benefit that the implementation details can be platform dependent, enabling the best use to be made of each platform. One drawback, however, is that ANDF is not a complete portability solution for Dylan, as both a runtime system and a token library must first be provided for each new platform.

With the one exception of function calls, there is no reason to believe that any of the features of Dylan need be implemented in a significantly less efficient manner when compiled with ANDF compared with a carefully crafted native-code approach. Harlequin's Dylan implementation depends heavily on tail-call optimisation for speed. That optimisation is now possible with ANDF, via the callee parameter extensions — but these have not been tested with Dylan.

We had hoped to experiment with callee parameters, but we have not been able to do this thoroughly due to difficulties we encountered while using the ANDF tools to integrate the producer components. Our developers had to spend a great deal of time in investigating and overcoming these difficulties, due to the restricted feedback given by the ANDF tools that were available. One particular problem was that there was no debugger support for callee parameters, and also code compiled to use callee parameters was not testable from code compiled with C compatible calls. Ultimately the project deadlines ensured that no further experiments could take place.

At the time of writing, these same pressures have also prevented us from formally addressing performance issues. However, we believe that any performance penalties associated with ANDF are small in comparison with the gains that we expect from additional work on the optimization phase of the Dylan compiler. We are optimistic that aplications written in Dylan will eventually perform with comparable efficiency to applications written in traditional static languages.

Previous work [MG95] has shown that Dylan can be used with ANDF to build portable software components which are well-suited to inter-language working. The only restrictions which are required for this are those considered to be normal for good component design.

We may thus conclude that, given a runtime system and a token library for a target platform, ANDF can support portable software and component software written in Dylan. This software may make use of advanced language features such as dynamism and garbage collection, as demonstrated by our prototype Dylan producer. The incremental approach adopted has been highly successful in producing an efficient and extensible product which can form a part of Harlequin’s wider commercial developments.

# 9. References and Bibliography

[BW88] Boehm, H-J., Weiser, M., “Garbage Collection in an Uncooperative Environment”, Software Practice and Experience, vol. 18, #9, September 1988.

[Mann93a] Mann, T., “Initial Evaluation of TDF Support for Garbage Collection”, GLUE Deliverable 4.2.2a, Harlequin Ltd., 1993.

[MG95] Mann, T., Green, S., “Report on Inter-Language Working in ANDF”, GLUE Deliverable 4.2.3, Harlequin Ltd., 1995.