Function Specification

Overview

The complete functional specification of the Co-Dfns compiler can be found in this document. The compiler is designed and specified using a black-box method, specifying the behavior in terms of input and stimulus histories and responses. Because we are writing a compiler and not some other system, the actual user facing toggles on the system are surprisingly small, despite the large potential for behavior. This is because the majority of input comes in the forms of programs. Therefore, in specification, it is critical to specify not only the normal user behaviors, but also the behaviors on critical abstractions of program input and form.

The compiler itself also has a tendency towards sparse responses. In traditional program design, the responses of the system would come in the form of textual, GUI, or hardware responses that had a very user visible form. In contrast, the majority of software "states" that the user can consider, when correct input is entered, should respond with no output and no response. These states are there conceptually or abstractly, rather than displaying specific outputs whenever they are entered. They cannot be elided from the function specification, however, because when input is *not* correct, then these states represent the set of possible error responses and the types of reporting that will be given in the cases where the system does not receive the intended input. Thus, these states can be considered null responses unless errors need to be considered or handled. Indeed, the majority of user visible responses in the compiler take the form of error handling responses, and the compiler is at its most quiet when all things are going as planned.

The approach to black box specification here will take this form and the nature of the visible interface of a compiler into account. We encode the input of the system at an abstraction level that lets us encode the appropriate error responses of the system and all *potential* user visible behavior of the system in response to any user alterable or user derived input, even though the majority of the time, the compiler might be quite silent. This requires an abstraction level that allows us to talk about the structure and form of the programs that are given as input into the program, but because of the nature of program text, we have chosen an abstraction level only deep enough to accurately reflect all of the possible compiler responses, and no deeper. Entering in any more precisely would defeat the purpose of the black box abstraction and would complicate the specification process beyond feasibility.

In this same vein, the abstraction of the compiler responses goes only so far as to accurately reflect the paths that may lead to semantically valuable user visible changes in response to new inputs. Responses, therefore, are equally abstract, and represent only the sufficiently clear class of errors and conceptual states that reflect the useful and meaningful external behavior of the system.

Despite the relative abstractness of this specification, however, normal, standard black box abstraction techniques and specification methods provide the same level of rigor and usefulness to the task. The methods and approaches themselves are as little refined or modified as possible, in order to make these specifications as easy to understand and as rigorous as possible, without requiring a significant programmer overhead for developing a mathematical semantics that far exceeds the desired scope of the compiler project.

All behavior here is dictated by the *Software Requirements* which in turn have the *Programmer's Guide and Language Reference* for Dyalog APL as an implicit dependency. Likewise, this specification assumes and relies on the language reference as an implicit part of this specification and will clarify behaviors not given complete treatment here. Complete treatment may not be granted here in the case that the behavior is adequately documented in the language reference.

Software Boundaries

There are two primary sources of user input. The user will first call the compiler with a given program to compile, which represents the first external input. Secondly, the compiled namespace that is created as a result will be called repeatedly and used as its own thing. This represents the second potential input to the system. These are distinct because the input given to the second, and the behavior that is appropriate to it is entirely dependent on the input that is given as the program input. Thus, it is impossible to speak directly about the kinds of inputs and their responses for the second types of input, except very broadly. Instead, the semantics are encoded in the first input, and as long as the semantics are preserved in the compiler, then it will work as intended.

Thus, the main sources of input that will be considered are the programs that are given, rather than on the inputs that will be given to the programs described or compiled. This may or may not accurately reflect how the code is used in practice. Indeed, we normally expected to compile a program once and then run it multiple times. However, most of the more useful information comes from the program input, rather than the input to the compiled program.

As a final note, we have chosen to view the LLVM as an external software artifact over which we have no control. Thus, it should be considered an external artifact. It will receive inputs from the compiler and return resulting responses that we will use to create the final compiled object.

Table 1. Summary of External Entities

Name	Notes
Source Code	Primary input source from the user; specifies semantics and intended behavior of module behavior.

Name	Notes
Compiled Module	Behavior is determined completely by the input source code, and cannot be readily specified outside of a specific input beyond summary behavior.
LLVM	External software used to build the module, consumer of inputs from the compiler and producer of compiled modules.

Stimuli and Responses

We divide the set of stimuli and responses into two sets, corresponding to the source code and the module. We will not consider the specification of the LLVM interactions, as these are internal to the system and should not be user visible. Many of these stimuli are abstract stimuli based on multiple real stimuli. Others, such as those in the module set, are abstract because we cannot encode a specific stimuli set absent a specific source input.

The stimuli for the Source Input correspond very closely to those that might be used for the tokenization of program inputs. We abstractly consider the input history for source input as first an indication of which external function was called. Then the stream of argument values must be considered. This encodes the arguments received by the `CoDfns.Fix` function. We choose to keep the filename argument as a single unit, but the source input should be separated into its various token elements corresponding to the incoming token stream that the parser will have to deal with, at least, conceptually. In particular, we are abstracting the consuming of whitespace that might appear throughout the source input.

In addition to the raw token stimuli, the processing of source code is an inherently recursive process, and thus we have a set of *recursive stimuli* which represent terms which we will define by enumeration, but that themselves are used within enumerations, possibly within itself. This allows us to encode recursive properties without bringing the recursive problems into the enumerations themselves.

Each recursive stimuli and the top-level source input have a subset of the total possible stimuli to accept as valid stimuli. All other stimuli are implicitly illegal.

Table 2. Recursive Stimuli for Source Input

Symbol	Name	Meaning
E	Expression	Any expression that evaluates to an array value
Fn	Ambivelant Function	An ambivelant user-defined function
Fnm	Monadic User Operator	An user-defined monadic operator

Symbol	Name	Meaning
Fnd	Dyadic User Operator	An user-defined dyadic operator
Fe	Function Expression	An expression evaluating to a function value

All of the function stimuli may be enumerated as one, since they have the same syntax at this abstract level. When enumerating a recursive stimuli, we allow potentially any error response, as well as illegla, wait, and okay. The wait response indicates the sequence as yet is not a valid stimuli, but that it may yet become a valid sequence. The okay response indicates that the sequence as is constitutes a valid sequence, but need not be final or unextendable.

As a rule, during enumeration, one should consider the use of a recursive stimuli illegal unless it is used to capture some nested property, or when it is used at the top-level, where no opportunity for non-termination exists. When it is used, one must carefully mark any sequence at the same level that may be a prefix of the enumeration of the recursive stimuli as subsumed by said stimuli. A prefix must have the same response to be a prefix. If a stimuli might pair up with another token, they cannot cross enumeration depths or levels, but much pair against one another in the same level. That is, a top-level (may not be closed in a recursive stimuli. The intent is to make each enumeration as self-contained as possible, and to make sure that nested recursion other than tail recursion is marked by the use of recursive stimuli, rather than trying to handle that recursion through enumeration.

Table 3. Stimuli for Source Input

Symbol	Name	Meaning
{	Left Brace	A left brace token
}	Unbalanced Right Brace	A right brace token
[	Left Bracket	A left bracket token
]	Right Bracket	A right bracket token
(	Left Parenthesis	A left parenthesis token
)	Right Parenthesis	A right parenthesis token
;	Index Separator	The index separator for bracket indexing
:	Conditional	The conditional token
::	Error Guard	The error guard token
⋄	Statement Separator	Statement separtor token
←	Assignment	An assignment token
⍣	Power Operator	The Power Operator
Break	Interrupt/Break	User signalled interrupt

Symbol	Name	Meaning
D	Dyadic Primitive	A primitive function that can be called dyadically.
Eot	End of Transmission	The end of the input to the compiler
Fix	Call `Fix`	A call to the `Fix` function, arguments to follow.
Fnb	Bad Filename	A pathname that is somehow an invalid syntax or otherwise invalid to be used as a pathname.
Fne	Empty Filename	A filename/pathname that specifies a file that does not yet exist.
Fnf	Found Filename	A filename that specifies a file that already exists in the filesystem.
Lle	LLVM Error	Any LLVM derived error
Lls	LLVM Success	Any LLVM derived success
M	Monadic Primitive	A primtive function that can be called monadically.
N	Literal Number	A valid, literal number
Nl	Newline	A newline character of some sort
Nse	Namespace End	The ending token for a namespace script, usually `:EndNamespace`.
Nss	Namespace Start	The starting token for a namespace script, usually `:Namespace`.
Om	Monadic Operator	An operator that takes a single functional argument
Od	Dyadic Operator	An operator that takes two functional arguments
S	String	A literal character array string
Va	Array Variable	A variable bound to an array
Vi	Illegal Variable	A variable occurring in an illegal context
Vf	Function Variable	A variable bound to a function
Vom	Monadic Operator Variable	A variable bound to a monadic operator
Vod	Dyadic Operator Variable	A variable bound to a dyadic operator
Vu	Unbound Variable	A variable that has not been bound

Each of the enumeration targets has a set of stimuli that are valid. Alll other stimuli are illegal. These are chosen because any possible occurance of another stimuli in the

enumeration ought to be subsumed by one of the other recursive stimuli. This helps to quell any complexity that may occur in enumerating everything out long hand.

Table 4.

Enumeration	Stimuli Set
Top-level	⋄ Break Eot Fix Fnb Fne Fnf Lle Lls Nl E Fe Fnm Fnd Nse Nss
Expressions	[ ] ( ) ; ← ⍤ Break N S Va Vi Vu E Fe
Functions	{ } : :: ⋄ ← Break Nl Va Vi Vf Vom Vod Vu E Fe Fnm Fnd
Func. Expr.	( ) ← ⍤ Break D M Om Od Vi Vf Vom Vod Vu E Fe Fn Fnm Fnd

We have chosen to encode our compiler responses based primarily on the class of the response. In cases of success, we have a single response; all our other responses classify various types of error cases. We choose to go no further than is necessary to distinguish user visible errors types. We do not include source input location in our error responses, but it is assumed to exist in the output if reasonably feasible. We further assume that all intermediate internal states not producing user visible output will have no explicit output responses, despite representing the majority of states in the system. Each error state corresponds to a specific error code reported by the Dyalog interpreter. We include only the error codes produced by the compiler and not errors occuring only at runtime.

Table 5. Responses for Source Input

Code	Name	Meaning
11	Domain Error	Indicates compiler detected domain error
22	File Name Error	When a file matching the pathname for the shared object exists already
34	File System No Space	Attempting a file operation failed because of insufficient space
3	Index Error	Compiler detected an out of bounds index operation
99	Internal Error	Internal system error; may indicate LLVM error
1003	Interrupt	Received a system interrupt indicating an immediate exit
5	Length Error	Compiler has detected a shape mismatch but not a rank error
10	Limit Error	A system limitation has been encountered
16	Nonce Error	Unimplemented feature reserved for future use

Code	Name	Meaning
4	Rank Error	Compiler has detected a rank error of an argument
2	Syntax Error	Compiler has encountered a line that is not a meaningful statement
6	Value Error	The compiler has found a reference to an unbound variable or a function call returning no result where one was expected
1	Ws Full	The compiler has run out of memory
N/A	Namespace	Indicates a successful execution of the compiler and the return of a semantically equivalent namespace

After a module is compiled, it is usually invoked and executed in various ways. We separate compilation from invocation so that we may enumerate their sequence histories separately, but also because the stimuli are encoded so differently. Most Source Input stimuli are rather close to some specific concrete token that is not defined in terms of anything else, but the corresponding concrete stimuli for a Module Invocation stimulus is always dependent on a specific module. Very few, if any, stimuli will be very concrete. Each stimulus history should have few tokens, as we are representing function calls. We note that all functions are ambivalent when written in Co-Dfns.

Table 6. Stimuli for Module Invocation

Symbol	Name	Description
Fv	Valid Function	A valid reference to a function in the module
Var	Bound Variable	A valid reference to an array in the module
Ub	Unbound reference	A reference to an unbound variable in the module
In	Valid input	Input to a function that will not lead to a runtime error
Err	Erroneous input	Input to a function that will lead to an unguarded runtime error
Call	Function call	Either a monadic or dyadic call to a module function

Our responses when calling or referencing into a compiled module are much more abstract than our responses for the compiler, particularly so for the error responses. This is necessary simply because we cannot know ahead of time what inputs will generate what errors. Instead, we make our responses very abstract. We assume that when actually programmed, the runtime errors will correspond to the

appropriate code and signal. The same goes for correct, or valid, input. We must abstract away until we can only say that the output is equivalent to the result of the same function interpreted instead of a fixed or actual value.

Table 7. Responses for Module Invocation

Name	Description
Value	A value returned by module that is equivalent ot the value returned by an equivalent interpreted module invocation
Value Error	An error signalled when a reference to an unbound variable occurs.
Error	Any runtime error signalled by erroneous input to a module invocation. Must be the same error as would be signalled by an equivalent interpreted module invocation.

Sequence Enumeration

Derived Requirements

Canonical Sequence Analysis

Specification Functions

Black Box Definitions