The Co-dfns Compiler

Aaron W. Hsu

August 11, 2022

Contents

1 Introduction

1.1 How to Read a WEB

2 User's Guide

3 Co-dfns Architecture

This section describes the "big picture" parts of the Co-dfns compiler. The intent here is to try to show how all of the various moving parts of the compiler fit together, to provide a sort of road map that will give you a precise plan for understanding how the various components affect one another. One of the most important things to understand in any compiler is the net effect a local change in the code can have on the rest of the system, so I hope that this section will help to clarify this.

The design of the Co-dfns compiler is one of austerity and minimalism. My intent is, was, and hopefully shall remain that of producing an exceptionally clear design that avoids or eliminates unnecessary code and complexity within the design. I attack this problem in many ways, but I primarily attempt to do this by both reducing the size of the code surface in total, that is, write less code, as well as reducing the number of entry points and paths through that code. In other words, my ideal design is one in which you enter the compiler in some limited, but well defined and useful set of entry points, and then proceed in a linear fashion through the code as the execution path, resulting finally in your result. This is the "ultimate" in data flow, functionally oriented programming.

The ramifications of this design choice implies a few important things. Firstly, it implies that I reduce and eliminate any code that represents boilerplate or that does not actively contribute to the "big picture" of the code. This is required in an extreme degree if I am to reduce the overall complexity of the design. This also implies that there is very little intentional redundancy in the shape and style of the source, making it very terse and compact. Since there are intentionally very few entry and exit points through the control flow of the code, this reduces the number of dependencies for me to be aware of when dealing with a single piece of code, but this also comes at the cost of not being able to see many examples of the interfaces with that code. Often, there will be one, and only one place, in which a given piece of code is used, and I do not want the code to needlessly store excess information in its source that doesn't need to be there.

This all culminates in something that can be quite shocking at first: making a change to the source is almost always a big deal. If all the source code is meaningful and carefully constructed, this also means that changing this code is almost always non-trivial, because if the code represented something trivial, I would have tried to remove it from the code so that only the "big things" were in the code itself. Thus, anyone who wishes to view and read the compiler code should take it upon themselves to appreciate the way in which the code flows together, and how the flow of the program runs, as doing so will be essential to understanding how to make changes to the source without breaking something. Fortunately, this does come with the intended benefits of a very short and simple codebase that has clear flow through the system, it just means that if you want to change something, make sure you realize that you are almost always likely to be working at the "architectural" level, rather than at the small and trivial level of details.

The compiler is designed to fit into a single Dyalog APL namespace, and importantly, we do not define additional nested namespaces or other forms of name hiding. I intentionally want to restrict the namespace to a single global one. This single global namespace should therefore contain the carefully curated names that matter, and any that do not matter should, ideally, not be defined or used. The namespace itself can be divided into three main groupings: the public facing entry-points into the system, the compiler logic itself, and the utilities or other elements that serve to support the others. This gives use the following code outline.

7 ⟨* 7⟩≡

```
:Namespace codfns

          ⟨Global Settings 10a⟩
          ⟨The Fix API 13⟩
          ⟨User-command API 15a⟩

          ⟨Parser 17⟩
          ⟨Compiler 23⟩
          ⟨Code Generator 25b⟩
          ⟨Interface to the backend C compiler 26⟩
          ⟨Linking with Dyalog 27⟩

          ⟨Must Have APL Utilities 129c⟩
          ⟨Basic tie and put utilities 132c⟩
          ⟨The opsys utility 133b⟩
          ⟨AST Record Structure 15b⟩
          ⟨Converters between parent and depth vectors 15c⟩
          ⟨XML Rendering 133a⟩
          ⟨Pretty-printing AST trees 130⟩
```

```
:EndNamespace
```

Root chunk (not used in this document).
Defines:
`codfns`, used in chunks 8, 16b, 24d, 26, 32b, 34b, 42, 48, 89, 91, 106, 112, 119, 132b, and 134–40.

This ⟨* 7⟩ chunk is meant to be stored to a file. We have a build system for doing this that depends on the contents of the ⟨*Tangle Commands* 8⟩ chunk. Thus, we follow the convention here of updating the contents of the ⟨*Tangle Commands* 8⟩ chunk each time that we initially define a new chunk that is intended to be output to a file during the tangling process. See more about the build infrastructure later in this document.

8 ⟨*Tangle Commands* 8⟩≡

```
echo "Tangling src/codfns.apln..."
notangle codfns.nw > src/codfns.apln
```

This definition is continued in chunks 16b, 32b, 34b, 42b, 48b, 91, 106b, 112b, 119b, 132b, 135, 137, and 138c.
This code is used in chunk 134.
Defines:
`codfns.apln`, never used.
Uses `codfns` 7 and `src` 139.

The primary user-facing interfaces into the compiler are ⟨*The Fix API* 13⟩ and the ⟨*User-command API* 15a⟩. These are the ways that you primarily drive the entire compiler. I intentionally expose the rest of the compiler interfaces without hiding them so that people who wish to leverage these other parts of the system without using the "entire" compiler pipeline are able to do so, but I do not consider this a public interface.

This distinction matters because of our testing philosophy and our version numbering. Generally speaking, our version numbering scheme only tracks a major or minor change in the compiler when the externally facing interfaces receive some fundamental changes. Changes to the internal changes are *not* considered for this versioning scheme. Moreover, since I intend for there to be great freedom in changing and altering the behavior of these internal pipeline interfaces, these interfaces are not directly tested, and the test suite should *not* include testing against these internal interfaces. We philosophically only test against the external interfaces, and eschew internal unit tests.[1]

The utility functions defined below the core compiler pipeline represent functionality that is tangential to the main compiler operation. However, these utilities also tend to represent some specific insight into the design of the compiler. Understanding the core AST structure and design as well as getting a grip on how to manipulate the core tree manipulation structures are vital to understanding the rest of the code. Therefore, this section spends more time on discussing these topics before the upcoming sections dealing with a more detailed exposition of the compiler itself. However, there are utilities that we consider more advanced, such as the pretty-printing functions and XML rendering that are topics of interest to advanced users of the compiler, but which are not part of the main compiler pipeline. Even though these functions have intentionally general application and are likely to be useful not only to those working on the compiler itself but also to those who are using more advanced compiler features, these utilities are not critical to a deep understanding of the compiler, so these are not discussed in this section. Instead, we discuss those topics in the section on developer tooling and infrastructure concerns.

The remaining parts of this section will describe the external facing interfaces to the compiler as well as the core underlying data structures and idioms that form the underlying skeleton and foundation for writing and working with any aspect of the compiler. These are all feature and component agnostic elements of the system that do not belong solely to only a single part, but that impact all other

[1]You can read more of my opinions on this matter in my article, "The Fallacy of Unit Testing".

elements of the compiler source code, and so it pays especially well to pay attention and understand this code to a high degree.

3.1 Global Settings

There are some global options that we assume to exist throughout the compiler. These set the standard behaviors as well as serve as knobs that can be tweaked in some cases to identify what behaviors we want from the rest of the compiler.

First, we have a set of read-only global constants that are defined to configure our APL environment. These are the typical ones, and we try to stick to the defaults, except that we are sane, and thus we use `⎕IO` set to `0`.

10a ⟨*Global Settings* 10a⟩≡

```
⎕IO ⎕ML ⎕WX←0 1 3
```

This definition is continued in chunks 10–12.
This code is used in chunk 7.
Defines:
`⎕IO`, used in chunk 131.
`⎕ML`, used in chunk 131.
`⎕WX`, never used.

Additionally, we set a `VERSION` constant to track changes to the system through the distributions. We use semantic versioning[2] as our versioning scheme. That being said, we also do not have particular qualms about changing the public API at a rapid pace, provided that we document this.

10b ⟨*Global Settings* 10a⟩+≡

```
VERSION←4 1 0
```

This code is used in chunk 7.
Defines:
`VERSION`, never used.

[2]https://semver.org/

We depend on ArrayFire[3] for much of our GPU backend functionality. This means we need to know two things, where ArrayFire is installed and which ArrayFire backend we should use when compiling. We only really need to know where ArrayFire is installed on UNIX style systems, as these systems seem to be much more variable in this regard, and there is an environment variable that we can use in Windows to find out where ArrayFire is installed more conveniently on that platform. We default to using `'cuda'` as our main option, but we also support the following options for `AF∆LIB`:

```
cuda opencl cpu
```

Using `''` for `AF∆LIB` will use ArrayFire's unified backend, but we don't default to this because we have seen some issues on some platforms with reliability problems. To avoid this, we choose to use `cuda` as the default, which tends to either work or fail explicitly, which allows the user to respond rather than crashing ungracefully in the case of the unified backend.

The least reliable backend we have seen is the `opencl` one, which seems to be more hit or miss depending on the underlying stability of the OpenCL drivers that are installed on the user's system. In particular, some Linux OpenCL installations seem to be particularly fragile. In such cases, always make sure that a good, solid OpenCL library is being used.

11 ⟨*Global Settings* 10a⟩+≡

```
AF∆PREFIX←'/opt/arrayfire'
AF∆LIB←'cuda'
```

This code is used in chunk 7.
Defines:
`AF∆LIB`, used in chunks 15a, 26, and 140a.
`AF∆PREFIX`, used in chunk 26.

[3] https://arrayfire.com/

On Windows, we rely on the Visual Studio C/C++ compiler to build our runtime and user code. We have settled on trying to stay as up to date with this as possible. However, there are many different installation paths used by Visual Studio, which can make it difficult to know where to look unless we hardcode each location. Instead, we assume that Visual Studio will not be a primary interest to our users, making it likely that they will be installing Visual Studio only as a dependency for using Co-dfns. In this case, it is likely that they will be using the Community version. Thus, we default to using the latest version of Visual Studio of which we are aware and using the Community version of this, which Microsoft does not charge for.

If a different version of Visual Studio is installed, then it is important to figure out what the right path should be to locate the Visual Studio installation. The main thing we need to get from this path is access to the `vcvarsall.bat` batch file. This file configures the `cmd.exe` environment to be able to find the Visual Studio compiler and work in the right way. In the 2002 Community addition, and apparently most new versions of Visual Studio, this is located in the `VC\Auxiliary\Build\` subdirectory of the main installation folder. When changing this path, we want to make sure that the following path points to the correct `vcvarsall.bat` file:

```
VS∆PATH,'\VC\Auxiliary\Build\vcvarsall.bat'
```

Most users will simply need to alter `Community` to match the edition of Visual Studio 2022 that they have installed on their system.

12 ⟨*Global Settings* 10a⟩+≡

```
VS∆PATH←'\Program Files\Microsoft Visual Studio'
VS∆PATH,←'\2022\Community'
```

This code is used in chunk 7.
Defines:
`VS∆PATH`, used in chunks 26 and 140a.

3.2 The Fix API

One of the core entry points into the compiler is through the `Fix` function. This function is designed to mimic and more or less replace the use of the `⎕FIX` function found in Dyalog APL. Its design models that behavior, and it is important as an entry-point because it exercises most of the core elements of the compiler. In particular, the design of the compiler's pipeline is demonstrated most fully in this function.

Parse → *Compile* → *Generate* → *Backend* → *Link*

The interfaces to the `⎕FIX` function and the Co-dfns `Fix` function differ in a few key ways. The left argument to `Fix` is a character vector giving the name to use when generating files and other artifacts. This does *not* affect the name of the resulting namespace, since that is defined, if at all, in the file source itself. The `⍺` argument only affects the name of the files and other outputs that `Fix` generates.

We also print out which part of the compiler we are in when we enter that "phase". Doing this helps to give us an intuitive sense of how fast each phase is and whether one phase is taking an abnormally long time or not. It also helps in debugging.

13 ⟨*The Fix API* 13⟩≡

```
Fix←{
          _←a n s src←PS ⍵⊣⍞←'P'
          _←            TT _⊣⍞←'C'
          _←            GC _⊣⍞←'G'
          _←          ⍺ CC _⊣⍞←'B'
                      n NS _⊣⍞←'L'
}
```

This code is used in chunk 7.
Defines:
`Fix`, used in chunk 15a.
Uses `PS` 17 and `src` 139.

The input requirements for `Fix` are not listed in the definition itself, because both the parser `PS` and the `Fix` function need to use the same basic checks, and since the `Fix` function calls the parser as its first entry point, it doesn't make much sense to duplicate that work in both places. The requirements are as follows:

- Scalar/Vector
- Character type
- Simple or Vector of Vectors

We generate a `DOMAIN ERROR` if the inputs are not well-formed.

14a ⟨*Verify source input* ⍵*, set* `IN` 14a⟩≡
```
IN←⍵

err←'PARSER EXPECTS SCALAR OR VECTOR INPUT'
1<≢⍴IN:err ⎕SIGNAL 11

err←'PARSER EXPECTS SIMPLE OR VECTOR OF VECTOR INPUT'
2<|≡IN:err ⎕SIGNAL 11

⟨Normalize the input formatting 14b⟩

err←'PARSER EXPECTS CHARACTER ARRAY'
0≠10|⎕DR IN:err ⎕SIGNAL 11
```
This code is used in chunk 17.
Uses `SIGNAL` 20b.

The input formatting that is accepted means that newlines could be denoted either with `LF`, `CR`, or `CRLF` sequences inside of the vectors themselves or they could be denoted by having separate vectors for the various lines, or even a mixture of both. To simplify this situation we want to normalize them so that we are always dealing with some combination of `LF`, `CR`, and `CRLF` sequences within the file itself, rather than dealing with the nested situation. This ensures that after verification of the input, everything will work off of the same format. We intentionally put a newline at the end of the file even if we may not require one because it is possible that we are dealing with a file that is missing its final newline. By always adding one, we ensure that every line in the input is always terminated by a line ending. Life is also simpler if we just use LF as our line ending instead of something else, this means that future code must be aware that there could be mixed line endings in the file.

14b ⟨*Normalize the input formatting* 14b⟩≡
```
IN←∊(⊆IN),¨⎕UCS 10
```
This code is used in chunk 14a.

3.3 The User Command API

15a ⟨*User-command API* 15a⟩≡
```
∇Z←Help _
 Z←'Usage: <object> <target> [-af={cpu,opencl,cuda}]'
∇

∇r←List
 r←⎕NS¨1⍴⊂⍬ ⋄ r.Name←,¨⊂'Compile' ⋄ r.Group←⊂'CODFNS'
 r[0].Desc←'Compile an object using Co-dfns'
 r.Parse←⊂'2S -af=cpu opencl cuda '
∇

∇ Run(C I);Convert;in;out
⍝ Parameters
⍝      AF∆LIB        ArrayFire backend to use
 Convert←{⍺(⎕SE.SALT.Load'[SALT]/lib/NStoScript -noname').ntgennscode ⍵}
 in out←I.Arguments ⋄ AF∆LIB←I.af''⊃⍨I.af≡0
 S←(⊂':Namespace ',out),2↓0 0 0 out Convert ##.THIS.⍎in
 →0⌿⍨'Compile'≢C
 {##.THIS.⍎out,'←⍵'}out Fix S⊣⎕EX'##.THIS.',out
∇
```
This code is used in chunk 7.
Uses `AF∆LIB` 11 and `Fix` 13.

3.4 AST Record Structure

15b ⟨*AST Record Structure* 15b⟩≡
```
f∆←'ptknfsrdx'
N∆←'ABCEFGKLMNOPSVZ'
A B C E F G K L M N O P S V Z←1+⍳15
```
This code is used in chunk 7.

3.5 Converters between parent and depth vectors

15c ⟨*Converters between parent and depth vectors* 15c⟩≡
```
P2D←{z←⍪⍳≢⍵ ⋄ d←⍵≠,z ⋄ _←{p⊣d+←⍵≠p←⍺[z,←⍵]}⍣≡⍨⍵ ⋄ d(⍋(-1+d)↑⍤0 1⊢⌽z)}
D2P←{0=≢⍵:⍬ ⋄ p⊣2{p[⍵]←⍺[⍺⍸⍵]}⌿⊢∘⊂⌸⍵⊣p←⍳≢⍵}
```
This code is used in chunk 7.

4 Testing

We use the APLUnit testing framework to facilitate our testing of the Co-dfns compiler. The test harness is designed around a testing philosophy in which we ever only write black-box tests that work on the whole compiler using inputs that could be created or are expected to be creatable by end-users. That is, we do no "unit testing" of our source code, but only whole program testing.

The testing framework is provided by the `ut.apln` file, which is not part of this literate program and so is not included in this document. In order to make some of the testing more convenient, we define the function `TEST` to run the tests that exist in the `tests\` subdirectory. Each of these tests has a specific number which defines the test, and we refer to the tests by number when running them. Both of these testing functions assume that we are running inside of the `tests\` directory or one configured identically to it.

The `TEST` function takes either `'ALL'` as its input or a test number in the form of an integer. Given an integer, we call the test matching that number in the current working directory.

The `'ALL'` option causes `TEST` to run all of the tests that are defined in the current working directory. This command is a nicety, since we can technically do all of this by iterating the `TEST` function over the range of test numbers, but this would not create the aggregate statistics that we would like to see at the end of the testing report. By using `'ALL'` we get to see a complete summary of the results of testing all the code, rather than just the individual testing results on a per testing group/number basis.

16a ⟨`TEST` 16a⟩≡
```
TEST←{
        #.UT.(print_passed print_summary)←1
        'ALL'≡⍵:#.UT.run './'
        path←'./t',(1 0⍕(4⍴10)⊤⍵),'_*_tests.dyalog'
        #.UT.run ⊃⊃0⎕NINFO⍠1⊢path
}
```
Root chunk (not used in this document).
Defines:
`TEST`, used in chunks 16b and 115a.

The `TEST` function is part of the utilities that exist outside of the `codfns` namespace, so we define a file for it.

16b ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling src/TEST.aplf..."
notangle -R'[[TEST]]' codfns.nw > src/TEST.aplf
```
This code is used in chunk 134.
Defines:
`TEST.aplf`, never used.
Uses `codfns` 7, `src` 139, and `TEST` 16a.

5 Co-dfns Compiler

5.1 Parser

The first, and in many ways, the most complex element of the compiler is the parser. APL has a number of unique issues when it comes to adequately parsing the language, but the most important is handling the context-sensitive nature of parsing variables: depending on the type of a variable, the parse tree can look very different. To manage this, we make use of a linear, multi-pass style of parser in which the parsing process consists of numerous small passes over the input, each time refining the input into something more like the final result. The parser should take some input that matches the input requirements of the `Fix` function and produce a suitable output AST.

$$PS :: Source \rightarrow AST \times ExportTypes \times SymbolTable \times Source$$

We can think of the parser as starting with a forest of trees, each of which contains a single root node that represents a single character in from the input source, with all trees arranged in the source order. During each pass of the parser, we progressively combine these trees into more complex trees until we end up at the end with a single tree per parsed module. In other words, we take a fully flat forest of single-node trees and progressively increase the depth while reducing the number of root-nodes until we have our desired AST structure.

We divide the parsing roughly into two main phases, the tokenization phase and the parsing phase. Unlike most compilers, we don't have a strict division in these two phases, so, as they say, think of them more like guidelines than actual rules[4].

17 ⟨*Parser* 17⟩≡

```
PS←{
        ⟨Verify source input ⍵, set IN 14a⟩

        ⟨Parsing Constants 18a⟩
        ⟨Line and error reporting utilities 20b⟩

        ⟨Tokenize input 21⟩
        ⟨Parse token stream 22⟩

        ⟨Compute parser exports 117b⟩
        ⟨Adjust AST for output 18b⟩
}
```

[4]https://www.youtube.com/watch?v=WJVBvvS57j0

This code is used in chunk 7.
Defines:
PS, used in chunks 13 and 139.

When parsing, it's very helpful to have names for line endings.

18a ⟨*Parsing Constants* 18a⟩≡
```
CR LF←⎕UCS 13 10
```
This code is used in chunk 17.

5.1.1 Output of the Parser

After we finish all of our parsing, we need to take the resulting AST and convert that into something that is suitable for output to the rest of the system. We do this in a few ways.

When we finish parsing, we expect the following fields:

Field	Description
d	Depth vector
t	Node type
k	Node sub-class or "kind"
n	Name/value field
pos	Starting index for source position
end	Exclussive index for source end position
xn	Names of top-level exported bindings
xt	Types of top-level exported bindings
sym	Symbol Table
IN	Canonical source code

On parser output, we want to convert the AST to an order that follows a depth-first, preorder traversal order, so that we can switch from using the parent vector to the depth vector. We use this output as our main output because it is space efficient for storage, and it works well as a canonical form to use. Because applications may want to only use the parser and not the rest of the compiler, we want to choose an output format that is suitable for external as well as internal use. This has some performance overheads, but it is probably worth it regardless, as reordering at this point to allow a depth vector enables some nice assumptions in the rest of the compiler. We use the P2D utility to reorder all of our AST columns. Note that things like the exported bindings and the symbol table are not strictly part of the AST structure, because they are of a different length and type than the other columns.

18b ⟨*Adjust AST for output* 18b⟩≡
```
d i←P2D p ⋄ d n t k pos end I∘⊢←⊂i
```
This definition is continued in chunks 19 and 20a.
This code is used in chunk 17.

There is an inefficiency in the AST representation at this point, where the `n` field contains character vectors. This inefficiency was necessary while building up the AST because we were not sure what symbols would be created before we parsed them, but at this point, we know the full set of symbols that we have in the AST. This means that we can convert the `n` field to a symbol table representation. In this case, we want the `n` field to pair with a `sym` list that contains all the unique symbols in the source. We want `old_n≡sym[|new_n]` to hold for this new `n` field. In other words, we want the new `n` field to contain negative integers whose magnitudes are valid indices into the `sym` symbol table. This means that there is only one character vector per unique symbol or numeric literal in the source code, which can greatly reduce memory usage. Moreover, it is much faster to compare symbols that are represented by numeric index rather than character vector. Most of the work we expect to be done on the `n` field, so that we never have to pull in `sym` unless we want to know the actual value of the symbol. This actually mimics the feature of symbols in other languages like Scheme, but it comes with an additional efficiency benefit in that we do not require the use of a full generalized pointer to represent a symbol if we have fewer symbols. This means that we are very likely only going to need a single byte or a couple of bytes per symbol to represent it in the `n` field.

The choice to make all of our symbols negative in value is somewhat strange, but we have a good reason for doing so. The `n` field is a single field that we use to contain general data for every node, and as such, it represents a sort of union type of all sorts of different data. In particular, we also want to be able to support using the `n` field to point to other nodes in the AST, which is a feature we rely heavily on in the compiler transformations. However, this feature would conflict with using the `n` field as an index into the `sym` table, rather than as an index into the AST. By making symbol pointers negative, we put them into a separate space than the positive AST node pointers, allowing us to store both pointers in the same field. This may seem like a little bit of a strange hack, but it actually makes reasoning about things a little easier, because we can tend to think of `n` as a name, even if that name is pointing to an AST or a symbol, and avoids needless space duplication or the need to remember to update multiple fields that are only relevant for some nodes.

We map the 0th index to be a null or empty symbol. We also want to reserve the first four symbol slots $[1, 4]$ so that they will *always* refer to the same symbols, namely, `⍵`, `⍺`, `⍺⍺`, and `⍵⍵`.

This gives us the following definitions for `sym` and `n`.

19 ⟨*Adjust AST for output* 18b⟩+≡

```
sym←∪('')(,'⍵')(,'⍺')'⍺⍺' '⍵⍵',n
n←-sym⍳n
```

This code is used in chunk 17.

Finally, we want to return our AST structure in a meaningful way. Logically, we have the AST proper, which consists of these fields:

```
d t k n pos end
```

The above fields are returned as an inverted table, where each column is a vector of the same length. We also want to return the variable environment, which gives the names of our top-level bindings and their types, also as an inverted table. Finally, we must return a canonical representation of the source code that is suitable as an indexing target for the `pos` and `end` fields, as well as the symbol table. Thus, we have a four element vector as the return value:

```
AST TopBindingTypes SymbolTable InputSource
```

Which gives us the following return value.

20a ⟨*Adjust AST for output* 18b⟩+≡

```
  (d t k n pos end)(xn xt)sym IN
```

This code is used in chunk 17.
Uses `xn` 117b and `xt` 117b.

5.1.2 Handling Parsing Errors

20b ⟨*Line and error reporting utilities* 20b⟩≡

```
  linestarts←(⍸1⍪2>⌿IN∊CR LF)⍪≢IN
  mkdm←{⍺←2 ⋄ line←linestarts⍸⍵ ⋄ no←'[',(⍕1+line),'] '
          i←(~IN[i]∊CR LF)⌿i←beg+⍳linestarts[line+1]-beg←linestarts[line]
          (⎕EM ⍺)(no,IN[i])(' ^'[i∊⍵],⍨' '⍴⍨≢no)}
  quotelines←{
          lines←∪linestarts⍸⍵
          nos←(1 0⍴⍨2×≢lines)⍀'[',(⍕⍪1+lines),⍤1⊢'] '
          beg←linestarts[lines] ⋄ end←linestarts[lines+1]
          m←∊∘⍵¨i←beg+⍳¨end-beg
          ¯1↓∊nos,(~∘CR LF¨⍪,(IN∘I¨i),⍪' ▔'∘I¨m),CR}
  SIGNAL←{⍺←2 '' ⋄ en msg←⍺ ⋄ EN∘←en ⋄ DM∘←en mkdm ⊃⍵
          dmx←('EN' en)('Category' 'Compiler')('Vendor' 'Co-dfns')
          dmx,←⊂'Message'(msg,CR,quotelines ⍵)
          ⎕SIGNAL⊂dmx}
```

This code is used in chunk 17.
Defines:
  `linestarts`, never used.
  `mkdm`, never used.
  `quotelines`, used in chunk *55*.
  `SIGNAL`, used in chunks 14a, 24–27, *55*, *57*a, 67, 72–74, 83b, 99–101, 106c, 108a, 112d, 114–17, 119d, 129c, and 132c.
Uses `dmx` 44a.

5.1.3 Tokenizing the Input

21 ⟨*Tokenize input* 21⟩≡

```
⍝ Group input into lines as a nested vector
pos←(⍳≢IN)⊆⍨~IN∊CR LF

⟨Mask potential strings 56⟩
⟨Remove comments 49⟩
⟨Check for unbalanced strings 57a⟩
⟨Flatten parser representation 50⟩
⟨Tokenize strings 57b⟩
⟨Convert ⋄ to Z nodes 51a⟩
⟨Define character classes 51b⟩
⟨Remove insignificant whitespace 51c⟩
⟨Verify that all open characters are valid 55⟩

x←IN[pos]
⟨Tokenize numbers 64⟩
⟨Tokenize variables 83a⟩
⟨Tokenize primitives and atoms 98d⟩
⟨Compute dfns regions and type, with } as a child 112d⟩
⟨Check for out of context dfns formals 83b⟩
⟨Compute trad-fns regions 114c⟩
⟨Identify label colons vs. others 115d⟩
⟨Tokenize keywords 116a⟩
⟨Tokenize system variables 99b⟩

⍝ Delete all characters we no longer need from the tree
d tm t pos end(⌿⍨)←⊂(t≠0)∨x∊'()[]{}:;'

⟨Tokenize labels 115e⟩
```

This code is used in chunk 17.

5.1.4 Parsing Token Stream

22 ⟨*Parse token stream* 22⟩≡

```
⍝ Now that all compound data is tokenized, reify n field before tree-building
n←{1↓⍎¨'0',ω}@{t=N}(⊂'')@{t∊Z F}1 ⎕C@{t∊K S}IN∘I¨pos+⍳¨end-pos
⟨Type-specific processing of the n field 59⟩

⟨Check that all keywords are valid 116b⟩
⟨Check that namespaces are at the top level 116c⟩
⟨Verify that all structured statements appear within trad-fns 119d⟩
⟨Verify that system variables are defined 99c⟩

⍝ Compute parent vector from d
p←D2P d

⟨Compute the nameclass of dfns 112e⟩

⍝ We will often wrap a set of nodes as children under a Z node
gz←{
        z←ω↑⍨-0≠≢ω ⋄ ks←¯1↓ω
        t[z]←Z ⋄ p[ks]←⊃z ⋄ pos[z]←pos[⊃ω] ⋄ end[z]←end[⊃⌽z,ks]
        z
}

⟨Nest top-level root lines as Z nodes 116d⟩
⟨Wrap all dfns expression bodies as Z nodes 112f⟩

⍝ Drop/eliminate any Z nodes that are empty or blank
_←p[i]{msk[α,ω]←~∧⌿IN[pos[ω]]∊WS}⌸i←⍸(t[p]=Z)∧p≠⍳≢p⊣msk←t≠Z
tm n t k pos end(⌿⍨)←⊂msk ⋄ p←(⍸~msk)(⊢-1+⍸)msk⌿p

⟨Parse :Namespace syntax 117a⟩
⟨Parse guards to (G (Z ...) (Z ...)) 115a⟩
⟨Parse brackets and parentheses into ¯1 and Z nodes 106c⟩
⟨Convert ; groups within brackets into Z nodes 100a⟩
⟨Parse Binding nodes 101b⟩
⟨Mark system variables as P nodes with appropriate kinds 99d⟩
⟨Mark atoms, characters, and numbers as kind 1 84a⟩
⟨Mark APL primitives with appropriate kinds 99a⟩
⟨Anchor variables to earliest binding in the matching frame 112g⟩
⟨Convert M nodes to FO nodes 119e⟩
⟨Convert α and ω to V nodes 83c⟩
⟨Convert αα and ωω to P2 nodes 83d⟩
⟨Infer the type of bindings, groups, and variables 102a⟩
⟨Strand arrays into atoms 84b⟩
⟨Parse dyadic operator bindings 102b⟩
```

```
⟨Rationalize F[X] syntax 101a⟩
⟨Group function and value expressions 106d⟩
⟨Parse function expressions 108a⟩
⟨Parse assignments 103a⟩
⟨Enclose V[X;...] for expression parsing 100c⟩
⟨Parse trains 108c⟩
⟨Parse value expressions 107b⟩
⟨Rationalize V[X;...] 100d⟩

⍝ Sanity check
ERR←'INVARIANT ERROR: Z node with multiple children'
ERR assert(+⌿(t[p]=Z)∧p≠⍳≢p)=+⌿t=Z:

⍝ Count parentheses in source information
ip←p[i←⍸(t[p]=Z)∧n[p]∊⊂,'('] ⋄ pos[i]←pos[ip] ⋄ end[i]←end[ip]

⍝ VERIFY Z/B NODE TYPES MATCH ACTUAL TYPE

⍝ Eliminate Z nodes from the tree
zi←p I@{t[p[⍵]]=Z}⍣≡ki←⍸msk←(t[p]=Z)∧t≠Z
p←(zi@ki⍳≢p)[p] ⋄ t k n pos end(⊣@zi⍨)←t k n pos end I¨⊂ki
t k n pos end⌿⍨←⊂msk←~msk∨t=Z ⋄ p←(⍸~msk)(⊢-1+⍸)msk⌿p
```
This code is used in chunk 17.
Uses `assert` 129c and `WS` 51b.

5.2 Compiler Transformations

23 ⟨*Compiler* 23⟩≡

```
TT←{
        ((d t k n ss se)exp sym src)←⍵

        ⍝ Compute parent vector and reference scope
        r←I@{t[⍵]≠F}⍣≡⍨p⊣2{p[⍵]←⍺[⍺⍸⍵]}⌿⊢∘⊂⌸d⊣p←⍳≢d

        ⟨Lift dfns to the top-level 113a⟩
        ⟨Wrap expressions as binding or return statements 113b⟩
        ⟨Lift guard tests 115b⟩
        ⟨Count strand and indexing children 84c⟩
        ⟨Lift and flatten expressions 107a⟩
        ⟨Compute slots and frames 113d⟩
        ⟨Record exported top-level bindings 117c⟩

        p t k n f s r d xi sym
}
```
This code is used in chunk 7.
Uses `src` 139 and `xi` 117c.

5.3 Code Generator

24a ⟨*Map generators over the linearized AST; return* 24a⟩≡
```
d i←P2D p ⋄ ast←(⍉↑d p t k n(⍳≢p)fr sl fd)[i;] ⋄ ks←{⍵⊂[0]⍨(⊃⍵)=⍵[;0]}
NOTFOUND←{('[GC] UNSUPPORTED NODE TYPE ',N∆[⊃⍵],⍕⊃⌽⍵)⎕SIGNAL 16}
dis←{0=2⊃h←,1↑⍵:'' ⋄ (≢gck)=i←gck⍳⊂h[2 3]:NOTFOUND h[2 3] ⋄ h(⍎i⊃gcv)ks 1↓⍵}
∊,∘(⎕UCS 13 10)¨pref,⊃,⌿(,⌿Zp¨⍸t=F),(,⌿Zx¨xi),(⊂⊂''),dis¨ks ast
```
This code is used in chunk 25b.
Uses `SIGNAL` 20b and `xi` 117c.

24b ⟨*Symbol ⟷ Name mapping* 24b⟩≡
```
syms←0⍴⊂'' ⋄ nams←0⍴⊂''
```
This definition is continued in chunks 100e, 103b, 113e, and 120–29.
This code is used in chunk 25b.

24c ⟨*Node ⟷ Generator mapping* 24c⟩≡
```
gck←0⍴⊂0 0 ⋄ gcv←0⍴⊂''
```
This definition is continued in chunks 83e, 84d, 98b, 100f, 102c, 103c, 107c, 108b, 113, 115c, and 117d.
This code is used in chunk 25b.

24d ⟨*Prefix code for all generated files* 24d⟩≡
```
pref ←⊂'#include "codfns.h"'
pref,←⊂''
pref,←⊂'EXPORT int'
pref,←⊂'DyalogGetInterpreterFunctions(void *p)'
pref,←⊂'{'
pref,←⊂'    return set_dwafns(p);'
pref,←⊂'}'
pref,←⊂''
```
This code is used in chunk 25b.
Uses `codfns` 7, `codfns.h` 34a, and `set_dwafns` 47a.

24e ⟨*Node-specific code generators* 24e⟩≡
```
Zp←{
          n←'fn',⍕⍵
          ⟨Declare top-level function bindings 109a⟩
          'UNKNOWN FUNCTION TYPE'⎕SIGNAL 16
}
```
This definition is continued in chunks 25a, 83f, 98c, 102d, 107d, 113, 114, and 118.
This code is used in chunk 25b.
Uses `SIGNAL` 20b.

25a ⟨*Node-specific code generators* 24e⟩+≡
```
Zx←{
        n←sym⊃⍨|n[⍵] ⋄ rid←⍕rf[⍵]
        k[⍵]=0:⊂''
```
⟨*Declare top-level array structures* 84e⟩
⟨*Declare top-level closures* 109b⟩
```
        ⍎'''UNKNOWN EXPORT TYPE''⎕SIGNAL 16'
}
```
This code is used in chunk 25b.
Uses `EXPORT` 35 and `SIGNAL` 20b.

25b ⟨*Code Generator* 25b⟩≡
```
GC←{
        p t k n fr sl rf fd xi sym←⍵
```
⟨*Symbol* ⟷ *Name mapping* 24b⟩
⟨*Node* ⟷ *Generator mapping* 24c⟩

⟨*Prefix code for all generated files* 24d⟩
⟨*Node-specific code generators* 24e⟩

⟨*Map generators over the linearized AST; return* 24a⟩
```
}
```
This code is used in chunk 7.
Uses `xi` 117c.

5.4 Backend C Compiler Interface

26 ⟨*Interface to the backend C compiler* 26⟩≡

```
CC←{
         vsbat←VS∆PATH,'\VC\Auxiliary\Build\vcvarsall.bat'
         soext←{opsys'.dll' '.so' '.dylib'}
         libdir←opsys ''' ' '/lib64'' ' '/lib'' '
         ccf←{' -o ''',ω,'.',α,''' ''',ω,'.c'' -laf',AF∆LIB,' > ',ω,'.log 2>&1'}
         cci←{'-I''',AF∆PREFIX,'/include'' -L''',AF∆PREFIX,libdir}
         cco←'-std=c99 -Ofast -g -Wall -fPIC -shared '
         cco,←'-Wno-parentheses -Wno-misleading-indentation '
         ucc←{ωω(⎕SH αα,' ',cco,cci,ccf)ω}
         gcc←'gcc'ucc'so'
         clang←'clang'ucc'dylib'
         vsco←{z←'/W3 /wd4102 /wd4275 /O2 /Zc:inline /Zi /FS /Fd"',ω,'.pdb" '
                  z,←'/WX /MD /EHsc /nologo '
                  z,'/I"%AF_PATH%\include" /D "NOMINMAX" /D "AF_DEBUG" '}
         vslo←{z←'/link /DLL /OPT:REF /INCREMENTAL:NO /SUBSYSTEM:WINDOWS '
              z,←'/LIBPATH:"%AF_PATH%\lib" /OPT:ICF /ERRORREPORT:PROMPT /TLBID:1
                  z,'/DYNAMICBASE "af', AF∆LIB, '.lib" "codfns.lib" '}
         vsc0←{~⎕NEXISTS vsbat:'VISUAL C?'⎕SIGNAL 99 ⋄ '""',vsbat,'" amd64'}
         vsc1←{' && cd "',(⊃⎕CMD'echo %CD%'),'" && cl ',(vsco ω),' "',ω,'.c" '}
         vsc2←{(vslo ω),'/OUT:"',ω,'.dll" > "',ω,'.log"'}
         vsc←{⎕CMD ('%comspec% /C ',vsc0,vsc1,vsc2)ω}
         _←(⍎opsys'vsc' 'gcc' 'clang')α⊣ω put α,'.c'⊣1 ⎕NDELETE f←α,soext⍬
         ⎕←⍪⊃⎕NGET(α,'.log')1
         ⎕NEXISTS f:f ⋄ 'COMPILE ERROR' ⎕SIGNAL 22}
```

This code is used in chunk 7.
Uses AF∆LIB 11, AF∆PREFIX 11, codfns 7, opsys 133b, put 132c, SIGNAL 20b, vsbat 140a, vsc 140a, and VS∆PATH 12.

5.5 Linking with Dyalog

27 ⟨*Linking with Dyalog* 27⟩≡

```
NS←{
        MKA←{mka⊂ω} ⋄ EXA←{exa ⍬ ω}
        Display←{α←'Co-dfns' ⋄ W←w_new⊂α ⋄ 777::w_del W
         w_del W⊣W αα{w_close α:⍎'⎕SIGNAL 777' ⋄ α αα ω}⍣ωω⊢ω}
        LoadImage←{α←1 ⋄ ~⎕NEXISTS ω:⎕SIGNAL 22 ⋄ loadimg ⍬ ω α}
        SaveImage←{α←'image.png' ⋄ saveimg ω α}
        Image←{~2 3∨.=≢⍴ω:⎕SIGNAL 4 ⋄ (3≠⊃⍴ω)∧3=≢⍴ω:⎕SIGNAL 5 ⋄ ω⊣w_img ω α}
        Plot←{2≠≢⍴ω:⎕SIGNAL 4 ⋄ ~2 3∨.=1⊃⍴ω:⎕SIGNAL 5 ⋄ ω⊣w_plot (⍉ω) α}
        Histogram←{ω⊣w_hist ω,α}
        Rtm∆Init←{
                _←'w_new'⎕NA'P ',ω,'|w_new <C[]'
                _←'w_close'⎕NA'I ',ω,'|w_close P'
                _←'w_del'⎕NA ω,'|w_del P'
                _←'w_img'⎕NA ω,'|w_img <PP P'
                _←'w_plot'⎕NA ω,'|w_plot <PP P'
                _←'w_hist'⎕NA ω,'|w_hist <PP F8 F8 P'
                _←'loadimg'⎕NA ω,'|loadimg >PP <C[] I'
                _←'saveimg'⎕NA ω,'|saveimg <PP <C[]'
                _←'exa'⎕NA ω,'|exarray >PP P'
                _←'mka'⎕NA'P ',ω,'|mkarray <PP'
                _←'FREA'⎕NA ω,'|frea P'
                _←'Sync'⎕NA ω,'|cd_sync'
                0 0 ⍴ ⍬}
        mkna←{α,'|',('∆'⎕R'__'⊢ω),'_cdf P P P'}
        mkf←{
                fn←α,'|',('∆'⎕R'__'⊢ω),'_dwa '
                z←⊂'Z←{A}',ω,' W'
                z,←⊂':If 0=⎕NC''⍙.',ω,'_mon'''
                z,←⊂'        ''',ω,'_mon''⍙.⎕NA''',fn,'>PP P <PP'''
                z,←⊂'        ''',ω,'_dya''⍙.⎕NA''',fn,'>PP <PP <PP'''
                z,←⊂':EndIf'
                z,←⊂':If 0=⎕NC''A'''
                z,←⊂'       Z←⍙.',ω,'_mon 0 0 W'
                z,←⊂':Else'
                z,←⊂'       Z←⍙.',ω,'_dya 0 A W'
                z,←⊂':EndIf'
                z
        }
        ns←#.⎕NS⍬ ⋄ _←'∆⍙'ns.⎕NS¨⊂⍬ ⋄ ∆ ⍙←ns.(∆ ⍙)
        ∆.names←(0⍴⊂''),(2=1⊃α)⌿0⊃α
        fns←'Rtm∆Init' 'MKA' 'EXA' 'Display'
        fns,←'LoadImage' 'SaveImage' 'Image' 'Plot' 'Histogram'
        fns,←'soext' 'opsys' 'mkna'
```

```
          _←∆.⎕FX∘⎕CR¨fns
          ∆.(decls←ω∘mkna¨names)
          _←ns.⎕FX¨(⊂''),ω∘mkf¨∆.names
          _←'Z←Init'
          _,←⊂'Z←Rtm∆Init ''',ω,''''
          _,←⊂'→0⌿⍨0=≢names'
          _,←⊂'names ##.⍙.⎕NA¨decls'
          _←∆.⎕FX _
          ns
  }
```

This code is used in chunk 7.
Uses PP 132a and SIGNAL 20b.

29 ⟨*DWA Function Export* 29⟩≡
```
z,←c'EXPORT int'
z,←cn,'_dwa(struct localp *zp, struct localp *lp, struct localp *rp)'
z,←c'{'
z,←c'	struct array *z, *l, *r;'
z,←c'	int err;'
z,←c''
z,←c'	l = NULL;'
z,←c'	r = NULL;'
z,←c''
z,←c'	fn',rid,'(NULL, NULL, NULL, NULL);'
z,←c''
z,←c'	err = 0;'
z,←c''
z,←c'	if (lp)'
z,←c'		err = dwa2array(&l, lp->pocket);'
z,←c''
z,←c'	if (err)'
z,←c'		dwa_error(err);;'
z,←c''
z,←c'	if (rp)'
z,←c'		dwa2array(&r, rp->pocket);'
z,←c''
z,←c'	if (err) {'
z,←c'		release_array(l);'
z,←c'		dwa_error(err);'
z,←c'	}'
z,←c''
z,←c'	err = (',n,'->fn)(&z, l, r, ',n,'->fv);'
z,←c''
z,←c'	release_array(l);'
z,←c'	release_array(r);'
z,←c''
z,←c'	if (err)'
z,←c'		dwa_error(err);'
z,←c''
z,←c'	err = array2dwa(NULL, z, zp);'
z,←c'	release_array(z);'
z,←c''
z,←c'	if (err)'
z,←c'		dwa_error(err);'
z,←c''
z,←c'	return 0;'
z,←c'}'
z,←c''
```
This code is used in chunk 109b.

Uses `array2dwa` 92, `dwa2array` 92, `dwa_error` 45a, and `release_array` 86.

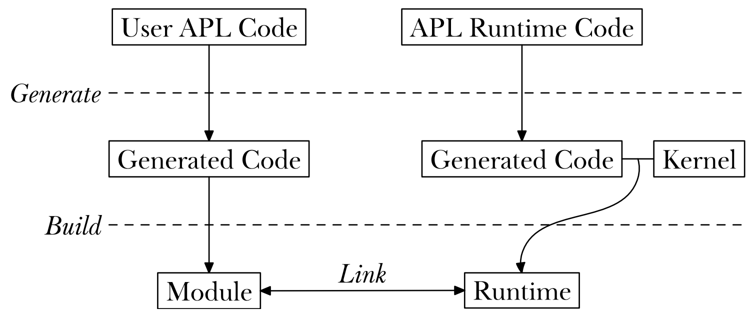

Figure 1: Process of Building and Linking the Runtime

5.6 APL Runtime Architecture

The runtime component of Co-dfns handles the code necessary for the output of the Code Generator to run. This includes support for all the supported language features as well as the runtime code for the built-in APL primitives and system functionality. The design of the runtime is meant to allow for as much of the runtime as possible to be implemented in APL. We also want to make it as easy as possible to target new languages for output from the compiler.

Conceptually, the code generator produces a code module that links against an already built runtime module that provides all the language support. Each module has some "backend target" language. In order to make retargeting the compiler as simple as possible and to implement most of the runtime as APL, we split the runtime code into an APL namespace, containg all the APL code that is applicable to all backends and that can be implemented in APL, and a backend kernel that contains all the backend language-specific code that we must use. We can split the compiler into a frontend *generate* and a backend *build* step. The generate phase takes the input APL source and generates code in the backend target language that depends on a runtime implementation. The build phase takes that code and uses the backend toolchain to link, compile, and otherwise assemble the code into an appropriate redistributable "binary". The C backend, for instance, takes APL and turns it into C code where a C compiler then builds and links it against a runtime, finally producing a DLL.

To build the runtime, the same basic approach is used. We use the compiler to generate a backend file from the APL runtime code. However, since no runtime exists for the runtime itself, we do *not* continue in the typical manner and build with the standard backend pipeline, which assumes the existence of a runtime. Instead, we merge the generated code with the kernel for that specific backend and build as its own standalone object.

This workflow is illustrated by Figure 1 showing how all of the pieces of the runtime interact with user code.

This architecture has some interesting advantages. First, most of the process for building the runtime is just like building any other piece of APL code. Second, only a small kernel and code generator need to be implemented for a new backend, with most of the work remaining in the APL runtime code. Third, the runtime may be implemented using a different backend language than that used for compiling the user code. All that is required is that the backend for the user code knows how to link to and access the code in the runtime object. This permits, for instance, a Scheme or Javascript backend to depend on a runtime implemented in C, thus enabling greater performance while hiding any integration hassles from the interface exposed by the user module. In theory, any combination of suitable backend languages may be used.

We put all the runtime primitives into a single Co-dfns namespace called `prim.apln`.

32a ⟨*prim.apln* 32a⟩≡

```
:Namespace prim

        ⟨APL Primitives 128a⟩
        ⟨System Primitives (never defined)⟩

:EndNamespace
```

Root chunk (not used in this document).
Defines:
`prim`, used in chunks 32b and 139.
`prim.apln`, used in chunk 32b.

32b ⟨*Tangle Commands* 8⟩+≡

```
echo "Tangling rtm/prim.apln..."
notangle -R'prim.apln' codfns.nw > rtm/prim.apln
```

This code is used in chunk 134.
Uses `codfns` 7, `prim` 32a, and `prim.apln` 32a.

Each primitive has its own unique considerations, so we leave the definition of these primitives to section 7.

For each backend we must have a unique kernel and code generator. Most of that content will be defined on a per-language feature basis below. The rest of this section focuses on the more generic and fundamental elements of the kernels, such as general organization, interface, and memory management.

5.7 GPU C Runtime Kernel

The main concern of a C runtime is managing memory and adequately handling access to the DWA system. Dyalog's DWA system permits us direct access to the underlying interpreter array format and memory manager. We could use this format directly but this will not work for GPU compute because the DWA interface connects array elements and header information in a way that makes GPU allocating them quite difficult, especially if we only want the elements on the GPU.

DWA has a specific array format, but we will delay specifying utility code for array handling until Section 6.6. In this section, we handle the following issues:

- DWA Initialization
- Header Structure
- Memory Management
- Datatype Management
- Error Reporting

We deal with the top-level error signalling behavior in this section, but for error signalling within functions, as well as arrays, module initialization function calls, and so forth, see the appropriate subsection of Language Features (Section 6).

5.7.1 The C Header

The first order of business is the main structure of the C runtime files and API. We could attempt to put all our runtime code into a single `kernel.c` file, but the result would require us to maintain includes in a way that prevents us from easily linking the include statements to each language feature implementation without encouraging needless duplicate includes. Instead, we assume that each langauge feature will be given its own C file and then we can manage includes independently. We will make use of a single `codfns.h` file that contains all the public entry points into the runtime.

34a ⟨*codfns.h* 34a⟩≡

```
#pragma once
⟨C runtime includes (never defined)⟩
⟨C runtime macros 35⟩
⟨C runtime enumerations 37b⟩
⟨C runtime structures 37a⟩
⟨C runtime declarations 39a⟩
```

Root chunk (not used in this document).
Defines:
`codfns.h`, used in chunks 24d, 34b, 42a, 48a, 89, 106a, 112a, 119a, and 140b.

34b ⟨*Tangle Commands* 8⟩+≡

```
echo "Tangling rtm/codfns.h..."
notangle -R'codfns.h' codfns.nw > rtm/codfns.h
```

This code is used in chunk 134.
Uses `codfns` 7 and `codfns.h` 34a.

Since we want to use this single header for the runtime code *and* the generated code that will import the runtime, an interesting situation arises regarding exports. Both generated and runtime code must export functions from their respective DLLs, but in the case of the runtime, these exported functions are also the functions that we must import into our generated code, we must annotate the edeclaration of such functions differently if we are importing than when we are exporting. Thus, when we are building the runtime, we want to export all our bindings, but when we are accessing the runtime from generated code we want to import those same bindings while exporting functions that we generate.

To handle this, we rely on three preprocessor definitions. When we are building the runtime, we will define `EXPORTING`, but we expect this to be undefined when building generated code. Then we have an `EXPORT` definition that always maps to the platform specific export decorator, while `DECLSPEC` will be the import spec or export spec depending on `EXPORTING`.

It used to be the case that each platform handled DLL importing and exporting differently, but modern compilers all handle the `__declspec` syntax, so we will use that for all platforms.

35 ⟨*C runtime macros* 35⟩≡

```
#define EXPORT __declspec(dllexport)
#ifdef EXPORTING
	#define DECLSPEC EXPORT
#else
	#define DECLSPEC __declspec(dllimport)
#endif
```

This code is used in chunk 34a.
Defines:
`DECLSPEC`, used in chunks 38–41, 44, 45, 47a, 86, 88a, 92, 97a, 105, 110, 111, and 119c.
`EXPORT`, used in chunks 25a and 119a.
`EXPORTING`, used in chunk 140a.

5.7.2 Memory and Datatype Management

Next, we deal with handling memory and multiple data types. Since the compiler assumes a stack machine model, we have a unified stack that will contain many different objects, such as functions and arrays, so we must have a way of handling the objects in a somewhat generic way.

While some generality is desirable, I must curtail my Scheme-esque impulse towards unnecessary dynamic generality. This is a runtime, after all, and experience shows that extra dynamic annotation can seriously impede scalability of the system and introduce unfortunate performance gotchas. Rather than chase this form of programmability, I am taking a page from Knuth's book and aiming for "re-editable" code that can be easily, but statically, extended. The goal is to avoid excess runtime allocation and indirection while at the same time making it easy to add and manage datatypes.

Any such memory or type management system must address the following questions:

- How do I make an object?
- How do I free an object?
- When do I free an object?
- How do I keep an object alive?
- How do I make new data types?

In APL, most values have a stack lifetime, which would encourage us to make use of a stack semantics in our runtime. However, for more involved APL, this assumption does not hold true. Instead, to manage our objects, we choose to make use of reference counting.[5] This maintains most of the predictability and low-overhead of a stack semantics but gives us the additional power to allow object lifetimes to extend beyond the lifetime of their definition context.

We do not have a requirement in our system for generic object creation (indeed, such a requirement is quite rare), but we do need to generically retain a reference to an object and to release an object. We want to enable this without too much indirection. To implement this, we simply require that all our datatypes be structures that share the following common fields. We call these types cells as a convenient term.

36 ⟨*Common cell fields* 36⟩≡

```
enum cell_type ctyp;
unsigned int refc;
```

[5] https://en.wikipedia.org/wiki/Reference_counting

This code is used in chunks 37a, 85b, 104b, and 109d.
Defines:
`ctyp`, used in chunks 38, 40a, 86, 105a, and 110.
`refc`, used in chunks 38, 39b, 41a, 86, 105a, and 110.
Uses `cell_type` 37b.

These fields help us to answer the two most important questions we must answer for any cell: what type of cell is it; and, is it currently referenced? By requiring all data structs to have these fields in common, we can cast them about and be basolutely sure that things will continue to work. We define a “void” cell type `struct cell_void` to be our minimal cell type.

37a ⟨*C runtime structures* 37a⟩≡
```
struct cell_void {
        ⟨Common cell fields 36⟩
};
```
This definition is continued in chunks 76a, 85b, 104b, and 109d.
This code is used in chunk 34a.
Defines:
`cell_void`, used in chunks 38–41.

The `enum cell_type` keeps track of all known cell types.

37b ⟨*C runtime enumerations* 37b⟩≡
```
enum cell_type {
        ⟨Cell type names 37c⟩
};
```
This definition is continued in chunk 85a.
This code is used in chunk 34a.
Defines:
`cell_type`, used in chunk 36.

We set the first $0th$ cell type to our void cell.

37c ⟨*Cell type names* 37c⟩≡
```
CELL_VOID
```
This definition is continued in chunks 84f, 104a, and 109c.
This code is used in chunk 37b.
Defines:
`CELL_VOID`, used in chunks 38 and 40c.

We do not make or define any generic way to create cells; you must make a constructor function suitable to the needs of the data type. At the moment, it is the responsibility of such makers to ensure that the common fields are appropriately initialized. A maker should return a `0` on success and a non-zero error on failure. It should also take a `struct cell_TYPE **` as the first argument to store the allocated cell in. We expect the slot passed to a creator will be a possibly previously utilized slot on a stack or something along these lines. This means that it is the caller's responsibility to ensure that this slot has already been released. Failure to do this would potentially lead to a memory leak. However, attempting to handle this within the cell maker function results in an API that is much too fragile and needlessly complex. We expect to generally follow the stylistic guideline that a function should allocate and own its own data and then release that data in the same function.

The basic cell maker for the `void` cell type looks like this:

38 ⟨*Cell definitions* 38⟩≡
```
DECLSPEC int
mk_void(struct cell_void **cell)
{
        struct cell_void *ptr;

        ptr = malloc(sizeof(struct cell_void));

        if (ptr == NULL)
                return 1;

        ptr->ctyp = CELL_VOID;
        ptr->refc = 1;
        *cell = ptr;

        return 0;
}
```
This definition is continued in chunks 39–41.
This code is used in chunk 42a.
Defines:
`mk_void`, used in chunk 39a.
Uses `CELL_VOID` 37c, `cell_void` 37a, `ctyp` 36, `DECLSPEC` 35, and `refc` 36.

A few points of style here. The error codes should try to follow the standard APL codes. Additionally, the target slot should not be mutated until we are sure that all is well and that the object is well-formed.

39a ⟨*C runtime declarations* 39a⟩≡
```
DECLSPEC int mk_void(struct cell_void **);
```
This definition is continued in chunks 39–41, 44, 45, 88a, 97a, 105b, 111a, 112c, and 119c.
This code is used in chunk 34a.
Uses `cell_void` 37a, `DECLSPEC` 35, and `mk_void` 38.

While we must define unique constructors for the various types, when releasing or freeing a cell of some kind, we *do* want to be able to generically free a cell. However, this must be done with a minimum of runtime overhead. First, we distinguish the terms "release" and "free". If an object is freed, that object's memory is fully returned to the memory manager, whereas releasing is about reducing the number of references to that object. When a cell has no references to it, then it is freed.

Each cell type will require its own unique release function that manages cleanly destroying the cell. The release function for the `void` cell type looks like this:

39b ⟨*Cell definitions* 38⟩+≡
```
DECLSPEC void
release_void(struct cell_void *cell)
{
        if (cell == NULL)
                return;

        if (--cell->refc)
                return;

        free(cell);
}
```
This code is used in chunk 42a.
Defines:
`release_void`, used in chunks 39c and 40c.
Uses `cell_void` 37a, `DECLSPEC` 35, and `refc` 36.

39c ⟨*C runtime declarations* 39a⟩+≡
```
DECLSPEC void release_void(struct cell_void *);
```
This code is used in chunk 34a.
Uses `cell_void` 37a, `DECLSPEC` 35, and `release_void` 39b.

To support generic cell release, we define a `release_cell` function.

40a ⟨*Cell definitions* 38⟩+≡

```
DECLSPEC void
release_cell(void *cell)
{
        if (cell == NULL)
                return;

        switch (((struct cell_void *)cell)->ctyp) {
        ⟨Cell release cases 40c⟩
        default:
                dwa_error(99);
        }
}
```

This code is used in chunk 42a.
Defines:
`release_cell`, used in chunks 40b, 105a, and 110.
Uses `cell_void` 37a, `ctyp` 36, `DECLSPEC` 35, and `dwa_error` 45a.

40b ⟨*C runtime declarations* 39a⟩+≡

```
DECLSPEC void release_cell(void *);
```

This code is used in chunk 34a.
Uses `DECLSPEC` 35 and `release_cell` 40a.

For each cell type, we must plug the type-specific release function into this `release_cell` switch to enable generic releasing for that type. For the `void` type, this looks as follows:

40c ⟨*Cell release cases* 40c⟩≡

```
case CELL_VOID:
        release_void(cell);
        break;
```

This definition is continued in chunks 88b, 105c, and 111b.
This code is used in chunk 40a.
Uses `CELL_VOID` 37c and `release_void` 39b.

The above mostly suffices for dealing with cells. However, we also want to conveniently bump the reference count of a cell seamlessly without explicitly setting `refc`. We often encounter the case where we are assigning a cell to a new slot, thus requiring a reference count increment. The following function `retain_cell` lets us do this in a single statment by writing:

```
slot2 = retain_cell(slot1);
```

41a ⟨*Cell definitions* 38⟩+≡

```
DECLSPEC void *
retain_cell(void *cell)
{
        if (cell != NULL)
                ((struct cell_void *)cell)->refc++;

        return cell;
}
```

This code is used in chunk 42a.
Defines:
`retain_cell`, used in chunks 41b, 83f, 98c, 102d, and 111c.
Uses `cell_void` 37a, `DECLSPEC` 35, and `refc` 36.

41b ⟨*C runtime declarations* 39a⟩+≡

```
DECLSPEC void *retain_cell(void *);
```

This code is used in chunk 34a.
Uses `DECLSPEC` 35 and `retain_cell` 41a.

Fortunately, this retention function requires no extra code as we extend the system with more data types. This gives us the following steps if we want to add a new data type to the runtime:

1. Add the cell type to ⟨*Cell type names* 37c⟩ as `, CELL_TYPE`.
2. Define the structure in ⟨*C runtime structures* 37a⟩, making sure that ⟨*Common cell fields* 36⟩ are the first fields.
3. Define an `int mk_type(struct cell_type **, ...)` function and declare it in ⟨*C runtime declarations* 39a⟩.
4. Define a `void release_type(struct cell_type *)` function and declare it in ⟨*C runtime declarations* 39a⟩.
5. Add a case to ⟨*Cell release cases* 40c⟩ on `CELL_TYPE` that calls `release_type` on `cell`.

The cell handling we put into a file on its own.

42a ⟨*cell.c* 42a⟩≡
```
#include <stdlib.h>

#include "codfns.h"

⟨Cell definitions 38⟩
```
Root chunk (not used in this document).
Defines:
`cell.c`, used in chunk 42b.
Uses `codfns` 7 and `codfns.h` 34a.

42b ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling rtm/cell.c..."
notangle -R'cell.c' codfns.nw > rtm/cell.c
```
This code is used in chunk 134.
Uses `cell.c` 42a and `codfns` 7.

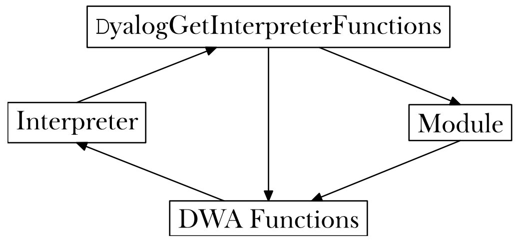

Figure 2: DWA module initialization

5.7.3 DWA Interface and Error Handling

Finally, we must handle the DWA connection between a Co-dfns compiled module and the interpreter. One constraint on this design is the need to make a Co-dfns module work with or without a DWA-driven interpreter. If we are interfacing solely with a foreign, C-based system, we still must function somehow.

DWA modules export `DyalogGetInterpreterFns` as a function to link the interpreter and the module.

The function receives a structure from the interpreter populated with function pointers that enable access to various interpreter features. A small design point comes into play here because we do not want to unnecessarily expose our underlying model to the user of the compiled module. In particular, if an user is not a Dyalog interpreter, they should not need to know about the DWA system in order to function. For example, they should not need to know or use `DyalogGetInterpreterFunctions` or the underlying functions. Thus, we must have a way to achieve similar functionality from different systems.

Our approach to this is to provide more generic and explicit function for setting things we want from any system and then to layer DWA initialization on top of that.

Fundamentally, the main thing that we care about for all systems is having some means of making a non-local escaping error report. This main error reporting is meant to mimic the extended signalling functionality of the interpreter documented in the `⎕DMX` object. The DWA equivalent of this structure is given by `struct dwa_dmx`.

43 ⟨*DWA structures and enumerations* 43⟩≡

```
struct dwa_dmx {
        unsigned int flags;
        unsigned int en;
        unsigned int enx;
        const wchar_t *vendor;
        const wchar_t *message;
        const wchar_t *category;
```

```
};
```

This definition is continued in chunks 46 and 98a.
This code is used in chunk 48a.
Defines:
`dwa_dmx`, used in chunks 44a and 45c.

In our APL model at the moment, there is only one main and universal ⎕DMX object at a time, so we define a single `dmx` binding to contain the current data.

44a ⟨*DWA definitions* 44a⟩≡

```
struct dwa_dmx dmx;
```

This definition is continued in chunks 44, 45, 47a, and 92.
This code is used in chunk 48a.
Defines:
`dmx`, used in chunks 20b, 44b, and 45a.
Uses `dwa_dmx` 43.

The reality of many FFI systems is that they do not do a good job of supporting C structs in the form of such global variables, so we must make sure that there is a meaningful way to access the system using nothing but function calls.

In the case of errors we have an interesting situation. In C, handling a long chain of errors demands that we are meticulous about how we handle the interaction of the call stack and any kind of early exit. In our case, this means that any time we finally call the non-local error function that we expect to never return, we may be quite far removed from the original site of the error. Thus, passing any complex data back up a call stack could be quite complex. Instead, we populate most of `dmx` that we care about using setter functions and then only have a very little to worry about passing up a call stack, namely, the error number itself.

we define a setter function `set_dmx_message` to handle setting `dmx.message`.

44b ⟨*DWA definitions* 44a⟩+≡

```
DECLSPEC void
set_dmx_message(wchar_t *msg)
{
        dmx.message = msg;
}
```

This code is used in chunk 48a.
Defines:
`set_dmx_message`, used in chunk 44c.
Uses `DECLSPEC` 35 and `dmx` 44a.

44c ⟨*C runtime declarations* 39a⟩+≡

```
DECLSPEC void set_dmx_message(wchar_t *);
```

This code is used in chunk 34a.
Uses `DECLSPEC` 35 and `set_dmx_message` 44b.

Our main non-returning function dwa_error handles some of the parts of dmx that we do not currently change, and then calls the internally initialized error function provided by whatever our interfacing system is.

45a ⟨*DWA definitions* 44a⟩+≡

```
DECLSPEC void
dwa_error(unsigned int n)
{
        dmx.flags = 3;
        dmx.en = n;
        dmx.enx = n;
        dmx.vendor = L"Co-dfns";
        dmx.category = NULL;

        dwa_error_ptr(&dmx);
}
```

This code is used in chunk 48a.
Defines:
dwa_error, used in chunks 29, 40a, 45b, and 86.
Uses DECLSPEC 35, dmx 44a, and dwa_error_ptr 45c.

45b ⟨*C runtime declarations* 39a⟩+≡

```
DECLSPEC void dwa_error(unsigned int);
```

This code is used in chunk 34a.
Uses DECLSPEC 35 and dwa_error 45a.

The above requires the calling interface set dwa_error_ptr, which we handle with set_codfns_error.

45c ⟨*DWA definitions* 44a⟩+≡

```
void (*dwa_error_ptr)(struct dwa_dmx *);

DECLSPEC void
set_codfns_error(void *fn)
{
        dwa_error_ptr = fn;
}
```

This code is used in chunk 48a.
Defines:
dwa_error_ptr, used in chunk 45a.
set_codfns_error, used in chunks 45d and 47b.
Uses DECLSPEC 35 and dwa_dmx 43.

45d ⟨*C runtime declarations* 39a⟩+≡

```
DECLSPEC void set_codfns_error(void *);
```

This code is used in chunk 34a.
Uses DECLSPEC 35 and set_codfns_error 45c.

To link this interface into the DWA functionality, we must extract the appropriate function pointers out of the structure passed to `DyalogGetInterpreterFunctions`. We assume that the code generator will create a suitable definition for `DyalogGetInterpreterFunctions` that calls the following `set_dwafns`, such as:

```
EXPORT int
DyalogGetInterpreterFunctions(void *fns)
{
        return set_dwafns(fns);
}
```

This established a link in each compiled module to the runtime DWA handling and allows us to keep the DWA logic inside the runtime. The DWA structure is relatively involved in its full expression, but we do not need the full power, so we can simplify our setup. We also want to talk about the structure more generically here without too much detail that may be more properly handled in the correct language feature section. At its heart, the structure is a set of functions, which we store as an array of `void *` pointers.

46 ⟨*DWA structures and enumerations* 43⟩+≡

```
struct dwa_wsfns {
        long long size;
        void *fns[18];
};

struct dwa_fns {
        long long size;
        struct dwa_wsfns *ws;
};
```

This code is used in chunk 48a.
Defines:
`dwa_fns`, used in chunk 47a.
`dwa_wsfns`, never used.

It is the job of the `set_dwafns` function to set the appropriate Codfns interface functions and follow the initialization expectations of the DWA system. On successful initialization, the function should return 0, but we must check compatibility by examining the given structure `size`, return `16` if something is not right.

47a ⟨*DWA definitions* 44a⟩+≡

```
DECLSPEC int
set_dwafns(void *p)
{
        struct dwa_fns *dwa;

        if (p == NULL)
                return 0;

        dwa = p;

        if (dwa->size < (long long)sizeof(struct dwa_fns))
                return 16;

        ⟨Set DWA interface functions 47b⟩

        return 0;
}
```

This code is used in chunk 48a.
Defines:
`set_dwafns`, used in chunk 24d.
Uses `DECLSPEC` 35 and `dwa_fns` 46.

Assuming that the DWA structure seems valid, we want to extract these functions into the appropriate names that we have created for them. An alternative would be to retain the structure and make indirect calls into that structure, but this is a little more awkward and would involve both more storage and more memory indirects for no more clarity and only more entanglement of the code. Instead, setting the correct names at the time of a `set_dwafns` call leads to a much cleaner dependency tree. At this point, only the `dwa_error` function has been designed and defined.

47b ⟨*Set DWA interface functions* 47b⟩≡

```
set_codfns_error(dwa->ws->fns[17]);
```

This code is used in chunk 47a.
Uses `set_codfns_error` 45c.

This covers the main global DWA handling, but we have more to do in other sections to handle DWA arrays and function calling. We benefit from having a few things together in a single C file, so we will store our DWA code in a single C file with an eye to making it easy to add in the appropriate code in later sections.

48a ⟨*dwa.c* 48a⟩≡

```
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <arrayfire.h>

#include "codfns.h"

⟨DWA macros 97b⟩
⟨DWA structures and enumerations 43⟩
⟨DWA definitions 44a⟩
```

Root chunk (not used in this document).
Defines:
`dwa.c`, used in chunk 48b.
Uses `codfns` 7 and `codfns.h` 34a.

48b ⟨*Tangle Commands* 8⟩+≡

```
echo "Tangling rtm/dwa.c..."
notangle -R'dwa.c' codfns.nw > rtm/dwa.c
```

This code is used in chunk 134.
Uses `codfns` 7 and `dwa.c` 48a.

6 Language Features

6.1 Comments and Whitespace

Early in the parsing process, we want to unify and simplify whitespace and comments in the code so that none of the future code has to worry about it. There are a few things to consider.

First, comments should be completely eliminated from the tree so that we never attempt to parse anything inside of a comment. we cannot make this our first step in the parser because character vectors may have `⍝` characters in them. It is okay to have "string-like" things within comments because we can safely ignore anything in a comment as long as we can reliably and accurately identify the semantically meaningful `⍝` characters from those in a string.

This makes comment parsing and character vector parsing an intertwined process. We must identify such strings first, which we can do by making a Boolean mask `msk` marking out the possible strings, but we cannot parse these yet because some of these may appear inside a comment and should not be parsed. Once we have the potential string regions, only all `⍝` outside of these regions must be the semantic comment starts. A little thought suffices to prove that no semantic comment can appear inside a potential string region: if a semantic marker was inside a potential string region, this would mean that there are no previous markers on the line, but that means that the string region must be a real region, and that means that the `⍝` character is not semantic.

We assume that we are still in our nested line representation at this point because strings and comments are line-local, so it is much easier to handle them in the nested form. Assuming that `msk` is the nested Boolean mask of potential regions, there are a few representations we could use, based on whether we include or exclude the leading or trailing `'` quote characters. Fortunately for us, this does not matter, since we are mostly interested in using `msk` to find the semantic `⍝` points. After that, we do not need `msk`. Since the start and end `'` characters will never match `⍝`, the search for semantic `⍝` characters is the same regardless. This allows us to filter `pos` and `msk` down with the following code. We do not need `end` yet because we do not care about the extra whitespace in our implied regions at the moment.

49 ⟨*Remove comments* 49⟩≡

```
pos msk⌿¨⍨←⊂∧⍀¨(~msk)⍲'⍝'=IN∘I¨pos
```

This code is used in chunk 21.

AFter handling comments, we must make sure that we have adequately checked our string syntax. After this, we still want to do a few more things to normalize the whitespace in our source. We want to normalize line endings by removing occurrences of `⋄` and using a single `Z` node to wrap all lines. We also want to reduce most of the clearly unnecessary whitespace so that we do not need to scan useless characters all the time during tokenization. We plan to eliminate all unneeded character nodes at the end of tokenization anyways, but there is no reason to make all the tokenization passes traverse so much blank space all the time.

After we have checked the syntax on character vectors, we no longer require the nested representation, but we find yet another interesting point of design. If we choose to handle `⋄` nodes before tokenizing strings, we are now free to do either, we must continue to use `msk` to make sure we do not match `⋄` characters that appear in strings. On the other hand, tokenizing strings is much more nicely expressed using a flattened representation. But when we go to eliminate whitespace, it might be nice to do this on a nested representation to gain access to the leading and trailing whitespace idioms.

In the end, I find it more objectionable to continue persisting the `msk` value longer than necessary, so my primary concern is to tokenize strings as quickly as I can instead of continuing to use the nested representation. This means flattening right away and then tokenizing strings right away. We can also observe that removing leading and trailing whitespace is simply a special case of removing duplicate or insignificant whitespace anywhere in the source. Once strings are tokenized, we are free to eliminate insignificant whitespace from anywhere in the source at once. This more general approach has a much richer invariant at the end of it anyways. This makes the case for early flattening a slam dunk.

Flattening takes the nested representations of `pos` and `msk` and converts them into simple arrays. When doing this we must retain the line divisions somehow. To do this, we introduce the `t` field to give a type to each character, which we now begin thinking of more like nodes in a fully flat and unconnected forest. We use type `0` for unparsed character data, but introduce our first type to represent a line, `Z`. We will continue to think of `Z` nodes as “miscellaneous container” nodes. At this point, we put a `Z` node as the start of each line, pointing to the first character of the line, given by `⊃¨pos`.

50 ⟨*Flatten parser representation* 50⟩≡

```
t←⊃0⍴⊂pos
t pos msk(∊,∘⍪⍨)←Z (⊃¨pos) 0
```

This code is used in chunk 21.

After strings have been appropriately tokenized, we are free to handle the final main points, which are to eliminate insignificant whitespace and to make all ⋄ characters into Z nodes. The latter is trivial.

51a ⟨*Convert ⋄ to* Z *nodes* 51a⟩≡
```
t[⍸'⋄'=IN[pos]]←Z
```
This code is used in chunk 21.

Eliminating insignificant whitespace is not as cut and dry. There is the question of how much to remove. We think the benefit of knowing that all whitespace is isngificant further down the compiler pipeline is a nice enough invariant to have that it is worth pursuing, not to mention the inherent increase in efficiency.

We observe that knowing that a group of spaces is insignificant requires knowing what is on the right *and* the left. It does not suffice to know only one side. It would be possible to compute this all at one go, but we can make this much easier by first reducing all contiguous spaces down so that there is no contiguous whitespace. This will ensure that it is much easier to check the left and right sides.

First, we must define what we consider valid whitespace. In this case, all newlines should have already been converted into Z nodes, and as far as I can tell, APL does not permit more exotic forms of whitespace in the source. That leaves only tabs and spaces.

51b ⟨*Define character classes* 51b⟩≡
```
WS←⎕UCS 9 32
```
This definition is continued in chunks 52–54.
This code is used in chunk 21.
Defines:
WS, used in chunks 22, 51c, 52a, and 55.

Now we should eliminate any contiguous whitespace characters. One thing we must remember at this point is how we must handle Z nodes. We must make sure not to eliminate any Z nodes, which might happen if we only check the value of IN[pos] because pos for a Z node is likely to point to a whitespace character. Contiguous whitespace is simply whitespace that has whitespace to its left. We could also define it as right instead of left, but defining it as left will have the nice side effect of removing all leading whitespace. Since a typical APL source should have more leading whitespace than trailing, editors often automatically remove trailing whitespace, this seems like a nice free win.

At this point, we have to update three fields: t, pos, and end.

51c ⟨*Remove insignificant whitespace* 51c⟩≡
```
t pos end⌿⍨←⊂~(t=0)∧(¯1⌽IN[pos])∊WS
```
This definition is continued in chunk 52a.
This code is used in chunk 21.
Uses WS 51b.

White the contiguous whitespace removed, we can focus on eliminating insignificant whitespace. The only way for a space to be significant is for both its left and right neighbors to be non-breaking or merging characters. In APL, these are alphabetic characters, digits, `¯`, `⍺`, `⍵`, `.`, and `⎕`. We do not need to get this absolutely perfect because tokenization will handle that; this is just to remove obvious excess before continuing.

52a ⟨*Remove insignificant whitespace* 51c⟩+≡
```
msk←⊃1 ¯1∧.((alp,num,'¯⍺⍵⎕.')∊⍨⌽)⊂x←IN[pos]
t pos end⌿⍨←⊂msk∨(t≠0)∨~x∊WS
```
This code is used in chunk 21.
Uses `alp` 52b, `num` 52c, and `WS` 51b.

6.2 Valid source input character set

An APL source should contain only a limited set of valid characters outside of character vectors and comments. We want to verify this is the case as early in the parser as possible since this limited character set is quite useful in the rest of the parser. However, we must do this after tokenizing away any character vectors and comments to avoid false positives on their contents.

While we are validating the characters, it is also a good time to classify various characters into their appropriate categories. We do that first. Most obvious is the set of alphabetic characters. These are the characters in addition to numeric digits that constitute valid characters for variable names.

52b ⟨*Define character classes* 51b⟩+≡
```
alp←'ABCDEFGHIJKLMNOPQRSTUVWXYZ_
alp,←'abcdefghijklmnopqrstuvwxyz'
alp,←'ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝß
alp,←'àáâãäåæçèéêëìíîïðñòóôõöøùúûüþ'
alp,←'∆⍙ⒶⒷⒸⒹⒺⒻⒼⒽⒾⒿⓀⓁⓂⓃⓄⓅⓆⓇⓈⓉⓊⓋⓌⓍⓎⓏ'
```
This code is used in chunk 21.
Defines:
`alp`, used in chunks 52a, 55, 72a, and 83a.

The numbers should get their own unique class.

52c ⟨*Define character classes* 51b⟩+≡
```
num←⎕D
```
This code is used in chunk 21.
Defines:
`num`, used in chunks 52a, 55, 64, 72, and 83a.

Next are the syntax characters. These are characters that exist primarily as non-primitive annotations mainly useful in parsing, they may also represent components of compound tokens. We split these into two classes: class `syna` are the characters that may form more compound units, but that generally represent atomic values absent other context; class `synb` contains the rest, including `⍺` and `⍵`.

53 ⟨*Define character classes* 51b⟩+≡
```
syna←'⍬⎕⍞#'
synb←'¯[]{}()'':⍺⍵⋄;'
```
This code is used in chunk 21.
Defines:
  `syna`, used in chunks 55 and 98d.
  `synb`, used in chunk 55.

Primitives are a little more complex. Fortunately, all our primitives are essentially single characters[6], but we must handle operators with more care. Primitives that we always treat as functions present little challenge, but primitives that are operators or that may behave like operators are another story. The so-called schizophrenic primitives actually present the least trouble; we must parse them uniquely so they will get their own class. For the more mundane operators, though, the legitimate design space is larger. Obviously, there are monadic and dyadic operators, but we could treat them all as a single class. In the case of parsing, *not* distinguishing monadic and dyadic strikes me as a grand mistake, especially since these two sets are exclusive to one another. But, there are dyadic operators that take more than one type of operand, that is, they accept array and function operands. Do we want to split this classification up to differentiate operand types as well as operator arity?

I say no. The reasoning is simple. Operators as a class will always be exclusively divided by arity, but there is much less clean division of operand type classifications. Moreover, if we make the distinction at parse time, we must also handle user-defined operators somewhat uniquely. The end result is a vastly expanded state space for the problem with the only real benefit being a slightly earlier error message about operator type errors. It is not at all clear that this is even a good thing. We will not alter the parse tree in any way by choosing not to distinguish based on operand type, but we gain the ability to treat user-defined operators as the same class as primitive operators, greatly reducing the state space without loss of overall fidelity.

Finally, since we will be handling assignment uniquely anyways, we can just treat it as a function primitive for most cases without much trouble. This gives the following definitions.

54 ⟨*Define character classes* 51b⟩+≡

```
prmfs←'+-×÷|⌈⌊*⍟○!?~∧∨⍲⍱<≤=>≥≠
prmfs,←'≡≢⍴,⍪⌽⊖⍉↑↓⊂⊆⊃∊⍷∩∪⍳⍸⌷⍋⍒⍎⍕⊥⊤⊣⊢⌹∇←→'
prmmo←'¨⍨&⌶⌸'
prmdo←'∘.⍣⍠⌺⍤⍥@'
prmfo←'/⌿\⍀'
prms←prmfs,prmmo,prmdo,prmfo
```

This code is used in chunk 21.
Defines:
`prmdo`, used in chunk 99a.
`prmfo`, used in chunk 99a.
`prmfs`, used in chunk 99a.
`prmmo`, used in chunk 99a.
`prms`, used in chunks *55* and 98d.

[6] Ignore `∘.` for the moment, or imagine it as an application of `.` if it makes you feel better.

With the character classes defined, we can verify that all characters outside of strings are valid. We must remember to include `WS` in this set.

55 ⟨*Verify that all open characters are valid* 55⟩≡

```
  ∨⌿msk←~IN[pos]∊alp,num,syna,synb,prms,WS:{
          EM←'SYNTAX ERROR: INVALID CHARACTER(S) IN SOURCE',CR
          EM,←quotelines ⍸msk
          EM ⎕SIGNAL 2
  }⍬
```

This code is used in chunk 21.
Uses `alp` 52b, `num` 52c, `prms` 54, `quotelines` 20b, `SIGNAL` 20b, `syna` 53, `synb` 53, and `WS` 51b.

6.3 Strings and characters

APL has a single string syntax. As an atomic unit that exists at much the same level as that of a number, the main impact of string support occurs in the parser, code generator, and runtime primitives. It has minimal impact on the main compiler transformations.

Taking a high level view, we want to parse, compile, generate, and work in the runtime with strings. At the compile level, we should make it so that strings are handled in the same way that any simple array is handled. Likewise, handling character arrays should mostly work just the same as any other array as long as we have an appropriate type tag. It is in parsing that the most work is required. We must also ensure that we can properly convert the data into a good runtime representation during code generation.

Strings must be handled early on in the parser, since a character vector may contain all sorts of content, making it almost impossible to parse most other content without first parsing strings. However, comments also have this feature, and we must intertwin the parsing of strings with the parsing of comments. The fundamental issue is that comments may hold things that look like strings and strings may enclose things that look like comments. In principle, the first marker, either `'` or `⍝`, takes precedence, so we must figure out how to do that. Since comments completely block out all the rest of a line, the most information comes from checking each line for all things that look like strings first. Then, we can look for any comment markers that are not inside of strings and use that to eliminate any strings that are really just inside of comments. We can accomplish this on the nested `pos` representation using the common `≠⍀` idiom to produce `msk` that is used in the previous section. We must also remember to mark the double quotes separately to handle escaped quotes.

56 ⟨*Mask potential strings* 56⟩≡
```
  msk←(''''''∘⍷¨x)∨≠⍀¨''''=x←IN∘I¨pos
```
This code is used in chunk 21.

Once that is done, we must eliminate comments so that we can continue to parse the strings that are “real.” Before tokenizing the strings, we must check that they are balanced on a line. Since we are still using the nested representation at this point, we can do this pretty easily by checking the end of the line for any open strings. We should report all unbalanced string ranges that we find.

57a ⟨*Check for unbalanced strings* 57a⟩≡

```
0≠≢lin←⍸⊃∘⌽¨msk:{
        EM←'UNBALANCED STRING',('S'/⍨2≤≢lin),CR
        2 EM SIGNAL ∊(msk⌿¨pos)[lin]
}⍬
```

This code is used in chunk 21.
Uses SIGNAL 20b.

The msk value now contains well-formed strings in the source, ready for tokenization. It is nicer to do the tokenization on a flat representation, so we wait to perform this next step until we have flattened pos and msk. This means we have t to worry about, too.

At this point, after tokenizing strings, it will no longer be the case that we can think of each pos as pointing to a single character modulo whitespace. That makes this a good time to introduce the end field. To begin with, we will assume that all nodes point to a single character.

57b ⟨*Tokenize strings* 57b⟩≡

```
end←1+pos
```

This definition is continued in chunk 58.
This code is used in chunk 21.

We now must consider how we want to handle a string's node type in `t`. Thinking to what we want, eventually, we want all simple arrays to match in type. But at this moment, there is no real concept of the array as such, and there really will not be until we appropriately handle stranding. At that point, we can imagine a single array type with sub-kinds. At this point, we really have tokens and not any specific sub-typed AST structure. Thus, we want to avoid needing to introduce the `k` field for as long as possible within the parser. To do this, we will give tokens that are atomic, such as numbers, strings, and the `syna` class, their own node types until we have an appropriate conceptual representation for unifying them later on. In the case of strings, we will assign them type `C`.

We should take a moment to consider a few things: what node to convert to type `C`, where to begin the string region for `pos`, and where to end it with `end`. I think it makes the most sense to include the opening and closing quote characters in the range of the token, for at least two reasons. First, if we are using the `pos` and `end` data for something like syntax highlighting or text editing, it makes more sense for the whole unit to highlight; editing a string, say, to delete it, does not make much sense without the quotes, especially if we want to think of this as a single atomic unit. Additionally, if `IN[pos]` for a string points to `'` instead of an element inside of the string, we can know the token by its `pos` value as well as by its type `t`. This can make our future calculations simpler. Finally, we can avoid the somewhat problematic case of an empty string resulting in `pos = end`.

The starting point in this case is already pointing in the right spot if we choose to use the opening quote node as our new `C` node, meaning that we need only update `t` and `end` and not `pos`, so we will use that node. Our flattened `msk` value defined above will put a `1` in the opening quote position, and a `0` in the closing quote position, and `1`'s in all the string content positions. This makes it easy to use the `2<.⌿` and `2>⌿` idioms to select the opening and closing quote positions.

58a ⟨*Tokenize strings* 57b⟩+≡
```
t[i←⍸2<⌿0⍪msk]←C
end[i]←end[⍸2>⌿msk⍪0]
```
This code is used in chunk 21.

Once the `end` field is right, we no longer require the rest of the string nodes as elements in the forest, allowing us to remove them and free up space while hiding string data visibility, thus completing tokenization. We also no longer need the `msk` data, so we can let it go.

58b ⟨*Tokenize strings* 57b⟩+≡
```
t pos end⌿⍨←⊂(t≠0)∨~¯1⌽msk
```
This code is used in chunk 21.

And this is basically all that must be done to handle strings in the parser, assuming our array handling adequately unifies all the atomic elements into the appropriate simple and stranded array representations. However, our choice to make the `pos` and `end` fields contain the opening and closing quotes in a string means that we must process the `n` field of `C` nodes when it is created.

59 ⟨*Type-specific processing of the* `n` *field* 59⟩≡

```
n←⍎¨@{t=C}n
```

This definition is continued in chunk 74.
This code is used in chunk 22.

Table 1: Type associations between APL, C, ArrayFire, and Co-dfns

APL	C	ArrayFire	Co-dfns
80	uint8_t	u8	ARR_CHAR8
160	uint16_t	u16	ARR_CHAR16
320	uint32_t	u32	ARR_CHAR32

So much for parsing, and, indeed, compilation. Next, we must handle code generation. By the time we reach the code generator, the `C` nodes ought to have disappeared and all simple and strand arrays ought to belong to the same `A` type. Since array handling code is mostly common across all element types, we handle those elements in Section 6.6 on arrays; our only responsibility in this section is the unique processing necessary to deal with the character element type(s). For the generator, we must be able to map literal character arrays to a `data[]` array with an appropriate C data type. We must also map this to an appropriate `enum array_type`. This assumes that we will use this logic in some kind of "make data" helper that receives element data as a vector and generates the appropriate code.

Handling text in the compiler is an interesting problem because almost all Unicode-based text encodings have some sort of variable length aspect to them. In most languages that is fine, but for APL, which has random access indexing built in to its overall paradigm, the lack of random access-ness in these encodings is not really acceptable. That means we might want to take the UTF-32 model, but for most of the common cases, that is massively inefficient just to be able to handle the edges. Instead, we will adopt the same scheme that the Dyalog interpreter uses, which will have the added benefit of buffer compatibility between the two representations.

The Dyalog interpreter preserves arbitrary indexing at the cost of needing 3 data types for characters. Table 1 shows the APL types mapped to the C type we will use in the runtime. Rather than store textual data in one of the UTF encodings, this representationstores literal code points from Unicode, and it uses the smallest byte size that it can to store all code points in a buffer with the same element size. That is, if all the points in an array are in the range `[0,2*8)`, then we only need to use the 8-bit datatype, and so on.

We also make a note here that there is another datatype in the interpreter used to handle Classic character array types, but we are explicitly not going to support any Classic edition features or data types. To do the mapping, we will use `⎕DR` for testing the data type, but `⎕UCS` in its monadic form to get the code points.

60 ⟨*Element* `data` *and* `type` *generator cases* 60⟩≡

```
3>i←80 160 320⍳⎕DR ⍵:{
```

```
        bits←⍕i⊃8 16 32
        points←⊃{⍺,',',⍵}/⍕¨⎕UCS ⍵
        z←⊂'uint',bits,'_t data[] = {',points,'};'
        z,←⊂'enum array_type type = ARR_CHAR',bits,';'
z}⍵
```

This definition is continued in chunks 75 and 76b.
Root chunk (not used in this document).
Uses `array_type` 85a.

This ensures that the code generator can produce character data, but we must also add support for characters into the runtime proper. This means:

- Defining appropriate `enum array_type` values
- Mapping these values to the corresponding ArrayFire types
- Supporting conversion to/from DWA arrays

We used the following array types.

61a ⟨*Array element types* 61a⟩≡

```
, ARR_CHAR8, ARR_CHAR16, ARR_CHAR32
```

This definition is continued in chunk 77a.
This code is used in chunk 85a.
Defines:
`ARR_CHAR16`, used in chunks 61 and 62.
`ARR_CHAR32`, used in chunks 61 and 62.
`ARR_CHAR8`, used in chunks 61 and 62.

These array types must also map to ArrayFire types so that we can allocate them appropriate on the GPU. ArrayFire does not have any notion of character types, so we will use their unsigned types as a useful alternative.

61b ⟨*Cases for selecting device* `values` *dtype* 61b⟩≡

```
case ARR_CHAR8:
        dtype = u8;
        break;
case ARR_CHAR16:
        dtype = u16;
        break;
case ARR_CHAR32:
        dtype = u32;
        break;
```

This definition is continued in chunk 77b.
Root chunk (not used in this document).
Uses `ARR_CHAR16` 61a, `ARR_CHAR32` 61a, and `ARR_CHAR8` 61a.

In addition to getting data into an ArrayFire array, we must also be able to get data in and out of DWA arrays. Fortunately, this is fairly easy at this level because our element types are all the same size as the Dyalog interpreter's element types. This means that we do not need to do any pre- or post-processing of the incoming our outgoing data before we simply copy it over. And htat means the only thing we need to do is to map between DWA character types and Co-dfns character types.

Dyalog has these DWA character types:

62a ⟨*DWA character types* 62a⟩≡

```
, DWA_CHAR8, DWA_CHAR16, DWA_CHAR32
```

Root chunk (not used in this document).
Defines:
`DWA_CHAR16`, used in chunk 62.
`DWA_CHAR32`, used in chunk 62.
`DWA_CHAR8`, used in chunk 62.

All that remains to to map these to and from Co-dfns types.

62b ⟨*Cases for selecting* `type` *based on DWA type* 62b⟩≡

```
case DWA_CHAR8:
        type = ARR_CHAR8;
        break;
case DWA_CHAR16:
        type = ARR_CHAR16;
        break;
case DWA_CHAR32:
        type = ARR_CHAR32;
        break;
```

This definition is continued in chunk 81.
Root chunk (not used in this document).
Uses `ARR_CHAR16` 61a, `ARR_CHAR32` 61a, `ARR_CHAR8` 61a, `DWA_CHAR16` 62a, `DWA_CHAR32` 62a, and `DWA_CHAR8` 62a.

62c ⟨*Cases for selecting* `type` *based on array type* 62c⟩≡

```
case ARR_CHAR8:
        type = DWA_CHAR8;
        break;
case ARR_CHAR16:
        type = DWA_CHAR16;
        break;
case ARR_CHAR32:
        type = DWA_CHAR32;
        break;
```

This definition is continued in chunk 82.
Root chunk (not used in this document).
Uses `ARR_CHAR16` 61a, `ARR_CHAR32` 61a, `ARR_CHAR8` 61a, `DWA_CHAR16` 62a, `DWA_CHAR32` 62a, and `DWA_CHAR8` 62a.

And this concludes all the handling necessary to put characters into the system. It only deals with characters, and you need to read Section 6.6 to see how it all fits together. You can also see the numeric handling in Section 6.4 for a treatment of element types that are not as simple and straightforward.

6.4 Numbers

Supporting numbers in the compiler is a little more complex than handling characters because there are many more numeric forms than there are character forms, but the main strategy remains the same: we must tokenize and parse them into `N` nodes, ideally ignore them in the compiler transformations, and then generate the appropriate data and add datatype support for them in the runtime.

Our aim at the moment is to support the primary numeric types supported by Dyalog, as given in Table 2.

Table 2: Supported numeric datatypes and their equivalencies

APL	Co-dfns	C	ArrayFire	Convert?
11	`ARR_BOOL`	`char`	`b8`	Yes
83	`ARR_SINT`	`int16_t`	`s16`	Yes
163	`ARR_SINT`	`int16_t`	`s16`	No
323	`ARR_INT`	`int32_t`	`s32`	No
645	`ARR_DBL`	`double`	`f64`	No
1289	`ARR_CMPX`	`struct apl_cmpx`	`c64`	No

Notice that we do not support 128-bit decimal floats at the moment and that we treat type 83 like type 163 and type 11 as type 83 when we represent these values on device or in the runtime. This is because ArrayFire does not have any internal support for bitvectors or signed 8-bit integers. This has some performance and space considerations for user code, but in my previous experience, it is not even clear how much benefit we may get from using bitvectors on GPU acceleration devices. At any rate, the juice isn't worth the squeeze at this moment in terms of implementation complexity.

APL also has a number of syntaxes for specifying numeric values. We want to support the syntaxes in Table 3.

When handling numbers, we can imagine a future in which we support more types and more syntaxes than Dyalog APL may support, and in that case, it is important that we do not depend on the numeric parser built into the interpreter. Even with the numeric types overlapping entirely, we must ensure that we do not somehow lose precision that we may want. the benefit of using the built-in interpreter at the time of parsing is that all the numbers come into a

Table 3: Numeric APL syntaxes and whether they are C compatible

Form	Type	C compatible?
`123`	Integer	Yes
`¯123`	Signed	No
`12.3`	Floating	Yes
`¯12.3`	Signed float	No
`NeN`	Exponent	Yes
`NjN`	Complex	No

specific reified form rather than remaining in their textual form, so we can save a lot of effort simply by doing so, as long as we maintain precision. This also has the advantage of making the runtime generation code simpler, since we do not need to support so many numeric forms. this has enough advantages to make it worth doing internally, but we must still tokenize the numbers ourselves and manage the code generation.

The first step is tokenizing the numbers into atomic units. here, we must recognize the various numeric forms and distinguish numbers from digits used in names. By doing this before tokenizing things like variables, we can simplify the parsing of those other entities.

When we begin to tokenize, `x` contains the relevant characters we need to examine. The first challenge is to identify the clusters of digits that will form the core units of our numeric forms. Vitally, we want to distinguish digits that contribute to a number from those that are part of some kind of name. In APL this is more subtle than you may think, because a name may have, but not begin with, digits in it, which is typical, but a digit that is contiguous to a name and before it, such as `5x5` is *not* an error! Rather, it is parsed the same as `5 x5`, meaning a number followed by a name. This means we must eliminate digit clusters that are contiguous to alphabetic forms only on the right side. It gets a little more complex hwen we consider `e` and `j` forms. Our basic rule is that we have a precedence of `digits → . → ¯ → e → j`. Consider `1e1e1`. Here, this should be parsed as `1e1 e1`. This is because numeric forms are greedy (left associative) and you may only have a single `e` or `j` per unit at each level. This provides an order for handling the creation of `dm`, which should be a mask for all number groups. We can begin by initializing `dm` to all the possible numeric digits.

64 ⟨*Tokenize numbers* 64⟩≡

```
dm←x∊num
```

This definition is continued in chunks 67, 70, 72, and 73.
This code is used in chunk 21.

Defines:
`dm`, used in chunks 67, 70, 72, 73, 83a, 98d, and 108a.
Uses `num` 52c.

Our plan is to progressively expand `dm` to encompass all the elements of a valid numeric form while removing or eliminating any digits that belong to names.

Table 4: Precedence of numeric syntax and parse order

`dm` phase	Syntax	Notes
0	[0 – 9]	Must appear at least once
1	`.`	Only one per phase 0 group
2	`¯`	Must prefix phase 1 groups, one only
3	`e`	Must connect two phase 2 groups
4	`j`	Must connect two phase 3 groups

This progressive expansion serves to form a set of “phases” for parsing the numeric forms. The only syntax that must appear in a number are the numeric digits themselves; all other forms are optional. If we proceed one phase at a time, we can check for errors in the simpler forms before adding more complexity.

Since `e` and `j` are both valid variable name components, we have a potential conflict between numeric forms and names. APL resolves this by requiring the numeric `e` and `j` forms to have two numbers, one on each side, contiguous to it and only allowing one `e` or `j` per unit, though a single `j` unit may contain two `e` units. When combined with the greedy parsing rule for these forms, meaning the leftmost `e` or `j` is used for a number if more than one `e`/`j` appears contiguous to the same phase 2 number group.

All of this means an interesting parsing dependency: there may be some sequences that look like numbers, but actually are part of a name, but we do not know this until we are able to identify what `e` and `j` characters are part of a number or not. Fortunately, this issue does not cause problems because we can tackle it in much the same way as we handled strings and comments. Once any character in a sequence of alphanumeric characters definitely is a part of a name and not a number, all the subsequent characters in the sequence must be part of the same name and not a number. This is much like how a comment works. Thus, we can begin by identifying all potentially numeric `e` and `j` characters, eliminate those that are not the first in their units, and finally mask off potential numbers that actually form a part of a name.

The first syntax we should add is the dot. This is a blessedly simple form because there can be only a single occurrence, it may appear anywhere contiguous to a digit, and if we find more than one, we know that we can signal an error. We can only have these nice guarantees right now because we are not considering `e` or `j` at this point; we are only dealing with the smallest and most tightly bound compound number, which is the floating-point value.

We can add any dot we find if it is on either side and contiguous to a digit, which is just dm at the moment.

67a ⟨*Tokenize numbers* 64⟩+≡
```
dm∨←('.'=x)∧(¯1⌽dm)∨1⌽dm
```
This code is used in chunk 21.
Uses dm 64.

Now dm contains the digits and any contiguous dots. Before proceeding, we should verify that we do not have multiple dots in a single group.

67b ⟨*Tokenize numbers* 64⟩+≡
```
∨⌿msk←1<+⌿¨dm⊆'.'=x:{
        EM←'MULTIPLE . IN FLOAT'
        2 EM SIGNAL ∊msk/dm⊆pos
}⍬
```
This code is used in chunk 21.
Uses dm 64 and SIGNAL 20b.

Now we can add the high minuses. We can only have a single high minus in a numeric group. Thankfully, this also means that we do not permit something like ¯1¯2, as that would greatly complicate things. When checking for well formed syntax, we must check for duplicates/multiples in the same way as for dots, but we must also handle orphaned high minuses, since nay high minus that does not attach to a dm group must be a syntax error.

67c ⟨*Tokenize numbers* 64⟩+≡
```
dm∨←('¯'=x)∧1⌽dm
∨⌿msk←1<+⌿¨dm⊆'¯'=x:{
        EM←'MULTIPLE ¯ IN NUMBER'
        2 EM SIGNAL ∊msk⌿dm⊆pos
}⍬
∨⌿msk←('¯'=x)∧~dm:{
        EM←'ORPHANED ¯'
        2 EM SIGNAL msk⌿pos
}⍬
```
This code is used in chunk 21.
Uses dm 64 and SIGNAL 20b.

So much for the simple phases. Now we must handle the more complex compound cases for `e` and `j`. It is at this point that the wheels come off a little bit. The tokenizer in the Dyalog interpreter does very little lookahead. As a result, there were some interesting design decisions made to support dot and `e`/`j` since these are overloaded forms. We have already embraced the idea that multiple dots near digits should be treated as an error. However, with `e`, since the exponent to `e` must be an integer and not a float, we have the strange result, in the interpreter, that `1e0.5j3` and `1e.5j3` parse differently. In the first case, `0.5` binds to the `e` and results in a syntax error, but the second case parses without error as `1 e 0.5j3`! This is because, at the time of seeing the dot, the tokenizer recognizes that the `e.` combination can never lead to a valid parse for `e` as a number, and so it decides that `e` must be part of a name instead, and parsing then continues with recognizing the complex `0.5j3`.

This is madness.

I cannot in good conscience replicate this behavior into Co-dfns. Thus, I intend to deviate from this behavior, and so I will spend sometime justifying my decision for posterity.

The primary problem is that the behavior breaks cognitive predictability, which can be seen by how such behavior violates our parsing precedence tower (see Table 4). This makes the parsing code more idiosyncratic, mushes all the numeric forms into a much less crisp tower, and, perhaps worst of all, reduces or eliminates any human model of parsing based on a more chunked or abstract form, forcing the human mind to parse at a character-by-character operational level, which is most certainly *not* what APL is about.

The root of all of this is that Dyalog APL thinks of `0.5` as a character stream and not as a number. Instead, by the time we think about `e`, we should no longer worry about whether a dot is part of a number or not. Given `e.5`, there is no possible context in which we may want `0.5` to be anything but a number. This is how most people will parse this, and they ought to be able to do so. Now, given this, I should not expect `0.5` to parse differently than `.5` in my code. Anywhere, this should be a number, because the dot binds strongly to numbers. Anywhere that this model breaks ought to result in a syntax error as an ambiguous piece of code. Look at the following scary examples from the interpreter:

```
1e0.5j3 → SYNTAX ERROR
 e0.5   → e0 0.5
1e.5j3  → 1 e 0.5j3
```

Oh, the horror! In my opinion, I have two choices. I can take the attitude of attempting to parse these any way that I can, or, I can introduce a syntax error for all the cases that do not make sense to me. Given that I want to make the compiler produce more helpful errors and be a more congruent and consistent system, I am of a mind to treat these as syntax errors. In the future, I can imagine supporting floating exponents, and that would permit the `1e0.5` and `1e.5` cases. The `e0.5` case strikes me as something that should always be an error. The principle in play here is that `D.D` should always parse as a number. The other stuff comes out of this.

How does this affect our handling of `e` and `j`? The first impact is one of simplification; any time we encounter `e` or `j` contiguous with a set of decimal numbers, we can confidently parse this as an exponent or complex form. This also has the effect of committing at this point to the decimal forms we already see in `dm`. That is, if they have a dot in them, we know that they must be a number or a syntax error. This makes the reasoning a little simpler, but it adds an additional syntax error case that we must handle. We must check and handle the case where a float appears as an exponent and also where a name is contiguous to a number.

Handling these requires that we know what `e` and `j` characters map to numeric forms and which do not. There remains a question of whether we should handle the contiguous name error now or later. We can first note that the name error must come from any name contiguous with a number and not just `e` or `j`. We can also make the observation that the name error remains valid even after we theoretically parse `e` and `j` forms. As long as a float is somewhere in a numeric form, we have this error possibility. The challenge is when we should error on a float exponent vs. an ambiguous name/number situation. Given that we must do most of the work to parse `e` and `j` anyways to handle the name error, it makes sense to put it after we have mostly parsed those, but by that time, we also could error on a float exponent.

I think it would be confusing to start a complaint about the contents of a number before confirming all number parses are unambiguous. Moreover, at least in theory, all the numbers are just potential numbers until we mask out the numbers that are really a part of names. So verifying the forms of `e` should occur much later. Before that, we must complete the rest of the numeric parsing.

Handling `e` and `j` is the same basic thing. We must find and mark potential `e`'s and `j`'s. We must make sure that each `dm` group matches against only a single `e` or `j`. So, a string of `e`'s such as `1e1e1` is the same as `1e1 e1`. We must do the `e`'s separate from and before the `j`'s are handled. This allows us to maintain the invariant that a `dm` group will only contain a single `e` at the time we handle `e`. After dealing with `j`, a `dm` group may contain more than one `e`.

While we must make `j` and `e` separately, we can mask off names as a unit later. This is safe to do because both `e` and `j` forms have numbery things on either side. This means that any potential `j` we find and mark will still be eliminated by the later masking, so there is no need to attempt to eliminate such false positives earlier on right after marking the `e`'s. Likewise, we will not miss any `j` forms for the same reason. By delaying the masking, we make it more convenient to handle and we can separate our concerns about masking and error handling from concerns about handling `e` and `j`. All we need concern ourselves with at this point is identifying `e` forms that would be `e` forms unless they are part of a name.

The approach is to check for an `e` contiguous to two `dm` groups. After this, we must eliminate as candidates all but the first `e` in any one `dm` group.

70a ⟨*Tokenize numbers* 64⟩+≡
```
dm∨←(msk←x∊'Ee')∧(¯1⌽dm)∧1⌽dm
dm←dm⍀∊{1@(⊃⍸ω)~ω}¨dm⊆msk
```
This code is used in chunk 21.
Uses `dm` 64.

And we must do the same basic thing to handle the `j` forms.

70b ⟨*Tokenize numbers* 64⟩+≡
```
dm∨←(msk←x∊'Jj')∧(¯1⌽dm)∧1⌽dm
dm←dm⍀∊{1@(⊃⍸ω)~ω}¨dm⊆msk
```
This code is used in chunk 21.
Uses `dm` 64.

Table 5: Parsing cases for dot; `V` = name, `N` = number, `D` = digit

Pattern	Class
`V.V`	Inner Product
`V.N`	Numeric
`VD.N`	Ambiguous
`N.V`	Numeric
`N.N`	Numeric

Since we have done the above in meticulous order and captured only the `e` and `j` forms that make syntactic sense, there are no syntax errors to signal at this point. Furthermore, `dm` now contains the potential numbers of our source, and all that remains is to figure out which ones are names instead of numbers.

The main syntax error we want to address at this point is the ambiguous parsing that we may get because of a dot. If we have a mask of names, it is not hard to identify these points; they are dots adjacent to a name on the left that ends in a digit where there is a digit on the right of the dot. We do want to figure out how to return a meaningful error message, and that is more difficult. Just highlighting the dot might be enough, but we should consider the impact of highlighting other stuff around the dot. We could highlight the number around the dot or even include the name in full as well. What will give the most clarity to the end user? I think we want to highlight the number, since that is the main contention. Thus, when we handle masking off names, we must ensure that we are not removing the numeric information that we may want.

We have another consideration when we handle the masking off of names. A high minus is a legtimate terminator of a name, but a dot, as we have seen above, may not be. The dot may legitamitely represent a reference to Inner Product, or be ambiguous, or be numeric, as we are masking off names. After masking names, we can tell the class of the dots by examining what is on their left and right, as seen in Table 5. All of this gives us a suitable strategy for handling the final cleanup of `dm`. After masking off the names we can fix up dots in `dm` and handle ambiguous numeric errors.

To mask off the names, we must simply recognize what units of digits are right of a contiguous unit of alphabetics. These are digits that must belong to names. We must be sure when we do this that we are not breaking `dm` units somehow and failing to recognize syntax errors. If there are only digits, then dropping some units will cause no other change, but what about dropping in a group with non-digits? In the case of dots, we already know that we will need to fix these up. For a high minus, it will always appear at the beginning of any digit unit, and so you cannot mask off such a unit, and any units masked off past it are fine. For `e` and `j`, the masking off of such units can only mean that they are not numeric. Thus, we can be reasonably sure that this approach cannot mask off digits that it should not. It *will* mask off all the digits that should be masked off.

72a ⟨*Tokenize numbers* 64⟩+≡

```
(msk⌿dm)←∊∧⍀¨(msk←x∊alp,num)⊆dm
```

This code is used in chunk 21.
Uses `alp` 52b, `dm` 64, and `num` 52c.

We can handle the ambiguous errors and the dot cleanup in any order, but the handling of the dots first will make it more convenient to identify ambiguous cases. The dots that are still in `dm` but that are isolated and alone are not numeric, so we should remove them.

72b ⟨*Tokenize numbers* 64⟩+≡

```
dm[⍸dm∧(x='.')∧~(¯1⌽dm)∨1⌽dm]←0
```

This code is used in chunk 21.
Uses `dm` 64.

And with that we are free to handle the ambiguous parsing errors, which is anywhere a numeric dot is contiguous with a non-numeric digit on its left.

72c ⟨*Tokenize numbers* 64⟩+≡

```
∨⌿msk←∨⌿¨dm⊆dm∧(x='.')∧¯1⌽(~dm)∧x∊num:{
        EM←'AMBIGUOUS PLACEMENT OF NUMERIC FORM'
        2 EM SIGNAL ∊msk⌿dm⊆pos
}⍬
```

This code is used in chunk 21.
Uses `dm` 64, `num` 52c, and `SIGNAL` 20b.

With all that handled, the `dm` mask now contains a full and accurate set of numeric forms in the source. At this point we have not checked our numbers to ensure that they are actually representable in our runtime, but we do not care about that during tokenization. The numbers at this point are at least theoretically representable in some theoretical system.

We will make one concession in this case, which is to check to ensure our exponents are integers and not floats. This is a simple check that we can make right now and is different than the limits on range and representation. We mostly just need to examine all the exponent parts of our numbers and check for a decimal point. This requires that we break apart the numeric parts that we want from the groups in `dm`. However, when doing so, we want to keep a good link back to the source so that we can highlight the errors that we want. In this case, any floating exponents are an error in some real part of the code, so we will want to highlight the whole real part when we find an error. The easy way to do this is to have a mask `rm` of the real parts.

73a ⟨*Tokenize numbers* 64⟩+≡

```
∨⌿msk←∨⌿¨'.'={1⊃(⍵⊆⍨~⍵∊'Ee'),⊂''}¨x⊆⍨rm←dm∧~x∊'Jj':{
         EM←'NON-INTEGER EXPONENT'
         2 EM SIGNAL ∊msk⌿rm⊆pos
}⍬
```

This code is used in chunk 21.
Uses `dm` 64 and `SIGNAL` 20b.

That is all that we want to do at this point in tokenization. We are now free to use `dm` to handle the final tokenization of these values. Unlike with comments and strings, there is not as much value to removing dead nodes at this point since we still have a number of other items to tokenize that we may want to deal with using `dm`, and since most numbers are quite small and we are unlikely to gain much advantage by the reduction at the moment. Instead, we merely need to update `t` and `end` for the starting node in each `dm` group as appropriate, with `pos` of course already pointing at the correct position. For the `end` field we want to use the `end` value from the last character in the `dm` group. We will use type `N` for the type of a number token.

73b ⟨*Tokenize numbers* 64⟩+≡

```
t[i←⍸2<⌿0⍪dm]←N
end[i]←end⌿⍨2>⌿dm⍪0
```

This code is used in chunk 21.
Uses `dm` 64.

Now that the numeric tokens are there, what else remains with the parser? Eventually, we want these `N` nodes removed into their own `A` nodes, but we will handle that in our handling of arrays. Right now, the `N` nodes are not processed, meaning that their `n` field will contain strings instead of real numeric values. There is an argument to be made that we should keep them in character form because this will allow us to use more numeric forms that may not be supported by Dyalog APL. That would shunt off handling of the numeric values to the runtime. Such a decision would be short-sighted: if we have a self-hosting compiler, this will not matter, and not evaluating the numeric values in the parser would inhibit many potential compilation passes that we may want to add.

This means that we must evaluate the `n` field of the `N` nodes into real numeric values. To do this, we could attempt to do all of the parsing ourselves, but handling that is a remarkably subtle endeavour with many pitfalls. Instead, we will rely on the `⎕VFI` system function to do this work for us. We will signal a syntax error if we are unable to parse one of the `n` fields.

74 ⟨*Type-specific processing of the* `n` *field* 59⟩+≡

```
msk vals←⎕VFI ⍕n⍕(t=N)⌿n
~∧⌿msk:{
        EM←'CANNOT REPRESENT NUMBER'
        2 EM SIGNAL ∊((t=N)⍀~msk)⌿pos+⍳¨end-pos
}⍬
n[⍸t=N]←vals
```

This code is used in chunk 22.
Uses `SIGNAL` 20b.

After this, by the end of the parser, the `N` nodes should have been converted/merged into `A` nodes, so there is nothing else I can think of to handle numeric values in the parser.

What about in the compiler transformations? As with character vectors, we should not need to deal with them at all.

This takes us to the code generator. We are in a similar position for code generation as we are with character vectors, in that we can assume that by the time we are at the code generator, we have a simple `A` type that we expect to encapsulate most of the generic array generation code. Here, it is our responsibility to generate a `data[]` array with appropriate elements and type, and to connect that with the appropriate `enum array_type`.

Unlike the character element type, we have more than one numeric type that we must handle, and each one may require a little bit different handling, but especially types such as complex numbers. To make this more concrete, Table 2 shows the Dyalog numeric type and the associated C type and `enum array_type` that we will use. We also indicate the underlying ArrayFire element type. Notice that some types will have some conversion, while others will not.

We must make a generator case for each numeric type to encode that data. For the real numbers, we do not need to do any processing of the data because the numbers will be cast automatically and correctly in their formatted form directly from APL.

75 ⟨*Element* `data` *and* `type` *generator cases* 60⟩+≡
```
  5>i←11 83 163 323 645⍳⎕DR ω:{
          ⎕PP←17
          ct←i⊃(⊂'char'),(2⍴⊂'int16_t'),'int32_t' 'double'
          at←i⊃'BOOL' 'SINT' 'SINT' 'INT' 'DBL'
          z←⊂ct,' data[] = {',(⊃{α,',',ω}⌿⍕¨ω),'};'
          z,←⊂'enum array_type type = ARR_',at,';'
  z}ω
```
Uses `array_type` 85a and `PP` 132a.

The "ugly duckling" in the room is the complex number. This is because platform support for complex numbers varies in how it is handled. The main culprit is Microsoft Visual Studio.[7] Because MSVC uses structs to represent complex numbers instead of the C99 style built-ins, Dyalog APL uses a struct-based model that matches the MSVC model, rather than the C99 `double complex` form.[8] We will follow this model and define our own complex number struct.

76a ⟨*C runtime structures* 37a⟩+≡
```
struct apl_cmpx {
        double real;
        double imag;
};
```
This code is used in chunk 34a.
Defines:
`apl_cmpx`, used in chunk 76b.

This matches the format used by Dyalog's interpreter, allowing us to pull data straight out of a DWA array. We are also fortunate in that ArrayFire also makes use of the struct-based approach, allowing us to do simple initialization without any data conversion. This means that we can define `data[]` fairly normally except that we must initialize it as a struct and not as single values.

76b ⟨*Element* `data` *and* `type` *generator cases* 60⟩+≡
```
1289=⎕DR ⍵:{
        ⎕PP←17
        mk_struct←{'{',(9○⍵),',',(11○⍵),'}'}
        comma←{⍺,',',⍵}
        vals←⊃comma/mk_struct¨⍵
        z←⊂'struct apl_cmpx data[] = {',vals,'};'
        z,←⊂'enum array_type type = ARR_CMPX;'
z}⍬
```
Uses `apl_cmpx` 76a, `ARR_CMPX` 77a, `array_type` 85a, and `PP` 132a.

---

[7] docs.microsoft.com/en-us/cpp/c-runtime-library/complex-math-support

[8] Actually, I am rather okay with this, being something of a C89 traditionalist in aesthetic anyways.

The above ensures that the array code generator will have the appropriate data to work with. All that remains is to add support for numeric types into the runtime.

To support numbers in the runtime we must address the following:

- Add appropriate `enum array_type` values
- Map `enum array_type` values to ArrayFire representations
- Add support for numeric conversion to/from DWA to ArrayFire

We defined the following array types for numerics:

77a ⟨*Array element types* 61a⟩+≡

```
, ARR_BOOL, ARR_SINT, ARR_INT, ARR_DBL, ARR_CMPX
```

This code is used in chunk 85a.
Defines:
`ARR_BOOL`, used in chunks 77b, 81, 82, 89, and 92.
`ARR_CMPX`, used in chunks 76b, 77b, 81, and 82.
`ARR_DBL`, used in chunks 77b, 81, 82, 89, and 92.
`ARR_INT`, used in chunks 77b, 81, 82, 89, and 92.
`ARR_SINT`, used in chunks 77b, 81, 82, 89, and 92.

Each of these element types corresponds to the specific ArrayFire type indicated in Table 2.

77b ⟨*Cases for selecting device* `values` *dtype* 61b⟩+≡

```
case ARR_BOOL:
        dtype = b8;
        break;
case ARR_SINT:
        dtype = s16;
        break;
case ARR_INT:
        dtype = s32;
        break;
case ARR_DBL:
        dtype = f64;
        break;
case ARR_CMPX:
        dtype = c64;
        break;
```

Uses `ARR_BOOL` 77a, `ARR_CMPX` 77a, `ARR_DBL` 77a, `ARR_INT` 77a, and `ARR_SINT` 77a.

When we handle DWA array inputs that come from the interpreter, we want to handle the input data and possibly pre-process the data if we need to. We assume that `data` is a pointer to the DWA numeric data buffer and that `count` contains the element count of `data`. We must do any processing to the DWA data in the cases where the representation in the DWA buffer does not match the runtime representation. We assume that we are casing over the DWA element types of the `data` buffer.

The DWA element type consists of both simple and compound numeric types, some of which overlap with the `enum array_type` values.

78a ⟨*Simple DWA numeric element types* 78a⟩≡

```
, DWA_BOOL, DWA_TINT, DWA_SINT, DWA_INT, DWA_DBL
```

Root chunk (not used in this document).
Defines:
`DWA_BOOL`, used in chunks 79–81.
`DWA_DBL`, used in chunks 81 and 82.
`DWA_INT`, used in chunks 81 and 82.
`DWA_SINT`, used in chunks 81 and 82.
`DWA_TINT`, used in chunks 80–82.

78b ⟨*Compound DWA numeric element types* 78b⟩≡

```
, DWA_CMPX, DWA_R, DWA_F, DWA_Q
```

Root chunk (not used in this document).
Defines:
`DWA_CMPX`, used in chunks 81 and 82.
`DWA_F`, never used.
`DWA_Q`, never used.
`DWA_R`, never used.

In most cases, there is no pre-processing necessary, since the representation matches. However, in the cases of Boolean arrays and tiny integers, this is not the case.

With a Boolean array, the DWA representation uses a bitvector encoding in which the first element in a byte is the most significant, which I am calling big endian. We must convert this to the `b8` format used in ArrayFire, which is simply using a single byte per Boolean.

79 ⟨*Cases for pre-processing DWA* `data` *buffer* 79⟩≡

```
case DWA_BOOL:{
        char *buf = calloc(count, sizeof(char));

        if (buf == NULL) {
                err = 1;
                break;
        }

        for (size_t i = 0; i < count; i++) {
                char off = 7 - (i % 8);
                uint8_t bytes = data;
                buf[i] = 1 & (bytes[i / 8] >> off);
        }

        data = buf;
        break;
}
```

This definition is continued in chunk 80a.
Root chunk (not used in this document).
Uses `DWA_BOOL` 78a.

In the `DWA_TINT` case, we do not have an 8-bit signed integer representation that we can use because of a limitation in the underlying ArrayFire implementation. Instead, we must convert these values to 16-bit values.

80a ⟨*Cases for pre-processing DWA* `data` *buffer* 79⟩+≡
```
case DWA_TINT:{
        int16_t *buf = calloc(count, sizeof(int16_t));

        if (buf == NULL) {
                err = 1;
                break;
        }

        for (size_t i = 0; i < count; i++)
                buf[i] = ((int8_t *)data)[i];

        data = buf;
        break;
}
```
Uses `DWA_TINT` 78a.

Since we have allocated new memory for these data types, we must also remember to clean them up at the end, which we handle with these cleanup cases.

80b ⟨*Cases for cleaning up the DWA* `data` *buffer* 80b⟩≡
```
case DWA_BOOL:
case DWA_TINT:
        free(data);
        break;
```
Root chunk (not used in this document).
Uses `DWA_BOOL` 78a and `DWA_TINT` 78a.

We must also handle getting the right array type for a given DWA element types. We assume that when converting from a DWA value to a runtime array that we will want to calculate a `type` value from each DWA element type.

81 ⟨*Cases for selecting* `type` *based on DWA type* 62b⟩+≡
```
case DWA_BOOL:
        type = ARR_BOOL;
        break;
case DWA_TINT:
case DWA_SINT:
        type = ARR_SINT;
        break;
case DWA_INT:
        type = ARR_INT;
        break;
case DWA_DBL:
        type = ARR_DBL;
        break;
case DWA_CMPX:
        type = ARR_CMPX;
        break;
```
Uses `ARR_BOOL` 77a, `ARR_CMPX` 77a, `ARR_DBL` 77a, `ARR_INT` 77a, `ARR_SINT` 77a, `DWA_BOOL` 78a, `DWA_CMPX` 78b, `DWA_DBL` 78a, `DWA_INT` 78a, `DWA_SINT` 78a, and `DWA_TINT` 78a.

The previous cases will now allow us to go from a DWA value to a runtime value, but we must also go in the other direction. We must be able to take a runtime buffer and convert it into a DWA value. Fortunately, going in the opposite direction is much easier, because the runtime numeric types all have a bit-compatible analogue in the DWA numeric element types. This means that we can do a straight bulk copy into the DWA buffer assuming that we know the correct DWA type for each `enum array_type` without any preprocessing. We will assume a `switch` statement over the runtime array type wherein we set `type` to the appropriate dwa type.

82 ⟨*Cases for selecting* `type` *based on array type* 62c⟩+≡

```
case ARR_BOOL:
        type = DWA_TINT;
        break;
case ARR_SINT:
        type = DWA_SINT;
        break;
case ARR_INT:
        type = DWA_INT;
        break;
case ARR_DBL:
        type = DWA_DBL;
        break;
case ARR_CMPX:
        type = DWA_CMPX;
        break;
```

Uses `ARR_BOOL` 77a, `ARR_CMPX` 77a, `ARR_DBL` 77a, `ARR_INT` 77a, `ARR_SINT` 77a, `DWA_CMPX` 78b, `DWA_DBL` 78a, `DWA_INT` 78a, `DWA_SINT` 78a, and `DWA_TINT` 78a.

With the above in place, we now can handle data buffers in literal and DWA form and we can correctly store numeric data as the correct ArrayFire type.

Obviously, this is not a complete handling of array values. All we have done here is manage the numeric logic. The logic for array handling that is common among all elements and units, such as strings and variables, will be discussed in a separate section. For now, this completes the handling of numeric values across the parser, compiler, code generator, and runtime.

6.5 Variables

83a ⟨*Tokenize variables* 83a⟩≡
```
t[i←⍸2</0⍪vm←(~dm)∧x∊alp,num]←V ⋄ end[i]←end/⍨2>/vm⍪0

⍝ Tokenize ⍺, ⍵ formals
fm←{mm←⌽⊃(>∘⊃,⊢)/⌽m←⍺=' ',⍵ ⋄ 1↓¨(mm∧~m1)(mm∧m1←1⌽m)}
am aam←'⍺'fm x ⋄ wm wwm←'⍵'fm x
((am∨wm)/t)←A ⋄ ((aam∨wwm)/t)←P ⋄ ((aam∨wwm)/end)←end/⍨¯1⌽aam∨wwm
```
This code is used in chunk 21.
Uses `alp` 52b, `dm` 64, and `num` 52c.

83b ⟨*Check for out of context dfns formals* 83b⟩≡
```
∨/(d=0)∧(t=P)∧IN[pos]∊'⍺⍵':'DFN FORMAL REFERENCED OUTSIDE DFNS'⎕SIGNAL 2
```
This code is used in chunk 21.
Uses `SIGNAL` 20b.

83c ⟨*Convert* ⍺ *and* ⍵ *to* `V` *nodes* 83c⟩≡
```
t←V@(i←⍸(t=A)∧n∊,¨'⍺⍵')⊢t ⋄ vb[i]←i
```
This code is used in chunk 22.

83d ⟨*Convert* ⍺⍺ *and* ⍵⍵ *to* `P2` *nodes* 83d⟩≡
```
k[⍸(t=P)∧n∊'⍺⍺' '⍵⍵']←2
```
This code is used in chunk 22.

83e ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡
```
gck,←(V 0)(V 1)(V 2)(V 3)(V 4)
gcv,←'Va' 'Va' 'Vf' 'Vo' 'Vo'
```
This code is used in chunk 25b.

83f ⟨*Node-specific code generators* 24e⟩+≡
```
Va←{id←(|4⊃⍺)⊃'' 'r' 'l' 'aa' 'ww',5↓sym
          z ←⊂'*stkhd++ = retain_cell(',id,');'
  z}
```
This code is used in chunk 25b.
Uses `retain_cell` 41a.

6.6 Arrays

84a ⟨*Mark atoms, characters, and numbers as kind* `1` 84a⟩≡
```
k[⍸t∊A C N]←1
```
This code is used in chunk 22.

84b ⟨*Strand arrays into atoms* 84b⟩≡
```
i←|i⊣km←0<i←i[⍋|(i,⍨←-∪p[i]),p[i←⍸t[p]∊B Z]]
msk←(t[i]∊C N)∨msk∧⊃1 ¯1∨.⌽⊂msk←km∧(t[i]∊A C N V Z)∧k[i]=1
np←(≢p)+⍳≢ai←i⌿⍨am←2>⌿msk⍪0 ⋄ p←(np@ai⍳≢p)[p] ⋄ p,←ai ⋄ km←2<⌿0⍪msk
t k n pos end(⊣,I)←⊂ai ⋄ k[ai]←1 6[∨⌿¨msk⊆t[i]≠N]
t n pos(⊣@ai⍨)←A(⊂'')(pos[km⌿i]) ⋄ p[msk⌿i]←ai[(msk←msk∧~am)⌿¯1++⍀km]
i←⍸(t[p]=A)∧(k[p]=6)∧t=N
p,←i ⋄ t k n pos end(⊣,I)←⊂i ⋄ t k n(⊣@i⍨)←A 1(⊂'')
```
This code is used in chunk 22.

84c ⟨*Count strand and indexing children* 84c⟩≡
```
n[⍸(t∊A E)∧k=6]←0 ⋄ n[p⌿⍨(t[p]∊A E)∧k[p]=6]+←1
```
This code is used in chunk 23.

84d ⟨*Node ⟷ Generator mapping* 24c⟩+≡
```
gck,←(A 1)(A 6)
gcv,←'Aa' 'As'
```
This code is used in chunk 25b.

84e ⟨*Declare top-level array structures* 84e⟩≡
```
k[ω]=1:{
         z ←⊂'struct array *',n,';'
z}ω
```
This code is used in chunk 25a.

84f ⟨*Cell type names* 37c⟩+≡
```
, CELL_ARRAY
```
This code is used in chunk 37b.
Defines:
`CELL_ARRAY`, used in chunks 86 and 88b.

85a ⟨*C runtime enumerations* 37b⟩+≡
```
enum array_type {
        ARR_SPAN
        ⟨Array element types 61a⟩
        , ARR_MIXED, ARR_NESTED
};

enum array_storage {
        STG_HOST, STG_DEVICE
};
```
This code is used in chunk 34a.
Defines:
`array_storage`, used in chunks 85b and 86.
`array_type`, used in chunks 60, 75, 76b, 85b, 86, and 89.

85b ⟨*C runtime structures* 37a⟩+≡
```
struct cell_array {
        ⟨Common cell fields 36⟩
        enum array_storage storage;
        enum array_type type;
        void *values;
        unsigned int rank;
        unsigned long long shape[];
};
```
This code is used in chunk 34a.
Defines:
`cell_array`, used in chunks 86, 88a, 97a, and 109–111.
Uses `array_storage` 85a and `array_type` 85a.

86 ⟨*Array definitions* 86⟩≡
```
DECLSPEC int
mk_array(struct cell_array **dest,
    enum array_type type, enum array_storage storage,
    unsigned int rank, unsigned long long *shape, void *values)
{
        struct cell_array *arr;
        size_t  size;
        int     err;

        size = sizeof(struct cell_array) + rank * sizeof(unsigned long long);
        arr = malloc(size);

        if (arr == NULL)
                return 1;

        arr->ctyp       = CELL_ARRAY;
        arr->refc       = 1;
        arr->type       = type;
        arr->storage    = storage;
        arr->rank       = rank;
        arr->values     = NULL;

        size = 1;

        for (unsigned i = 0; i < rank; ++i) {
                arr->shape[i] = shape[i];
                size *= shape[i];
        }

        err = 0;

        switch (storage) {
        case STG_DEVICE:
                err = fill_device_array(arr, values, size, type);
                break;

        case STG_HOST:
                err = fill_host_array(arr, values, size, type);
                break;

        default:
                err = 16;
        }

        if (err) {
```

```
                free(arr);
                return err;
        }

        *dest = arr;

        return 0;
}



DECLSPEC void
release_array(struct cell_array *arr)
{
        if (arr == NULL)
                return;

        arr->refc--;

        if (arr->refc)
                return;

        if (arr->type == ARR_NESTED) {
                struct cell_array **values = arr->values;

                for (unsigned int i = 0; i < arr->rank; i++)
                        release_array(values[i]);
        }

        if (arr->values)
                switch (arr->storage) {
                case STG_HOST:
                        free(arr->values);
                        break;
                case STG_DEVICE:
                        af_release_array(arr->values);
                        break;
                default:
                        dwa_error(999);
                }

        free(arr);
}
```

This code is used in chunk 89.
Defines:
`mk_array`, used in chunks 88a and 92.

release_array, used in chunks 29 and 88.
Uses array_storage 85a, array_type 85a, CELL_ARRAY 84f, cell_array 85b, ctyp 36, DECLSPEC 35, dwa_error 45a, and refc 36.

88a ⟨*C runtime declarations* 39a⟩+≡
```
DECLSPEC int mk_array(struct cell_array **, ...);
DECLSPEC void release_array(struct cell_array *);
```
This code is used in chunk 34a.
Uses cell_array 85b, DECLSPEC 35, mk_array 86, and release_array 86.

88b ⟨*Cell release cases* 40c⟩+≡
```
case CELL_ARRAY:
        release_array(cell);
        break;
```
This code is used in chunk 40a.
Uses CELL_ARRAY 84f and release_array 86.

89 ⟨*array.c* 89⟩≡
```
#include <stddef.h>
#include <stdlib.h>
#include <arrayfire.h>

#include "codfns.h"

#if AF_API_VERSION < 38
#error "Your ArrayFire version is too old."
#endif

int
fill_device_array(struct array *arr, void *vals, size_t size, enum array_type typ
{
        af_dtype        aftyp;

        arr->values = NULL;

        switch (typ) {
        case ARR_BOOL:
                aftyp = b8;
                break;

        case ARR_SINT:
                aftyp = s16;
                break;

        case ARR_INT:
                aftyp = s32;
                break;

        case ARR_DBL:
                aftyp = f64;
                break;

        case ARR_CMP:
                aftyp = c64;
                break;

        case ARR_NESTED:
        case ARR_CHAR:
        case ARR_MIXED:
        default:
                return 16;
        }
```

```
        if (!size) {
                size = 1;

                return af_constant(&arr->values, 0, 1, &size, aftyp);
        }

        return af_create_array(&arr->values, vals, 1, &size, aftyp);
}

int
fill_host_array(struct array *arr, void *vals, size_t size, enum array_type typ
{
        struct array **data;
        struct pocket **pkts;
        int     err;

        if (typ != ARR_NESTED)
                return 16;

        arr->values = NULL;

        if (!size)
                size++;

        pkts = vals;
        data = calloc(size, sizeof(struct array *));

        if (data == NULL)
                return 1;

        for (size_t i = 0; i < size; i++) {
                err = dwa2array(&data[i], pkts[i]);

                if (err) {
                        free(data);
                        return err;
                }
        }

        arr->values = data;

        return 0;
}
```

⟨*Array definitions* 86⟩
Root chunk (not used in this document).

Defines:
`array.c`, used in chunk 91.
Uses `ARR_BOOL` 77a, `ARR_DBL` 77a, `ARR_INT` 77a, `ARR_SINT` 77a, `array_type` 85a, `codfns` 7, `codfns.h` 34a, and `dwa2array` 92.

91 ⟨*Tangle Commands* 8⟩+≡

```
echo "Tangling rtm/array.c..."
notangle -R'array.c' codfns.nw > rtm/array.c
```

This code is used in chunk 134.
Uses `array.c` 89 and `codfns` 7.

92 ⟨*DWA definitions* 44a⟩+≡
```
struct pocket *
getarray(enum dwa_type type, unsigned rank, long long *shape, struct localp *lp
{
        return (dwa->ws->getarr)(type, rank, shape, lp);
}

char *
cnvu8_ch(uint8_t *buf, size_t count)
{
        char *res;

        res = calloc(count, sizeof(char));

        if (res == NULL)
                return res;

        for (size_t i = 0; i < count; i++)
                res[i] = 1 & (buf[i/8] >> (7 - (i % 8)));

        return res;
}

int16_t *
cnvi8_i16(int8_t *buf, size_t count)
{
        int16_t *res;

        res = calloc(count, sizeof(int16_t));

        if (res == NULL)
                return res;

        for (size_t i = 0; i < count; i++)
                res[i] = buf[i];

        return res;
}

DECLSPEC int
dwa2array(struct array **tgt, struct pocket *pkt)
{
        struct  array *arr;
        long    long *shape;
        void    *data;
        size_t  count;
```

```
	int	err;
	unsigned	int rank;

	rank	= pkt->rank;
	shape	= pkt->shape;
	data	= DATA(pkt);

	switch (pkt->type) {
	case 15: /* Simple */
		switch (pkt->eltype) {
		case APLU8:
			count = 1;

			for (unsigned int i = 0; i < rank; i++)
				count *= shape[i];

			data = cnvu8_ch(data, count);

			if (data == NULL) {
				err = 1;
				goto done;
			}

		   err = mk_array(&arr, ARR_BOOL, STG_DEVICE, rank, shape, data)

			free(data);
			break;

		case APLTI:
			count = 1;

			for (unsigned int i = 0; i < rank; i++)
				count *= shape[i];

			data = cnvi8_i16(data, count);

			if (data == NULL) {
				err = 1;
				goto done;
			}

		   err = mk_array(&arr, ARR_SINT, STG_DEVICE, rank, shape, data)

			free(data);
			break;
```

```
                case APLSI:
                   err = mk_array(&arr, ARR_SINT, STG_DEVICE, rank, shape, data)
                        break;

                case APLI:
                    err = mk_array(&arr, ARR_INT, STG_DEVICE, rank, shape, data)
                        break;

                case APLD:
                    err = mk_array(&arr, ARR_DBL, STG_DEVICE, rank, shape, data)
                        break;

                case APLZ:
                    err = mk_array(&arr, ARR_CMP, STG_DEVICE, rank, shape, data)
                        break;

                default:
                        err = 16;
                }
                break;
        case 7: /* Nested */
                switch (pkt->eltype) {
                case APLP:
                   err = mk_array(&arr, ARR_NESTED, STG_HOST, rank, shape, data)
                        break;

                default:
                        err = 16;
                }
                break;

        default:
                err = 16;
        }

done:
        if (err)
                return err;

        *tgt = arr;

        return 0;
}

DECLSPEC int
array2dwa(struct pocket **dst, struct array *arr, struct localp *lp)
```

```
{
        struct  pocket *pkt;
        unsigned        int rank;
        long    long *shape;
        enum    dwa_type dtyp;
        size_t  count, esiz;
        int     err;

        if (arr == NULL) {
                if (lp)
                        lp->pocket = NULL;

                goto done;
        }

        rank = arr->rank;
        shape = arr->shape;

        if (rank > 15)
                return 16;

        switch (arr->type) {
        case ARR_BOOL:
                dtyp = APLTI;
                esiz = sizeof(int8_t);
                break;

        case ARR_SINT:
                dtyp = APLSI;
                esiz = sizeof(int16_t);
                break;

        case ARR_INT:
                dtyp = APLI;
                esiz = sizeof(int32_t);
                break;

        case ARR_DBL:
                dtyp = APLD;
                esiz = sizeof(double);
                break;

        case ARR_CMP:
                dtyp = APLZ;
                esiz = sizeof(dcomplex);
                break;
```

```
case ARR_NESTED:
        dtyp = APLP;
        esiz = sizeof(void *);
        break;

case ARR_MIXED:
case ARR_CHAR:
default:
        return 16;
}

pkt = getarray(dtyp, rank, shape, lp);

count = 1;
for (size_t i = 0; i < rank; i++)
        count *= shape[i];

switch (arr->storage) {
case STG_DEVICE:
        err = af_get_data_ptr(DATA(pkt), arr->values);

        if (err)
                return err;

        break;

case STG_HOST:
        memcpy(DATA(pkt), arr->values, esiz * count);
        break;

default:
        return 999;
}

if (arr->type == ARR_NESTED) {
        void **values = DATA(pkt);

        for (size_t i = 0; i < count; i++) {
           err = array2dwa(&(struct pocket *)values[i], values[i], NULL)

                if (err)
                        return err;
        }
}
```

```
done:
        if (dst)
                *dst = pkt;

        return 0;
}
```

This code is used in chunk 48a.
Defines:
`array2dwa`, used in chunks 29 and 97a.
`dwa2array`, used in chunks 29, 89, and 97a.
Uses `ARR_BOOL` 77a, `ARR_DBL` 77a, `ARR_INT` 77a, `ARR_SINT` 77a, `DATA` 97b, `dcomplex` 97b, `DECLSPEC` 35, `dwa_type` 98a, and `mk_array` 86.

97a ⟨*C runtime declarations* 39a⟩+≡

```
DECLSPEC int dwa2array(struct cell_array **, void *);
DECLSPEC int array2dwa(void **, struct cell_array *, void *);
```

This code is used in chunk 34a.
Uses `array2dwa` 92, `cell_array` 85b, `DECLSPEC` 35, and `dwa2array` 92.

97b ⟨*DWA macros* 97b⟩≡

```
#if defined(_WIN32)
#define dcomplex _Dcomplex
#else
#define dcomplex double complex
#endif

#define DATA(pp) ((void *)&(pp)->shape[(pp)->rank])
```

This code is used in chunk 48a.
Defines:
`DATA`, used in chunk 92.
`dcomplex`, used in chunk 92.

98a ⟨*DWA structures and enumerations* 43⟩+≡
```
enum dwa_type {
        APLNC=0, APLU8, APLTI, APLSI, APLI, APLD,
        APLP,    APLU,  APLV,  APLW,  APLZ, APLR, APLF, APLQ
};

struct pocket {
        long    long length;
        long    long refcount;
        unsigned        int type        : 4;
        unsigned        int rank        : 4;
        unsigned        int eltype      : 4;
        unsigned        int _0          : 13;
        unsigned        int _1          : 16;
        unsigned        int _2          : 16;
        long    long shape[1];
};
```
This code is used in chunk 48a.
Defines:
`dwa_type`, used in chunk 92.

6.7 Primitives

98b ⟨*Node ⟷ Generator mapping* 24c⟩+≡
```
gck,←(P 0)(P 1)(P 2)(P 3)(P 4)
gcv,←'Pv' 'Pv' 'Pf' 'Po' 'Po'
```
This code is used in chunk 25b.

98c ⟨*Node-specific code generators* 24e⟩+≡
```
Pf←{id←(symsⳁsym[|4⊃⍺])⊃nams
        z ←⊂'*stkhd++ = retain_cell(',id,');'
z}
```
This code is used in chunk 25b.
Uses `retain_cell` 41a.

6.7.1 APL Primitives

98d ⟨*Tokenize primitives and atoms* 98d⟩≡
```
t[⍸(~dm)∧x∊prms]←P ⋄ t[⍸x∊syna]←A
```
This code is used in chunk 21.
Uses `dm` 64, `prms` 54, and `syna` 53.

99a ⟨*Mark APL primitives with appropriate kinds* 99a⟩≡
```
k[⍸n∊,¨prmfs]←2 ⋄ k[⍸n∊,¨prmmo]←3 ⋄ k[⍸n∊,¨prmdo]←4
k[⍸n∊,¨prmfo]←5
k[i←⍸msk←(n∊⊂,'∘')∧1⌽n∊⊂,'.']←3 ⋄ end[i]←end[i+1] ⋄ n[i]←⊂,'∘.'
t k n pos end⌿⍨←⊂msk←~¯1⌽msk ⋄ p←(⍸~msk)(⊢-1+⍸)msk⌿p
```
This code is used in chunk 22.
Uses prmdo 54, prmfo 54, prmfs 54, and prmmo 54.

6.7.2 System Functions and Variables

99b ⟨*Tokenize system variables* 99b⟩≡
```
si←⍸('⎕'=IN[pos])∧1⌽t=V
t[si]←S ⋄ end[si]←end[si+1] ⋄ t[si+1]←0
```
This code is used in chunk 21.

99c ⟨*Verify that system variables are defined* 99c⟩≡
```
SYSV←,¨'Á' 'A' 'AI' 'AN' 'AV' 'AVU' 'BASE' 'CT' 'D' 'DCT' 'DIV' 'DM'
SYSV,←,¨'DMX' 'EXCEPTION' 'FAVAIL' 'FNAMES' 'FNUMS' 'FR' 'IO' 'LC' 'LX'
SYSV,←,¨'ML' 'NNAMES' 'NNUMS' 'NSI' 'NULL' 'PATH' 'PP' 'PW' 'RL' 'RSI'
SYSV,←,¨'RTL' 'SD' 'SE' 'SI' 'SM' 'STACK' 'TC' 'THIS' 'TID' 'TNAME' 'TNUMS'
SYSV,←,¨'TPOOL' 'TRACE' 'TRAP' 'TS' 'USING' 'WA' 'WSID' 'WX' 'XSI'
SYSF←,¨'ARBIN' 'ARBOUT' 'AT' 'C' 'CLASS' 'CLEAR' 'CMD' 'CONV' 'CR' 'CS' 'CSV'
SYSF,←,¨'CY' 'DF' 'DL' 'DQ' 'DR' 'DT' 'ED' 'EM' 'EN' 'EX' 'EXPORT'
SYSF,←,¨'FAPPEND' 'FCHK' 'FCOPY' 'FCREATE' 'FDROP' 'FERASE' 'FFT' 'IFFT'
SYSF,←,¨'FHIST' 'FHOLD' 'FIX' 'FLIB' 'FMT' 'FPROPS' 'FRDAC' 'FRDCI' 'FREAD'
SYSF,←,¨'FRENAME' 'FREPLACE' 'FRESIZE' 'FSIZE' 'FSTAC' 'FSTIE' 'FTIE'
SYSF,←,¨'FUNTIE' 'FX' 'INSTANCES' 'JSON' 'KL' 'LOAD' 'LOCK' 'MAP' 'MKDIR'
SYSF,←,¨'MONITOR' 'NA' 'NAPPEND' 'NC' 'NCOPY' 'NCREATE' 'NDELETE' 'NERASE'
SYSF,←,¨'NEW' 'NEXISTS' 'NGET' 'NINFO' 'NL' 'NLOCK' 'NMOVE' 'NPARTS'
SYSF,←,¨'NPUT' 'NQ' 'NR' 'NREAD' 'NRENAME' 'NREPLACE' 'NRESIZE' 'NS'
SYSF,←,¨'NSIZE' 'NTIE' 'NUNTIE' 'NXLATE' 'OFF' 'OR' 'PFKEY' 'PROFILE'
SYSF,←,¨'REFS' 'SAVE' 'SH' 'SHADOW' 'SIGNAL' 'SIZE' 'SR' 'SRC' 'STATE'
SYSF,←,¨'STOP' 'SVC' 'SVO' 'SVQ' 'SVR' 'SVS' 'TCNUMS' 'TGET' 'TKILL' 'TPUT'
SYSF,←,¨'TREQ' 'TSYNC' 'UCS' 'VR' 'VFI' 'WC' 'WG' 'WN' 'WS' 'XML' 'XT'
SYSD←,¨'OPT' 'R' 'S'
∨⌿msk←(t=S)∧~n∊'⎕',¨SYSV,SYSF,SYSD:{
        ERR←2'INVALID SYSTEM VARIABLE, FUNCTION, OR OPERATOR'
        ERR SIGNAL∊pos[⍵]{⍺+⍳⍵-⍺}¨end[⍵]
}⍸msk
```
This code is used in chunk 22.
Uses SIGNAL 20b.

99d ⟨*Mark system variables as* P *nodes with appropriate kinds* 99d⟩≡
```
k[⍸(t=S)∧n∊'⎕',¨SYSV]←1 ⋄ k[⍸(t=S)∧n∊'⎕',¨SYSF]←2 ⋄ k[⍸(t=S)∧n∊'⎕',¨SYSD]←4
t[⍸t=S]←P
```
This code is used in chunk 22.

6.8 Brackets

6.8.1 Indexing

100a ⟨*Convert* `;` *groups within brackets into* `Z` *nodes* 100a⟩≡
```
_←p[i]{k[z←⊃⍪⌿gz¨g←⍵⊂⍨¯1⌽IN[pos[⍵]]∊';]']←1 ⋄ t[z]←Z P[1=≢¨g]}⌸i←⍸t[p]=¯1
```
This code is used in chunk 22.

100b ⟨*Verify brackets have function/array target* 100b⟩≡
```
x←{⍵⌿⍨~∧⍀t[⍵]=¯1}∪⌽¨x
0∨.=≢¨x:'BRACKET SYNTAX REQUIRES FUNCTION OR ARRAY TO ITS LEFT'⎕SIGNAL 2
```
This code is used in chunk 102a.
Uses `SIGNAL` 20b.

100c ⟨*Enclose* `V[X;...]` *for expression parsing* 100c⟩≡
```
i←i[⍋p[i←⍸(t[p]∊B Z)∧(k[p]=1)∧p≠⍳≢p]] ⋄ j←i⌿⍨jm←t[i]=¯1
t[j]←A ⋄ k[j]←¯1 ⋄ p[i⌿⍨1⌽jm]←j
```
This code is used in chunk 22.

100d ⟨*Rationalize* `V[X;...]` 100d⟩≡
```
i←i[⍋p[i←⍸(t[p]=A)∧k[p]=¯1]] ⋄ msk←~2≠⌿¯1,ip←p[i] ⋄ ip←∪ip ⋄ nc←2×≢ip
t[ip]←E ⋄ k[ip]←2 ⋄ n[ip]←⊂'' ⋄ p[msk⌿i]←msk⌿(≢p)+1+2×¯1++⍀~msk
p,←2⌿ip ⋄ t,←nc⍴P E ⋄ k,←nc⍴2 6 ⋄ n,←nc⍴,¨'[' ''
pos,←2⌿pos[ip] ⋄ end,←∊(1+pos[ip]),⍪end[ip] ⋄ pos[ip]←pos[i⌿⍨~msk]
```
This code is used in chunk 22.

100e ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,';' ⋄ nams,←⊂'span'
```
This code is used in chunk 25b.

100f ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡
```
gck,←⊂E 6
gcv,←⊂'Ei'
```
This code is used in chunk 25b.

6.8.2 Axis Operator

101a ⟨*Rationalize* `F[X]` *syntax* 101a⟩≡

```
_←p[i]{
        ⊃m←t[⍵]=¯1:'SYNTAX ERROR:NOTHING TO INDEX'⎕SIGNAL 2
        k[⍵⌿⍨m∧¯1⌽(k[⍵]∊2 3 5)∨¯1⌽k[⍵]=4]←4
0}⌸i←⍸(t[p]∊B Z)∧(p≠⍳≢p)∧k[p]∊1 2
i←⍸(t=¯1)∧k=4 ⋄ j←⍸(t[p]=¯1)∧k[p]=4
(≢i)≠≢j:{
        2'AXIS REQUIRES SINGLE AXIS EXPRESSION'SIGNAL ∊pos[⍵]+⍳¨end[⍵]-pos[⍵]
}⊃⍪⌿{⊂⍺⌿⍨1<≢⍵}⌸p[j]
∨⌿msk←t[j]≠Z:{
        2'AXIS REQUIRES NON-EMPTY AXIS EXPRESSION'SIGNAL ∊pos[⍵]+⍳¨end[⍵]-pos[⍵
}msk⌿p[j]
p[j]←p[i] ⋄ t[i]←P ⋄ end[i]←1+pos[i]
```

This code is used in chunk 22.
Uses `SIGNAL` 20b.

6.9 Bindings and Types

101b ⟨*Parse Binding nodes* 101b⟩≡

```
⍝ Mark bindable nodes
bm←(t=V)∨(t=A)∧n∊,¨'⎕⍞'
bm←{bm⊣p[i]{bm[⍺]←(V ¯1≡t[⍵])∨∧⌿bm[⍵]}⌸i←⍸(~bm[p])∧t[p]=Z}⍣≡bm

⍝ Binding nodes
_←p[i]{
        t[⍵⌿⍨(n[⍵]∊⊂,'←')∧0,¯1↓bm[⍵]]←B
        b v←{(⊃¨x)(1↓¨x←⍵⌿⍨{t[⊃⍵]=B}¨⍵)}¯1⌽¨⍵⊂⍨1,¯1↓t[⍵]∊P B
        ∨⌿~bm[∊v]:'CANNOT BIND ASSIGNMENT VALUE'⎕SIGNAL 2
        p[⍵]←(⍺,b)[0,¯1↓+⍀t[⍵]=B]
        n[b]←n[∊v] ⋄ t[∊v]←¯7 ⋄ pos[b]←pos[∊v] ⋄ end[b]←end[⊃⌽⍵]
0}⌸i←⍸(t[p]=Z)∧p≠⍳≢p
t k n pos end⌿⍨←⊂msk←t≠¯7 ⋄ p←(⍸~msk)(⊢-1+⍸)msk⌿p
```

This code is used in chunk 22.
Uses `SIGNAL` 20b.

102a ⟨*Infer the type of bindings, groups, and variables* 102a⟩≡

```
  z x←↓⍉p[i]{αω}⌸i←⍸(t[p]∊B Z)∧p≠⍳≢p
  ⟨Verify brackets have function/array target 100b⟩
  _←{
          k[msk⌿z]←k[x⌿⍨msk←(k[⊃¨x]≠0)∧1=≢¨x]
          z x⌿⍨←⊂~msk

          k[z⌿⍨msk←k[⊃¨x]=4]←3
          z x⌿⍨←⊂~msk

          k[z⌿⍨msk←{(2 3 5∊⍨k[⊃ω])∨4=(ω,≢k)[0⍳⍨∧⍀k[ω]=1]⌷k,0}∘⌽¨x]←2
          z x⌿⍨←⊂~msk

          k[z⌿⍨msk←k[⊃∘⌽¨x]=1]←1
          z x⌿⍨←⊂~msk

          k[i]←k[vb[i←⍸t=V]]
  ≢z}⍣(=∨0=⊣)≢z
  'FAILED TO INFER ALL BINDING TYPES'assert 0=≢z:
```

This code is used in chunk 22.

102b ⟨*Parse dyadic operator bindings* 102b⟩≡

```
  ⍝ PARSE B←D...
  ⍝ PARSE B←...D
```

This code is used in chunk 22.

102c ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡

```
  gck,←(B 1)(B 2)(B 3)(B 4)
  gcv,←'Bv' 'Bf' 'Bo' 'Bo'
```

This code is used in chunk 25b.

102d ⟨*Node-specific code generators* 24e⟩+≡

```
  Bf←{id←sym⊃⍨|4⊃α
          z ←⊂id,' = retain_cell(stkhd[-1]);'
  z}
```

This code is used in chunk 25b.
Uses `retain_cell` 41a.

6.10 Assignments

103a ⟨*Parse assignments* 103a⟩≡
```
⍝ Wrap all assignment values as Z nodes
i km←⍪⌿p[i]{(α⍪ω)(0,1∨ω)}⌸i←⍸(t[p]∊B Z)∧(p≠⍳≢p)∧k[p]∊1
j←i⌿⍨msk←(t[i]=P)∧n[i]∊⊂,'←' ⋄ nz←(≢p)+⍳zc←+⌿msk
p,←nz ⋄ t k n,←zc⍴¨Z 1(⊂'') ⋄ pos,←1+pos[j] ⋄ end,←end[p[j]]
zm←¯1⌽msk ⋄ p[km⌿i]←(zpm⌿(i×~km)+zm⍀nz)[km⌿¯1++⍀zpm←zm∨~km]

⍝ This is the definition of a function value at this point
isfn←{(t[ω]∊O F)∨(t[ω]∊B P V Z)∧k[ω]=2}

⍝ Parse modified assignment to E4(V, F, Z)
j←i⌿⍨m←msk∧(¯1⌽isfn i)∧¯2⌽(t[i]=V)∧k[i]=1 ⋄ p[zi←nz⌿⍨msk⌿m]←j
p[i⌿⍨(1⌽m)∨2⌽m]←2⌿j ⋄ t k(⊣@j⍨)←E 4 ⋄ pos end n{α[ω]@j⊢α}←vi zi,⊂vi←i⌿⍨2⌽m

⍝ Parse bracket modified assignment to E4(E6, O2(F, P3(←)), Z)
j←i⌿⍨m←msk∧(¯1⌽isfn i)∧(¯2⌽t[i]=¯1)∧¯3⌽(t[i]=V)∧k[i]=1
p[zi←nz⌿⍨msk⌿m]←ei←i⌿⍨3⌽m ⋄ t k end(⊣@ei⍨)←E 4(end[zi])
p t k n(⊣@(i⌿⍨2⌽m)⍨)←ei E 6(⊂'')
p,←j ⋄ t,←Pρ⍨≢j ⋄ k,←3ρ⍨≢j ⋄ n,←(≢j)ρ⊂,'←' ⋄ pos,←pos[j] ⋄ end,←end[j]
p t k n pos(⊣@j⍨)←ei O 2(⊂'')(pos[fi←i⌿⍨1⌽m]) ⋄ p[fi]←j

⍝ Parse bracket assignment to E4(E6, P2(←), Z)
j←i⌿⍨m←msk∧(¯1⌽t[i]=¯1)∧¯2⌽(t[i]=V)∧k[i]=1 ⋄ p[zi←nz⌿⍨msk⌿m]←ei←i⌿⍨2⌽m
t k end(⊣@ei⍨)←E 4(end[zi]) ⋄ p t k n(⊣@(i⌿⍨1⌽m)⍨)←ei E 6(⊂'')
p t k(⊣@j⍨)←ei P 2

⍝ Parse modified strand assignment
⍝ Parse strand assignment

⍝ SELECTIVE MODIFIED ASSIGNMENT
⍝ SELECTIVE ASSIGNMENT
```
This code is used in chunk 22.

103b ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'←' ⋄ nams,←⊂'get'
```
This code is used in chunk 25b.

103c ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡
```
gck,←⊂E 4
gcv,←⊂'Eb'
```
This code is used in chunk 25b.

104a ⟨*Cell type names* 37c⟩+≡
```
, CELL_BOX
```
This code is used in chunk 37b.
Defines:
`CELL_BOX`, used in chunk 105.

104b ⟨*C runtime structures* 37a⟩+≡
```
struct cell_box {
        ⟨Common cell fields 36⟩
        void *value;
};
```
This code is used in chunk 34a.
Defines:
`box`, used in chunks 105a, 106b, and 131.

105a ⟨*Box definitions* 105a⟩≡
```
DECLSPEC int
mk_box(struct cell_box **box, void *value)
{
        *box = malloc(sizeof(struct cell_box));

        if (*box == NULL)
                return 1;

        (*box)->ctyp    = CELL_BOX;
        (*box)->refc    = 1;
        (*box)->value   = value;

        return 0;
}

DECLSPEC void
release_box(struct cell_box *box)
{
        if (box == NULL)
                return;

        box->refc--;

        if (box->refc)
                return;

        release_cell(box->value);
        free(box);
}
```
This code is used in chunk 106a.
Defines:
`mk_box`, used in chunk 105b.
`release_box`, used in chunk 105.
Uses `box` 104b, `CELL_BOX` 104a, `ctyp` 36, `DECLSPEC` 35, `refc` 36, and `release_cell` 40a.

105b ⟨*C runtime declarations* 39a⟩+≡
```
DECLSPEC int mk_box(struct cell_box **, void *);
DECLSPEC void release_box(struct cell_box *);
```
This code is used in chunk 34a.
Uses `DECLSPEC` 35, `mk_box` 105a, and `release_box` 105a.

105c ⟨*Cell release cases* 40c⟩+≡
```
case CELL_BOX:
        release_box(cell);
        break;
```
This code is used in chunk 40a.
Uses `CELL_BOX` 104a and `release_box` 105a.

106a ⟨*box.c* 106a⟩≡
```
#include <stdlib.h>

#include "codfns.h"

⟨Box definitions 105a⟩
```
Root chunk (not used in this document).
Defines:
`box.c`, used in chunk 106b.
Uses `codfns` 7 and `codfns.h` 34a.

106b ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling rtm/box.c..."
notangle -R'box.c' codfns.nw > rtm/box.c
```
This code is used in chunk 134.
Uses `box` 104b, `box.c` 106a, and `codfns` 7.

6.11 Expressions

106c ⟨*Parse brackets and parentheses into* `¯1` *and* `Z` *nodes* 106c⟩≡
```
_←p[i]{
        x←IN[pos[ω]]
        bd←+⍀bm←(bo←'['=x)+-bc←']'=x
        pd←+⍀pm←(po←'('=x)+-pc←')'=x
        0≠⊃⌽bd:{
                ix←pos[ω]{x+⍳(⌈⌿ω)-x←⌊⌿α}⍤{ω⌿⍨0≠bd}end[ω]
                2'UNBALANCED BRACKETS'SIGNAL ix
        }ω
        0≠⊃⌽pd:{
                ix←pos[ω]{x+⍳(⌈⌿ω)-x←⌊⌿α}⍤{ω⌿⍨0≠pd}end[ω]
                2'UNBALANCED PARENTHESES'SIGNAL ix
        }ω
        (po⌿bd)∨.≠⌽pc⌿bd:{
                'OVERLAPPING BRACKETS AND PARENTHESES'⎕SIGNAL 2
        }ω
        p[ω]←(α,ω)[1+¯1@{ω=⍳≢ω}D2P +⍀¯1⌽bm+pm]
        t[bo⌿ω]←¯1 ⋄ t[po⌿ω]←Z
        end[po⌿ω]←end[⌽pc⌿ω] ⋄ end[bo⌿ω]←end[⌽bc⌿ω]
0}⌸i←⍸(t[p]=Z)∧p≠⍳≢p
t k n pos end⌿⍨←⊂msk←~IN[pos]∊')' ⋄ p←(⍸~msk)(⊢-1+⍸)msk⌿p
```
This code is used in chunk 22.
Uses `SIGNAL` 20b.

106d ⟨*Group function and value expressions* 106d⟩≡
```
i km←⍪⌿p[i]{(α⍪ω)(0,1∨ω)}⌸i←⍸(t[p]∊B Z)∧(p≠⍳≢p)∧k[p]∊1 2
```
This code is used in chunk 22.

107a ⟨*Lift and flatten expressions* 107a⟩≡

```
p[i]←p[x←p I@{~t[p[ω]]∊F G}⍣≡i←⍸t∊G A B C E O P V] ⋄ j←(⌽i)[⍋⌽x]
p t k n r{α[ω]@i⊢α}←⊂j ⋄ p←(i@j⊢⍳≢p)[p]
```

This code is used in chunk 23.

6.11.1 Value Expressions

107b ⟨*Parse value expressions* 107b⟩≡

```
i km←⍪⌿p[i]{(α⍪ω)(0,(2≤≢ω)∧1∨ω)}⌸i←⍸(t[p]∊B Z)∧(k[p]=1)∧p≠⍳≢p
msk←m2∨fm∧~¯1⌽m2←km∧(1⌽km)∧~fm←(t[i]=O)∨(t[i]≠A)∧k[i]=2
t,←E⍴⍨xc←+⌿msk ⋄ k,←msk⌿msk+m2 ⋄ n,←xc⍴⊂''
pos,←pos[msk⌿i] ⋄ end,←end[p[msk⌿i]]
p,←msk⌿¯1⌽(i×~km)+km×x←¯1+(≢p)++⍀msk ⋄ p[km⌿i]←km⌿x
```

This code is used in chunk 22.

107c ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡

```
gck,←(E 1)(E 2)
gcv,←'Em' 'Ed'
```

This code is used in chunk 25b.

107d ⟨*Node-specific code generators* 24e⟩+≡

```
Em←{
          z ←⊂'c = *--stkhd;'
          z,←⊂'w = *--stkhd;'
          z,←⊂'(c->fn)((struct array **)stkhd++, NULL, w, c->fv);'
          z,←⊂'release_cell(c);'
          z,←⊂'release_cell(w);'
z}
```

This code is used in chunk 25b.

6.11.2 Function Expressions

108a ⟨*Parse function expressions* 108a⟩≡
```
⍝ Mask and verify dyadic operator right operands
(dm←¯1⌽(k[i]=4)∧t[i]∊F P V Z)∨.∧(~km)∨k[i]∊0 3 4:{
        'MISSING RIGHT OPERAND'⎕SIGNAL 2
}⍬

⍝ Refine schizophrenic types
k[i⌿⍨(k[i]=5)∧dm∨¯1⌽(~km)∨(~dm)∧k[i]∊1 6]←2 ⋄ k[i⌿⍨k[i]=5]←3

⍝ Rationalize ∘.
jm←(t[i]=P)∧n[i]∊⊂,'∘.'
jm∨.∧1⌽(~km)∨k[i]∊3 4:'MISSING OPERAND TO ∘.'⎕SIGNAL 2
p←((ji←jm⌿i)@(jj←i⌿⍨¯1⌽jm)⍳≢p)[p] ⋄ t[ji,jj]←t[jj,ji] ⋄ k[ji,jj]←k[jj,ji]
n[ji,jj]←n[jj,ji] ⋄ pos[ji,jj]←pos[ji,ji] ⋄ end[ji,jj]←end[jj,jj]

⍝ Mask and verify monadic and dyadic operator left operands
∨⌿msk←(dm∧¯2⌽~km)∨(¯1⌽~km)∧mm←(k[i]=3)∧t[i]∊F P V Z:{
        2'MISSING LEFT OPERAND'SIGNAL ∊pos[⍵]+⍳¨end[⍵]-pos[⍵]
}i⌿⍨msk
msk←dm∨mm

⍝ Parse function expressions
np←(≢p)+⍳xc←≢oi←msk⌿i ⋄ p←(np@oi⍳≢p)[p] ⋄ p,←oi ⋄ t k n pos end(⊣,I)←⊂oi
p[g⌿i]←oi[(g←(~msk)∧(1⌽msk)∨2⌽dm)⌿xc-⌽+⍀⌽msk]
p[g⌿oi]←(g←msk⌿(1⌽mm)∨2⌽dm)⌿1⌽oi ⋄ t[oi]←O ⋄ n[oi]←⊂''
pos[oi]←pos[g⌿i][msk⌿¯1++⍀g←(~msk)∧(1⌽mm)∨2⌽dm]
ol←1+(k[i⌿⍨(2⌽mm)∨3⌽dm]=4)∨k[i⌿⍨(1⌽mm)∨2⌽dm]∊2 3
or←(msk⌿dm)⍀1+k[dm⌿i]=2
k[oi]←3 3⊥↑or ol
```
This code is used in chunk 22.
Uses `dm` 64 and `SIGNAL` 20b.

108b ⟨*Node ⟷ Generator mapping* 24c⟩+≡
```
gck,←(0 1)(0 2)(0 4) (0 5) (0 7) (0 8)
gcv,←'Ov' 'Of' 'Ovv' 'Ofv' 'Ovf' 'Off'
```
This code is used in chunk 25b.

6.12 Trains

108c ⟨*Parse trains* 108c⟩≡
```
⍝ TRAINS
```
This code is used in chunk 22.

6.13 Functions

109a ⟨*Declare top-level function bindings* 109a⟩≡
```
k[ω]∊0 2:{
         z ←⊂'int'
        z,←⊂n,'(struct array **z, struct array *l, struct array *r, void *fv[]);'
         z,←⊂''
z}ω
```
This code is used in chunk 24e.

109b ⟨*Declare top-level closures* 109b⟩≡
```
k[ω]=2:{
         z ←⊂'struct closure *',n,';'
         z,←⊂''
         ⟨DWA Function Export 29⟩
z}ω
```
This code is used in chunk 25a.

109c ⟨*Cell type names* 37c⟩+≡
```
, CELL_CLOSURE
```
This code is used in chunk 37b.
Defines:
`CELL_CLOSURE`, used in chunks 110 and 111b.

109d ⟨*C runtime structures* 37a⟩+≡
```
struct cell_closure {
         ⟨Common cell fields 36⟩
         int (*fn)(struct cell_array **,
             struct cell_array *, struct cell_array *, void **);
         unsigned int fs;
         void *fv[];
}
```
This code is used in chunk 34a.
Defines:
`cell_closure`, used in chunks 110 and 111.
Uses `cell_array` 85b.

110 ⟨*Closure definitions* 110⟩≡
```
DECLSPEC int
mk_closure(struct cell_closure **k,
    int (*fn)(struct cell_array **,
        struct cell_array *, struct cell_array *, void **),
    unsigned int fs)
{
        size_t sz;
        struct cell_closure *ptr;

        sz = sizeof(struct cell_closure) + fs * sizeof(void *);
        ptr = malloc(sz);

        if (ptr == NULL)
                return 1;

        ptr->ctyp = CELL_CLOSURE;
        ptr->refc = 1;
        ptr->fn = fn;
        ptr->fs = fs;

        *k = ptr;

        return 0;
}

DECLSPEC void
release_closure(struct cell_closure *k)
{
        if (k == NULL)
                return;

        k->refc--;

        if (k->refc)
                return;

        for (unsigned int i = 0; i < k->fs; i++)
                release_cell(k->fv[i]);

        free(k);
}
```
This definition is continued in chunk 111c.
This code is used in chunk 112a.
Defines:
`mk_closure`, used in chunk 111.

release_closure, used in chunk 111.
Uses cell_array 85b, CELL_CLOSURE 109c, cell_closure 109d, ctyp 36, DECLSPEC 35, refc 36, and release_cell 40a.

111a ⟨*C runtime declarations* 39a⟩+≡
```
DECLSPEC int mk_closure(struct cell_closure **,
    int (*)(struct cell_array **,
        struct cell_array *, struct cell_array *, void **),
    unsigned int);
DECLSPEC void release_closure(struct cell_closure *);
```
This code is used in chunk 34a.
Uses cell_array 85b, cell_closure 109d, DECLSPEC 35, mk_closure 110, and release_closure 110.

111b ⟨*Cell release cases* 40c⟩+≡
```
case CELL_CLOSURE:
        release_closure(cell);
        break;
```
This code is used in chunk 40a.
Uses CELL_CLOSURE 109c and release_closure 110.

111c ⟨*Closure definitions* 110⟩+≡
```
DECLSPEC int
apply_dop(struct cell_closure **z,
    struct cell_closure *op, void *l, void *r)
{
        int err;

        err = mk_closure(z, op->fn, op->fs+2);

        if (err)
                return err;

        (*z)->fv[0] = l;
        (*z)->fv[1] = r;

        memcpy(&(*z)->fv[2], op->fv, op->fs * sizeof(op->fv[0]));

        for (unsigned int i = 0; i < (*z)->fs; i++)
                retain_cell((*z)->fv[i]);

        return 0;
}
```
This code is used in chunk 112a.
Defines:
apply_dop, never used.
apply_mop, never used.
Uses cell_closure 109d, DECLSPEC 35, mk_closure 110, and retain_cell 41a.

112a ⟨*closure.c* 112a⟩≡
```
#include <stdlib.h>
#include <string.h>

#include "codfns.h"

⟨Closure definitions 110⟩
```
Root chunk (not used in this document).
Defines:
`closure.c`, used in chunk 112b.
Uses `codfns` 7 and `codfns.h` 34a.

112b ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling rtm/closure.c..."
notangle -R'closure.c' codfns.nw > rtm/closure.c
```
This code is used in chunk 134.
Uses `closure.c` 112a and `codfns` 7.

112c ⟨*C runtime declarations* 39a⟩+≡
This code is used in chunk 34a.

6.13.1 D-fns

112d ⟨*Compute dfns regions and type, with* } *as a child* 112d⟩≡
```
t[⍸'{'=x]←F ⋄ 0≠⊃d←¯1⌽+⍀1 ¯1 0['{}'⍳x]:'UNBALANCED DFNS'⎕SIGNAL 2
```
This code is used in chunk 21.
Uses `SIGNAL` 20b.

112e ⟨*Compute the nameclass of dfns* 112e⟩≡
```
k←2×t∊F ⋄ k[∪p⌿⍨(t=P)∧n∊⊂'⍺⍺']←3 ⋄ k[∪p⌿⍨(t=P)∧n∊⊂'⍵⍵']←4
```
This code is used in chunk 22.

112f ⟨*Wrap all dfns expression bodies as* `Z` *nodes* 112f⟩≡
```
_←p[i]{end[⍺]←end[⊃⌽⍵] ⋄ gz¨⍵⊂⍨1,¯1↓t[⍵]=Z}⌸i←⍸t[p]=F
'Non-Z dfns body node'assert t[⍸t[p]=F]=Z:
```
This code is used in chunk 22.

112g ⟨*Anchor variables to earliest binding in the matching frame* 112g⟩≡
```
rf←¯1@{~t[⍵]∊F G M}p[rz←I@{~(t[⍵]=Z)∧(t[p[⍵]]∊F G M)∨p[⍵]=⍵}⍣≡⍨p]
rf[i]←p[i←⍸t=G] ⋄ rz[i]←i ⋄ rf←rf I@{rz∊p[i]⊢∘⊃⌸i←⍸t[p]=G}rf
mk←{⍺[⍵],⍪n[⍵]}
fr←rf mk⊢fb←fb[⍳⍨rf mk⊢fb←fb I∘(⍳⍨)∪⊖rz mk⊢fb←⍸t=B] ⋄ fb,←¯1
vb←fb[fr⍳rf mk i]@(i←⍸t=V)⊢¯1⍴⍨≢p
vb[i⌿⍨(rz[i]<rz[b])∨(rz[i]=rz[b])∧i≥b←vb[i←i⌿⍨vb[i]≠¯1]]←¯1
_←{z/⍨¯1=vb[1⌷z]←fb[fr⍳⍉n I@1⊢z←rf I@0⊢⍵]}⍣≡⍉{rf[⍵],⍪⍵}⍸(t=V)∧vb=¯1
∨⌿msk←(t=V)∧vb=¯1:{
        6'ALL VARIABLES MUST REFERENCE A BINDING'SIGNAL∊pos[⍵]{⍺+⍳⍵-⍺}¨end[⍵]
}⍸msk
```
This code is used in chunk 22.

113a ⟨*Lift dfns to the top-level* 113a⟩≡
```
p,←n[i]←(≢p)+⍳≢i←⍸(t=F)∧p≠⍳≢p ⋄ t k n r(⊣,I)←⊂i ⋄ p r I⍨←⊂n[i]@i⊢⍳≢p
t[i]←C
```
This code is used in chunk 23.

113b ⟨*Wrap expressions as binding or return statements* 113b⟩≡
```
i←(⍸(~t∊F G)∧t[p]=F),{⍵⌿⍨2|⍳≢⍵}⍸t[p]=G ⋄ p t k n r⌿⍨←⊂m←2@i⊢1⍴⍨≢p
p r i I⍨←⊂j←(+⍀m)-1 ⋄ n←j I@(0≤⊢)n ⋄ p[i]←j←i-1
k[j]←-(k[r[j]]=0)∨0@({⊃⌽⍵}⌸p[j])⊢(t[j]=B)∨(t[j]=E)∧k[j]=4 ⋄ t[j]←E
```
This code is used in chunk 23.

113c ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡
```
gck,←(E ¯1)(E 0)
gcv,←'Ek'  'Er'
```
This code is used in chunk 25b.

113d ⟨*Compute slots and frames* 113d⟩≡
```
⍝ Compute slots for each frame
s←¯1,⍨∊⍳¨n[∪x]←⊢∘≢⌸x←0⌷⍉e←∪I∘⍋⍨rn←r[b],⍪n[b←⍸t=B]

⍝ Compute frame depths
d←(≢p)↑d ⋄ d[i←⍸t=F]←0 ⋄ _←{z⊣d[i]+←⍵≠z←r[⍵]}⍣≡i ⋄ f←d[0⌷⍉e],¯1
```
This code is used in chunk 23.

113e ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'∇' ⋄ nams,←⊂'this'
```
This code is used in chunk 25b.

113f ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡
```
gck,←(C 1)(C 2)(F 2)(F 3)(F 4)
gcv,←'Ca' 'Cf' 'Fn' 'Fm' 'Fd'
```
This code is used in chunk 25b.

113g ⟨*Node-specific code generators* 24e⟩+≡
```
Cf←{id←⍕4⊃⍺
        z ←⊂'mk_closure((struct closure **)stkhd++, fn',id,', 0);'
z}
```
This code is used in chunk 25b.

113h ⟨*Node-specific code generators* 24e⟩+≡
```
Ek←{
        z ←⊂'release_cell(*--stkhd);'
        z,←⊂''
z}
```
This code is used in chunk 25b.

114a ⟨*Node-specific code generators* 24e⟩+≡
```
Er←{
        z ←⊂'*z = *--stkhd;'
        z,←⊂'goto cleanup;'
        z,←⊂''
  z}
```
This code is used in chunk 25b.

114b ⟨*Node-specific code generators* 24e⟩+≡
```
Fn←{id←⍕5⊃⍺ ⋄ x←⍉⊃⍪⌿⍵ ⋄ t←2⌷x ⋄ k←3⌷x
        hsw←(t=O)∨(t=E)∧k∊1 2 ⋄ hsa←((t=E)∧k=2)∨(t=O)∧k∊4 5 7 8
        z ←⊂'int'
        z,←⊂'fn',id,'(struct array **z, '
        z,←⊂'    struct array *l, struct array *r, void *fv[])'
        z,←⊂'{'
        z,←⊂'       void    *stk[128];'
        z,←⊂'       void    **stkhd;'
        z,←hsw⌿⊂' void    *w;'
        z,←hsa⌿⊂' void    *a;'
        z,←hsw⌿⊂' struct  closure *c;'
        z,←⊂''
        z,←⊂'       stkhd = &stk[0];'
        z,←⊂''
        z,← ' ',¨⊃,⌿dis¨⍵
        z,←⊂'       *z = NULL;'
        z,←⊂''
        z,←⊂'cleanup:'
        z,←⊂'       return 0;'
        z,←⊂'}'
        z,←⊂''
  z}
```
This code is used in chunk 25b.

6.13.2 Trad-fns

114c ⟨*Compute trad-fns regions* 114c⟩≡
```
∨⌿Z≠t⌿⍨1⌽msk←(d=0)∧'∇'=x:'TRAD-FNS START/END LINES MUST BEGIN WITH ∇'⎕SIGNAL 2
0≠⊃tm←¯1⌽≠⍀(d=0)∧'∇'=x:'UNBALANCED TRAD-FNS'⎕SIGNAL 2
∨⌿Z≠t⌿⍨⊃1 ¯1∨.⌽⊂(2>⌿tm)⍪0:'TRAD-FNS END LINE MUST CONTAIN ∇ ALONE'⎕SIGNAL 2
```
This code is used in chunk 21.
Uses SIGNAL 20b.

6.14 Guards

115a ⟨*Parse guards to* `(G (Z ...) (Z ...))` 115a⟩≡
```
  _←p[i]{
          0=+⌿m←':'=IN[pos[⍵]]:⍬
          ⊃m:'EMPTY GUARD TEST EXPRESSION'⎕SIGNAL 2
          1<+⌿m:'TOO MANY GUARDS'⎕SIGNAL 2
          t[⍺]←G ⋄ p[ti←gz⊃tx cq←2↑(⊂⍬)⍪⍨⍵⊂⍨1,¯1↓m]←⍺ ⋄ k[ti]←1
          ci←≢p ⋄ p,←⍺ ⋄ t k pos end⍪←0 ⋄ n,←⊂'' ⋄ k[gz cq,ci]←1
  0}⌸i←⍸t[p[p]]=F
```
This code is used in chunk 22.
Uses `SIGNAL` 20b and `TEST` 16a.

115b ⟨*Lift guard tests* 115b⟩≡
```
  p[i]←p[x←¯1+i←{⍵⌿⍨~2|⍳≢⍵}⍸t[p]=G] ⋄ t[i,x]←t[x,i] ⋄ k[i,x]←k[x,i]
  n[x]←n[i] ⋄ p←((x,i)@(i,x)⊢⍳≢p)[p]
```
This code is used in chunk 23.

115c ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡
```
  gck,←⊂G 0
  gcv,←⊂'Gd'
```
This code is used in chunk 25b.

6.14.1 Error Guards

6.15 Labels

115d ⟨*Identify label colons vs. others* 115d⟩≡
```
  t[⍸tm∧(d=0)∧∊((~⊃)∧(<⍀∨⍀))¨':'=(t=Z)⊂IN[pos]]←L
```
This code is used in chunk 21.

115e ⟨*Tokenize labels* 115e⟩≡
```
  ERR←'LABEL MUST CONSIST OF A SINGLE NAME'
  ∨⌿(Z≠t[li-1])∨(V≠t[li←⍸1⌽msk←t=L]):ERR ⎕SIGNAL 2
  t[li]←L ⋄ end[li]←end[li+1]
  d tm t pos end(⌿⍨)←⊂~msk
```
This code is used in chunk 21.
Uses `SIGNAL` 20b.

115f ⟨*Parse labels* 115f⟩≡
```
  ⍝ XXX: Parse labels
```
Root chunk (not used in this document).

6.16 Statements

6.16.1 What is a keyword?

116a ⟨*Tokenize keywords* 116a⟩≡

```
ki←⍸(t=0)∧(d=0)∧(':'=IN[pos])∧1⌽t=V
t[ki]←K ⋄ end[ki]←end[ki+1] ⋄ t[ki+1]←0
ERR←'EMPTY COLON IN NON-DFNS CONTEXT, EXPECTED LABEL OR KEYWORD'
∨⌿(t=0)∧(d=0)∧':'=IN[pos]:ERR ⎕SIGNAL 2
```

This code is used in chunk 21.
Uses `SIGNAL` 20b.

116b ⟨*Check that all keywords are valid* 116b⟩≡

```
KW←'NAMESPACE' 'ENDNAMESPACE' 'END' 'IF' 'ELSEIF' 'ANDIF' 'ORIF' 'ENDIF'
KW,←'WHILE' 'ENDWHILE' 'UNTIL' 'REPEAT' 'ENDREPEAT' 'LEAVE' 'FOR' 'ENDFOR'
KW,←'IN' 'INEACH' 'SELECT' 'ENDSELECT' 'CASE' 'CASELIST' 'ELSE' 'WITH'
KW,←'ENDWITH' 'HOLD' 'ENDHOLD' 'TRAP' 'ENDTRAP' 'GOTO' 'RETURN' 'CONTINUE'
KW,←'SECTION' 'ENDSECTION' 'DISPOSABLE' 'ENDDISPOSABLE'
KW,¨⍨←':'
msk←~KW∊⍨kws←n⌿⍨km←t=K
∨⌿msk:('UNRECOGNIZED KEYWORD ',kws⊃⍨⊃⍸msk)⎕SIGNAL 2
```

This code is used in chunk 22.
Uses `SIGNAL` 20b.

6.16.2 Namespaces

116c ⟨*Check that namespaces are at the top level* 116c⟩≡

```
msk←kws∊':NAMESPACE' ':ENDNAMESPACE'
∨⌿msk∧km⌿tm:'NAMESPACE SCRIPTS MUST APPEAR AT THE TOP LEVEL'⎕SIGNAL 2
```

This code is used in chunk 22.
Uses `SIGNAL` 20b.

116d ⟨*Nest top-level root lines as `Z` nodes* 116d⟩≡

```
_←(gz 1⌽⊢)¨(t[i]=Z)⊂i←⍸d=0
'Non-Z top-level node'assert t[⍸p=⍳≢p]=Z:
```

This code is used in chunk 22.

117a ⟨*Parse* `:Namespace` *syntax* 117a⟩≡
```
nss←n∊⊂':NAMESPACE' ⋄ nse←n∊⊂':ENDNAMESPACE'
ERR←':NAMESPACE KEYWORD MAY ONLY APPEAR AT BEGINNING OF A LINE'
Z∨.≠t⌿⍨1⌽nss:ERR ⎕SIGNAL 2
ERR←'NAMESPACE DECLARATION MAY HAVE ONLY A NAME OR BE EMPTY'
∨⌿(Z≠t⌿⍨¯1⌽nss)∧(V≠t⌿⍨¯1⌽nss)∨Z≠t⌿⍨¯2⌽nss:ERR ⎕SIGNAL 2
ERR←':ENDNAMESPACE KEYWORD MUST APPEAR ALONE ON A LINE'
∨⌿Z≠t⌿⍨⊃1 ¯1∨.⌽⊂nse:ERR ⎕SIGNAL 2
t[nsi←⍸1⌽nss]←M ⋄ t[nei←⍸1⌽nse]←-M
n[i]←n[1+i←⍸(t=M)∧V=1⌽t] ⋄ end[nsi]←end[nei]
x←⍸p=⍳≢p ⋄ d←+⍀(t[x]=M)+-t[x]=-M
0≠⊃⌽d:':NAMESPACE KEYWORD MISSING :ENDNAMESPACE PAIR'⎕SIGNAL 2
p[x]←x[D2P ¯1⌽d]

⍝ Delete unnecessary namespace nodes from the tree, leave only M's
msk←~nss∨((¯1⌽nss)∧t=V)∨nse∨1⌽nse
t k n pos end⌿⍨←⊂msk ⋄ p←(⍸~msk)(⊢-1+⍸)msk⌿p
```
This code is used in chunk 22.
Uses `SIGNAL` 20b.

In the parser, the `xn` and `xt` fields are not part of the AST proper, but form an auxiliary analysis that is exceptionally useful, and so we include this as a part of the output of the parser. After parsing a module, we want to extract out the top-level bindings and what their types are, which we can then use to feed into things like the linker and other areas that might need to know what names are available in a given module. Top-level bindings are identified as bindings that appear as a part of an initialization function, also known as `F0`.

117b ⟨*Compute parser exports* 117b⟩≡
```
msk←(t=B)∧k[I@{t[⍵]≠F}⍣≡⍨p]=0
xn←(0⍴⊂''),msk⌿n ⋄ xt←msk⌿k
```
This code is used in chunk 17.
Defines:
`xn`, used in chunk 20a.
`xt`, used in chunk 20a.

117c ⟨*Record exported top-level bindings* 117c⟩≡
```
xi←⍸(t=B)∧k[r]=0
```
This code is used in chunk 23.
Defines:
`xi`, used in chunks 23–25.

117d ⟨*Node* ⟷ *Generator mapping* 24c⟩+≡
```
gck,←⊂F 0
gcv,←⊂'Fz'
```
This code is used in chunk 25b.

118 ⟨*Node-specific code generators* 24e⟩+≡

```
Fz←{id←⍕5⊃⍺ ⋄ awc←∨⌿(3⌷x){(⍵∊A O)∨(⍵=E)∧⍺>0}2⌷x←⍉⊃⍪⌿⍵
        z ←⊂'int init',id,' = 0;'
        z,←⊂''
        z,←⊂'EXPORT int'
        z,←⊂'init(void)'
        z,←⊂'{'
        z,←⊂' return fn',id,'(NULL, NULL, NULL, NULL);'
        z,←⊂'}'
        z,←⊂''
        z,←⊂'int'
        z,←⊂'fn',id,'(struct array **z, '
        z,←⊂'    struct array *l, struct array *r, void *fv[])'
        z,←⊂'{'
        z,←⊂'       void    *stk[128];'
        z,←⊂'       void    **stkhd;'
        z,← awc⌿⊂'        void    *a, *w;'
        z,← awc⌿⊂'        struct  closure *c;'
        z,←⊂''
        z,←⊂'       if (init',id,')'
        z,←⊂'               return 0;'
        z,←⊂''
        z,←⊂'       stkhd = &stk[0];'
        z,←⊂'       init',id,' = 1;'
        z,←⊂'       cdf_init();'
        z,←⊂''
        z,← '  ',¨⊃,⌿dis¨⍵
        z,←⊂'       return 0;'
        z,←⊂'}'
        z,←⊂''
z}
```

This code is used in chunk 25b.

119a ⟨*init.c* 119a⟩≡
```
#include "codfns.h"

int
init(void);

EXPORT int
cdf_init(void)
{
        return init();
}
```
Root chunk (not used in this document).
Defines:
`init.c`, used in chunk 119b.
Uses `codfns` 7, `codfns.h` 34a, and `EXPORT` 35.

119b ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling rtm/init.c..."
notangle -R'init.c' codfns.nw > rtm/init.c
```
This code is used in chunk 134.
Uses `codfns` 7 and `init.c` 119a.

119c ⟨*C runtime declarations* 39a⟩+≡
```
DECLSPEC int cdf_init(void);
```
This code is used in chunk 34a.
Uses `DECLSPEC` 35.

6.16.3 Structured Programming Statements

119d ⟨*Verify that all structured statements appear within trad-fns* 119d⟩≡
```
msk←kws∊KW~':NAMESPACE' ':ENDNAMESPACE' ':SECTION' ':ENDSECTION'
∨⌿msk←msk∧~km⌿tm:{
        msg←2'STRUCTURED STATEMENTS MUST APPEAR WITHIN TRAD-FNS'
        msg SIGNAL ∊{x+⍳end[⍵]-x←pos[⍵]}¨⍸km⍀msk
}⍬
```
This code is used in chunk 22.
Uses `SIGNAL` 20b.

119e ⟨*Convert* `M` *nodes to* `FO` *nodes* 119e⟩≡
```
t←F@{t=M}t
```
This code is used in chunk 22.

7 Runtime Primitives

7.1 Addition/Identity

120a ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'+' ⋄ nams,←⊂'add'
```
This code is used in chunk 25b.

7.2 And (Logical)

120b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'∧' ⋄ nams,←⊂'and'
```
This code is used in chunk 25b.

7.3 Bracket

120c ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'[' ⋄ nams,←⊂'brk'
```
This code is used in chunk 25b.

7.4 Catenate (First/Last Axis)

120d ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,',' ⋄ nams,←⊂'cat'
syms,←⊂,'⍪' ⋄ nams,←⊂'ctf'
```
This code is used in chunk 25b.

7.5 Circle/Trigonometrics

120e ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'○' ⋄ nams,←⊂'cir'
```
This code is used in chunk 25b.

7.6 Commute

120f ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍨' ⋄ nams,←⊂'com'
```
This code is used in chunk 25b.

7.7 Compose

121a ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
  syms,←⊂,'∘' ⋄ nams,←⊂'jot'
```
This code is used in chunk 25b.

7.8 Convolve

121b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
  syms,←⊂,'⎕CONV' ⋄ nams,←⊂'conv'
```
This code is used in chunk 25b.

7.9 Decode

121c ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
  syms,←⊂,'⊥' ⋄ nams,←⊂'dec'
```
This code is used in chunk 25b.

7.10 Disclose

121d ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
  syms,←⊂,'⊃' ⋄ nams,←⊂'dis'
```
This code is used in chunk 25b.

7.11 Division/Reciprocal

121e ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
  syms,←⊂,'÷' ⋄ nams,←⊂'div'
```
This code is used in chunk 25b.

7.12 Drop

121f ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
  syms,←⊂,'↓' ⋄ nams,←⊂'drp'
```
This code is used in chunk 25b.

7.13 Each

121g ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
  syms,←⊂,'¨' ⋄ nams,←⊂'map'
```
This code is used in chunk 25b.

7.14 Enclose

122a ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'⊂' ⋄ nams,←⊂'par'
```
This code is used in chunk 25b.

7.15 Encode

122b ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'⊤' ⋄ nams,←⊂'enc'
```
This code is used in chunk 25b.

7.16 Equal

122c ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'=' ⋄ nams,←⊂'eql'
```
This code is used in chunk 25b.

7.17 Exponent

122d ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'*' ⋄ nams,←⊂'exp'
```
This code is used in chunk 25b.

7.18 Factorial/Binomial

122e ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'!' ⋄ nams,←⊂'fac'
```
This code is used in chunk 25b.

7.19 Fast Fourier Transforms

122f ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'⎕FFT' ⋄ nams,←⊂'fft'
```
This code is used in chunk 25b.

122g ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
syms,←⊂,'⎕IFFT' ⋄ nams,←⊂'ift'
```
This code is used in chunk 25b.

7.20 Find

123a ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍷' ⋄ nams,←⊂'fnd'
```
This code is used in chunk 25b.

7.21 Grade Down

123b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍒' ⋄ nams,←⊂'gdd'
```
This code is used in chunk 25b.

7.22 Grade Up

123c ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍋' ⋄ nams,←⊂'gdu'
```
This code is used in chunk 25b.

7.23 Greater Than

123d ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'>' ⋄ nams,←⊂'gth'
```
This code is used in chunk 25b.

7.24 Greater Than or Equal

123e ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'≥' ⋄ nams,←⊂'gte'
```
This code is used in chunk 25b.

7.25 Index

123f ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⌷' ⋄ nams,←⊂'sqd'
```
This code is used in chunk 25b.

7.26 Index Generator

123g ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍳' ⋄ nams,←⊂'iot'
```
This code is used in chunk 25b.

7.27 Inner Product

124a ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'.' ⋄ nams,←⊂'dot'
```
This code is used in chunk 25b.

7.28 Intersection

124b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'∩' ⋄ nams,←⊂'int'
```
This code is used in chunk 25b.

7.29 Left

124c ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⊣' ⋄ nams,←⊂'lft'
```
This code is used in chunk 25b.

7.30 Less Than

124d ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'<' ⋄ nams,←⊂'lth'
```
This code is used in chunk 25b.

7.31 Less Than or Equal

124e ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'≤' ⋄ nams,←⊂'lte'
```
This code is used in chunk 25b.

7.32 Logarithm

124f ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍟' ⋄ nams,←⊂'log'
```
This code is used in chunk 25b.

7.33 Match

124g ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'≡' ⋄ nams,←⊂'eqv'
```
This code is used in chunk 25b.

7.34 Matrix Division

125a ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⌹' ⋄ nams,←⊂'mdv'
```
This code is used in chunk 25b.

7.35 Maximum/Ceiling

125b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⌈' ⋄ nams,←⊂'max'
```
This code is used in chunk 25b.

7.36 Membership

125c ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'∊' ⋄ nams,←⊂'mem'
```
This code is used in chunk 25b.

7.37 Minimum/Floor

125d ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⌊' ⋄ nams,←⊂'min'
```
This code is used in chunk 25b.

7.38 Multiplication

125e ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'×' ⋄ nams,←⊂'mul'
```
This code is used in chunk 25b.

7.39 Nest/Partition

125f ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⊆' ⋄ nams,←⊂'nst'
```
This code is used in chunk 25b.

7.40 Not

125g ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'~' ⋄ nams,←⊂'not'
```
This code is used in chunk 25b.

7.41 Not And (Logical)

126a ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
  syms,←⊂,'⍲' ⋄ nams,←⊂'nan'
```
This code is used in chunk 25b.

7.42 Not Equal

126b ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
  syms,←⊂,'≠' ⋄ nams,←⊂'neq'
```
This code is used in chunk 25b.

7.43 Not Match

126c ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
  syms,←⊂,'≢' ⋄ nams,←⊂'nqv'
```
This code is used in chunk 25b.

7.44 Not Or (Logical)

126d ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
  syms,←⊂,'⍱' ⋄ nams,←⊂'nor'
```
This code is used in chunk 25b.

7.45 Or (Logical)

126e ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
  syms,←⊂,'∨' ⋄ nams,←⊂'lor'
```
This code is used in chunk 25b.

7.46 Outer Product

126f ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
  syms,←⊂,'∘.' ⋄ nams,←⊂'oup'
```
This code is used in chunk 25b.

7.47 Power

126g ⟨*Symbol* ⟷ *Name mapping* 24b⟩+≡
```
  syms,←⊂,'⍣' ⋄ nams,←⊂'pow'
```
This code is used in chunk 25b.

7.48 Rank

127a ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍤' ⋄ nams,←⊂'rnk'
```
This code is used in chunk 25b.

7.49 Reduce

127b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'/' ⋄ nams,←⊂'red'
```
This code is used in chunk 25b.

127c ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⌿' ⋄ nams,←⊂'rdf'
```
This code is used in chunk 25b.

7.50 Roll

127d ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'?' ⋄ nams,←⊂'rol'
```
This code is used in chunk 25b.

7.51 Rotate (First/Last Axis)

127e ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⌽' ⋄ nams,←⊂'rot'
syms,←⊂,'⊖' ⋄ nams,←⊂'rtf'
```
This code is used in chunk 25b.

7.52 Residue

127f ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'|' ⋄ nams,←⊂'res'
```
This code is used in chunk 25b.

7.53 Right

127g ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⊢' ⋄ nams,←⊂'rgt'
```
This code is used in chunk 25b.

128a ⟨*APL Primitives* 128a⟩≡
```
rgt←{⍵}
```
This code is used in chunk 32a.
Defines:
`rgt`, used in chunk 131.

7.54 Scalar Each

128b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'%s' ⋄ nams,←⊂'scl'
```
This code is used in chunk 25b.

7.55 Scan

128c ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'\' ⋄ nams,←⊂'scn'
```
This code is used in chunk 25b.

128d ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍀' ⋄ nams,←⊂'scf'
```
This code is used in chunk 25b.

7.56 Shape

128e ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍴' ⋄ nams,←⊂'rho'
```
This code is used in chunk 25b.

7.57 Subtraction

128f ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'-' ⋄ nams,←⊂'sub'
```
This code is used in chunk 25b.

7.58 Take

128g ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'↑' ⋄ nams,←⊂'tke'
```
This code is used in chunk 25b.

7.59 Transpose

129a ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'⍉' ⋄ nams,←⊂'trn'
```
This code is used in chunk 25b.

7.60 Union

129b ⟨*Symbol ⟷ Name mapping* 24b⟩+≡
```
syms,←⊂,'∪' ⋄ nams,←⊂'unq'
```
This code is used in chunk 25b.

8 Utilities

8.1 Must haves

There are some APL functions that are so critical as to be worthy of primitive status.

- Indexing
- Under
- Assert

129c ⟨*Must Have APL Utilities* 129c⟩≡
```
I←{(⊂⍵)⌷⍺}
U←{⍺←⊢ ⋄ ⍵⍵⍣¯1⊢⍺ ⍺⍺⍥⍵⍵ ⍵}
assert←{
        ⍺←'assertion failure'
        0∊⍵:⍎'⍺ ⎕SIGNAL 8'
        1:shy←0
}
```
This code is used in chunk 7.
Defines:
`assert`, used in chunk 22.
Uses `SIGNAL` 20b.

8.2 AST Pretty-printing

130 ⟨*Pretty-printing AST trees* 130⟩≡

```
dct←{⍺[(2×2≠/n,0)+(1↑⍨≢m)+m+n←⌽∨\⌽m←' '≠⍺⍺ ⍵]⍵⍵ ⍵}
dlk←{((x⌷⍴⍵)↑[x←2|1+⍵⍵]⍺),[⍵⍵]⍺⍺@(⊂0 0)⍣('┌'=⊃⍵)⊢⍵}

dwh←{
        z←⊃⍪/((≢¨⍺),¨⊂⌈/≢∘⍉¨⍺)↑¨⍺
        ⍵('┬'dlk 1)' │├┌└─'(0⌷⍉)dct,z
}
dwv←{
        z←⊃{⍺,' ',⍵}/(1+⌈/≢¨⍺){⍺↑⍵⍪⍨'│'↑⍨≢⍉⍵}¨⍺
        ⍵('├'dlk 0)' ─┬┌┐│'(0⌷⊢)dct(⊣⍪1↓⊢)z
}

lb3←{
        ⍺←⍳≢⊃⍵
        z←(N∆{⍺[⍵]}@2⊢(2⊃⍵){⍺[|⍵]}@{0>⍵}@4↑⊃⍵)[⍺;]
        '(',¨')',¨⍨{⍺,';',⍵}⌿⍕¨z
}

pp3←{
        ⍺←'○' ⋄ lbl←⍺⍴⍨≢⍵
        d←(⍳≢⍵)≠⍵ ⋄ _←{z⊣d+←⍵≠z←⍺[⍵]}⍣≡⍨⍵
        lyr←{
                i←⍸⍺=d
                k v←↓⍉⍵⍵[i],∘⊂⌸i
                (⍵∘{⍺[⍵]}¨v)⍺⍺¨@k⊢⍵
        }⍵
        (⍵=⍳≢⍵)⌿⊃⍺⍺ lyr⌿(1+⍳⌈/d),⊂⍉∘⍪∘⍕¨lbl
}
```

This code is used in chunk 7.
Defines:
- `dct`, never used.
- `dlk`, never used.
- `dwh`, never used.
- `dwv`, never used.
- `lb3`, never used.
- `pp3`, never used.

8.3 Debugging utilities

The following utilities help to improve quality of life when working with the Co-dfns source code.

The `DISPLAY` function is taken from `https://dfns.dyalog.com` and helps to make debugging easier by allowing us to thread `DISPLAY` calls into expressions. I prefer to do something like this:

```
... {⍵⊣⎕←#.DISPLAY ⍵} ...
```

The function itself returns the character rendering of the code, so the above little expression is one that I use to insert and do debugging within an expression.

131 ⟨`DISPLAY` *Utility* 131⟩≡

```
  DISPLAY←{
          ⎕IO ⎕ML←0
          ⍺←1 ⋄ chars←⍺⊃'..''''|-' '┌┐└┘│─'
          tl tr bl br vt hz←chars
          box←{
                  vrt hrz←(¯1+⍴⍵)⍴¨vt hz
                  top←(hz,'⊖→')[¯1↑⍺],hrz
                  bot←(⊃⍺),hrz
                  rgt←tr,vt,vrt,br
                  lax←(vt,'⌽↓')[¯1↓1↓⍺],¨⊂vrt
                  lft←⍉tl,(↑lax),bl
                  lft,(top⍪⍵⍪bot),rgt
          }
          deco←{⍺←type open ⍵ ⋄ ⍺,axes ⍵}
          axes←{(-2⌈⍴⍴⍵)↑1+×⍴⍵}
          open←{(1⌈⍴⍵)⍴⍵}
          trim←{(~1 1⍷∧⌿⍵=' ')/⍵}
          type←{{(1=⍴⍵)⊃'+'⍵}∪,char¨⍵}
          char←{⍬≡⍴⍵:hz ⋄ (⊃⍵∊'¯',⎕D)⊃'#~'}∘⍕
          line←{(6≠10|⎕DR' '⍵)⊃' -'}
          {
                  0=≡⍵:' '⍪(open ⎕FMT ⍵)⍪line ⍵
                  1 ⍬≡(≡⍵)(⍴⍵):'∇' 0 0 box ⎕FMT ⍵
                  1=≡⍵:(deco ⍵)box open ⎕FMT open ⍵
                  ('∊'deco ⍵)box trim ⎕FMT ∇¨open ⍵
          }⍵
  }
```

Root chunk (not used in this document).
Defines:
`DISPLAY`, used in chunk 132.
Uses `box` 104b, `rgt` 128a, `⎕IO` 10a, and `⎕ML` 10a.

I also define a function PP that encapsulates the above usage pattern that I like to use, making the whole thing less verbose and a little more convenient.

132a ⟨PP *Utility* 132a⟩≡
```
PP←{ω⊣⎕←#.DISPLAY ω}
```
Root chunk (not used in this document).
Defines:
PP, used in chunks 27, 75, 76b, and 132b.
Uses DISPLAY 131.

Both of these function exist outside of the codfns namespace and so they get their own files inside of the src\ directory.

132b ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling src/DISPLAY.aplf..."
notangle -R'[[DISPLAY]] Utility' codfns.nw > src/DISPLAY.aplf

echo "Tangling src/PP.aplf..."
notangle -R'[[PP]] Utility' codfns.nw > src/PP.aplf
```
This code is used in chunk 134.
Defines:
DISPLAY.aplf, never used.
PP.aplf, never used.
Uses codfns 7, DISPLAY 131, PP 132a, and src 139.

8.4 Reading and Writing Files

It is helpful to be able to easily write files to disk, and the following put and tie utilities help us to do so when we want to. These are pretty standard, but they could maybe be replaced by ⎕NPUT or something like that.

132c ⟨*Basic* tie *and* put *utilities* 132c⟩≡
```
tie←{
        0::⎕SIGNAL ⎕EN
        22::ω ⎕NCREATE 0
        0 ⎕NRESIZE ω ⎕NTIE 0
}

put←{
        s←(¯128+256|128+'UTF-8'⎕UCS ω)⎕NAPPEND(t←tie α)83
        1:r←s⊣⎕NUNTIE t
}
```
This code is used in chunks 7 and 138b.
Defines:
put, used in chunks 26, 138b, and 139.
tie, used in chunk 138b.
Uses SIGNAL 20b.

8.5 XML Rendering

133a ⟨*XML Rendering* 133a⟩≡

```
  Xml←{⍺←0
           ast←⍺{d i←P2D⊃⍵ ⋄ i∘{⍵[⍺]}¨(⊂d),1↓⍺↓⍵}⍣(0≠⍺)⊢⍵
           d t k n←4↑ast
           cls←N∆[t],¨('-..'[1+×k]),¨⍕¨|k
           fld←{((≢⍵)↑3↓f∆),⍪⍵}¨↓⍉↑3↓ast
           ⎕XML⍉↑d cls(⊂'')fld
  }
```

This code is used in chunk 7.
Defines:
`Xml`, never used.

8.6 Detecting the Operating System

It is quite helpful to be able to easily detect the operating system that we are on. This turns out to be helpful in more areas than just the compiler.

133b ⟨*The* `opsys` *utility* 133b⟩≡

```
  opsys←{⍵⊃⍨'Win' 'Lin' 'Mac'⍳⊂3↑⊃'.'⎕WG'APLVersion'}
```

This code is used in chunks 7, 135c, and 137d.
Defines:
`opsys`, used in chunks 26, 135c, and 137d.

9 Developer Infrastructure

9.1 Building the Compiler

The Co-dfns compiler is written, developed, and distributed as a literate program. For more information about literate programming, see the resources available at `http://literateprogramming.com/`. We use noweb as our preferred literate programming tool because it is eminently simple, while still handling the majority of our needs and producing high quality output in LaTeX format with all the important elements of literate programming, including live hyperlinking and cross-references.

9.1.1 Tangling the Source

The process of tangling produces the executable source code for the compiler. Importantly, the tangled output is *not* meant to be used as the primary means of reading or debugging the source. Instead, it is meant primarily as the machine readable version of the code only.

With noweb, we need to invoke `notangle` once for each of the chunks that we wish to use to produce an output file. To make this easy, we build up a script to do this work for us.

For Linux and Mac, the following bash script creates these files. We use a separate chunk that we build up incrementally throughout the rest of this document as a record of all the chunks that we should create. Notice that we explicitly tangle the `TANGLE.sh` file as the last thing that we do; this helps to ensure that we are reliably executing the rest of the script before changing the contents of the file, as some systems will be affected and change execution behavior in strange ways if we change the `TANGLE.sh` file early on in the execution of the file.

134 ⟨TANGLE.sh 134⟩≡

```
#!/bin/bash

⟨Tangle Commands 8⟩

echo "Tangling TANGLE.sh..."
notangle -R'[[TANGLE.sh]]' codfns.nw > TANGLE.sh
```

Root chunk (not used in this document).
Defines:
`TANGLE.sh`, used in chunk 135a.
Uses `codfns` 7 and `TANGLE` 135c.

On Windows, the best way that we have found to do this is by installing noweb using the Cygwin project and then calling `TANGLE.sh` from a local `TANGLE.bat` file. This document assumes that you have already successfully built and installed via Cygwin a working Icon-driven noweb installation.

Users who prefer to work in a UNIX fashion via Cygwin or some other subsystem on Windows can follow the build scripts directly. For developers who prefer to work in a primarily Windows environment, the following `TANGLE.bat` build script assists in handling the calls into Cygwin so that you do not need to have a Cygwin terminal open all the time.

135a ⟨`TANGLE.bat` 135a⟩≡
```
set SH=C:\cygwin64\bin\bash.exe -l -c
%SH% "cd $OLDPWD && ./TANGLE.sh"
```
Root chunk (not used in this document).
Defines:
`TANGLE.bat`, used in chunk 135b.
Uses `TANGLE` 135c and `TANGLE.sh` 134.

135b ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling TANGLE.bat..."
notangle -R'[[TANGLE.bat]]' codfns.nw > TANGLE.bat
```
This code is used in chunk 134.
Uses `codfns` 7, `TANGLE` 135c, and `TANGLE.bat` 135a.

When tangled to the `TANGLE.aplf` file, the following script enables the user to simply type `TANGLE` within a Dyalog APL session to update the code tree from within Dyalog itself. This is much more convenient than keeping a Cygwin Terminal session open along with a Dyalog APL session while programming.

Note: this command expects to be run from within the root of the repository, not from, say, within the `testing` *directory.*

135c ⟨`TANGLE` 135c⟩≡
```
TANGLE;opsys
⟨The opsys utility 133b⟩
⎕CMD opsys '.\TANGLE.bat' './TANGLE.sh' './TANGLE.sh'
```
Root chunk (not used in this document).
Defines:
`TANGLE`, used in chunks 134 and 135.
Uses `opsys` 133b.

135d ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling TANGLE.aplf..."
notangle -R'[[TANGLE]]' codfns.nw > src/TANGLE.aplf
```
This code is used in chunk 134.
Defines:
`TANGLE.aplf`, never used.
Uses `codfns` 7, `src` 139, and `TANGLE` 135c.

9.1.2 Weaving the Source

Weaving is the process by which we produce the final printed output of this document, intended for reading and general human consumption. We rely on the LaTeX typesetting system to do this. Moreover, because we make heavy use of UTF-8 and prefer to have our own fonts installed and used, it is necessary to use the `xelatex` system instead of the typical LaTeX engine. In order to get the indexing right, we must run the engine twice. The first run will update the indexing files that will be picked up on the second run and incorporated into the final document. Note, we have tried to use the `lualatex` engine, which in theory should work just as well as the `xelatex` engine, but we get a strange error relating to noweb's style file, so we stick with `xelatex` for now.

Running this script also depends on having the appropriate fonts installed. In this case, please ensure that the following fonts are installed in your Windows font system so that they can be picked up by the TeX engine.

- Libre Baskerville (Regular, Italic, Bold)
- APL385 Unicode
- Lucida Sans Unicode
- Cambria Math

If you do not wish to use these fonts, edit the font specifications at the top of `codfns.nw` to the fonts that you do wish to use.

Note the use of `-delay -index` for options. We want to generate indexing, but we also need to make sure that we can use some of our own packages in the system,

Note: this command expects to be run from within the root of the repository, not from, say, within the `testing` *directory.*

136 ⟨`WEAVE.sh` 136⟩≡

```
#!/bin/bash
mkdir -p woven
noweave -delay -index codfns.nw > woven/codfns.tex
cd woven
xelatex --shell-escape codfns
xelatex --shell-escape codfns
```

Root chunk (not used in this document).
Defines:
`WEAVE.sh`, used in chunk 137.
Uses `codfns` 7.

137a ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling WEAVE.sh..."
notangle -R'[[WEAVE.sh]]' codfns.nw > WEAVE.sh
```
This code is used in chunk 134.
Uses `codfns` 7, `WEAVE` 137d, and `WEAVE.sh` 136.

And just like the tangling code, we want to define a `TANGLE.bat` batch file to call the Cygwin environment from Windows.

137b ⟨`WEAVE.bat` 137b⟩≡
```
set SH=C:\cygwin64\bin\bash.exe -l -c
%SH% "cd $OLDPWD && ./WEAVE.sh"
```
Root chunk (not used in this document).
Defines:
`WEAVE.bat`, used in chunk 137c.
Uses `WEAVE` 137d and `WEAVE.sh` 136.

137c ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling WEAVE.bat..."
notangle -R'[[WEAVE.bat]]' codfns.nw > WEAVE.bat
```
This code is used in chunk 134.
Uses `codfns` 7, `WEAVE` 137d, and `WEAVE.bat` 137b.

Like the ⟨`TANGLE` *Command* (never defined)⟩, the following command, when tangled to the `WEAVE.aplf` file enables weaving in a the Dyalog APL session by executing the `WEAVE` command.

137d ⟨`WEAVE` 137d⟩≡
```
WEAVE;opsys
⟨The opsys utility 133b⟩
⎕CMD opsys '.\WEAVE.bat' './WEAVE.sh' './WEAVE.sh'
```
Root chunk (not used in this document).
Defines:
`WEAVE`, used in chunk 137.
Uses `opsys` 133b.

137e ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling src/WEAVE.aplf..."
notangle -R'[[WEAVE]]' codfns.nw > src/WEAVE.aplf
```
This code is used in chunk 134.
Defines:
`WEAVE.aplf`, never used.
Uses `codfns` 7, `src` 139, and `WEAVE` 137d.

9.2 Building the Runtime

One of our goals with the Co-dfns runtime is to write as much of it as possible in APL. This means that we want to have at minimum a very small kernel that has been written in C, while most of the rest of the code is implemented in some APL files. This leads to a three part breakdown of the process to build the runtime.

138a ⟨*Build the runtime* 138a⟩≡
```
⟨Compile the primitives in prim.apln 139⟩
⟨Build codfns.dll DLL 140a⟩
⟨Copy the runtime files into tests\ 140b⟩
```
This code is used in chunk 138b.

We define the command `MK∆RTM` to build the runtime. This command takes a path to the root directory of the Co-dfns repository; this is to allow us to rebuild the runtime from anywhere in the system if we so choose.

138b ⟨`MK∆RTM` 138b⟩≡
```
MK∆RTM path;put;tie;src;vsbat;vsc;wsd

⟨Basic tie and put utilities 132c⟩
⟨Build the runtime 138a⟩
```
Root chunk (not used in this document).
Defines:
`MK∆RTM`, used in chunk 138c.
Uses `put` 132c, `src` 139, `tie` 132c, `vsbat` 140a, `vsc` 140a, and `wsd` 140a.

This file is another of our external utilities that exists outside of the `codfns` namespace, so it gets its own file in `src\`.

138c ⟨*Tangle Commands* 8⟩+≡
```
echo "Tangling src/MK∆RTM.aplf..."
notangle -R'[[MK∆RTM]]' codfns.nw > src/MK∆RTM.aplf
```
This code is used in chunk 134.
Defines:
`MK∆RTM.aplf`, never used.
Uses `codfns` 7, `MK∆RTM` 138b, and `src` 139.

The first step we must take is producing an appropriate C file that contains the primitives that we have defined in `prim.apln`. This means that we want to only compile the code in `prim.apln` as far as producing the C code. Since we do not have a full blown runtime yet, we will be compiling the `prim.c` file along with the rest of the runtime code, instead of the normal build process, which assumes that we already have a working runtime. This means that we only invoke the `GC TT PS` passes of the compiler pipeline, while avoiding the `CC` pass. We use the SALT system to load the source from `prim.apln` and then run the compiler passes that we want before storing the resulting code in the `rtm\prim.c` file.

139 ⟨*Compile the primitives in* `prim.apln` 139⟩≡

```
src←⎕SRC ⎕SE.SALT.Load path,'\rtm\prim.apln'
(path,'\rtm\prim.c')put codfns.{GC TT PS ⍵}src
```

This code is used in chunk 138a.
Defines:
`src`, used in chunks 8, 13, 16b, 23, 132b, 135d, 137, and 138.
Uses `codfns` 7, `prim` 32a, `PS` 17, and `put` 132c.

Once we have the `rtm\prim.c` file written appropriately, we can run the main compiler process. For simplicity, we just compile all of the `.c` files that are found in the `rtm\` subdirectory. We must ensure that we are appropriatelly invoking our ArrayFire dependencies as well as producing the appropriate debugging symbols most of the time.

140a ⟨*Build* `codfns.dll` *DLL* 140a⟩≡

```
vsbat←#.codfns.VS∆PATH
vsbat,'\VC\Auxiliary\Build\vcvarsall.bat'
wsd←path,'\'

vsc←'%comspec% /C ""',vsbat,'" amd64'
vsc,←'  && cd "',wsd,'\rtm"'
vsc,←'  && cl /MP /W3 /wd4102 /wd4275'
vsc,←'    /Od /Zc:inline /Zi /FS'
vsc,←'    /Fo".\\" /Fd"codfns.pdb"'
vsc,←'    /WX /MD /EHsc /nologo'
vsc,←'    /I"%AF_PATH%\include"'
vsc,←'    /D"NOMINMAX" /D"AF_DEBUG" /D"EXPORTING"'
vsc,←'    "*.c" /link /DLL /OPT:REF'
vsc,←'    /INCREMENTAL:NO /SUBSYSTEM:WINDOWS'
vsc,←'    /LIBPATH:"%AF_PATH%\lib"'
vsc,←'    /DYNAMICBASE "af',codfns.AF∆LIB,'.lib"'
vsc,←'    /OPT:ICF /ERRORREPORT:PROMPT'
vsc,←'    /TLBID:1 /OUT:"codfns.dll""'
```

This code is used in chunk 138a.
Defines:
`vsbat`, used in chunks 26 and 138b.
`vsc`, used in chunks 26, 138b, and 140b.
`wsd`, used in chunks 138b and 140b.
Uses `AF∆LIB` 11, `codfns` 7, `EXPORTING` 35, and `VS∆PATH` 12.

Finally, in order to write up the test harness to work right, we must copy the appropriate runtime files into the `tests\` directory so that we can find them when we finally start running our code there.

140b ⟨*Copy the runtime files into* `tests\` 140b⟩≡

```
⎕CMD ⎕←vsc
⎕CMD ⎕←'copy "',wsd,'rtm\codfns.h" "',wsd,'tests\"'
⎕CMD ⎕←'copy "',wsd,'rtm\codfns.exp" "',wsd,'tests\"'
⎕CMD ⎕←'copy "',wsd,'rtm\codfns.lib" "',wsd,'tests\"'
⎕CMD ⎕←'copy "',wsd,'rtm\codfns.pdb" "',wsd,'tests\"'
⎕CMD ⎕←'copy "',wsd,'rtm\codfns.dll" "',wsd,'tests\"'
```

This code is used in chunk 138a.
Uses `codfns` 7, `codfns.h` 34a, `vsc` 140a, and `wsd` 140a.

9.3 Loading the Compiler

In order to load the compiler into an APL session as well as all the development utilities, we assume that you have first managed to either load up a session with a bootstrapped version of the TANGLE command or that you already have a tangled src\ directory. If the src\ directory has not yet been created by running the TANGLE command, then this must be done before loading the compiler system. After tangling, the compiler can be loaded using the provided LOAD shortcut. This shortcut is meant to use the Dyalog Link system for hotloading the files in src\ into the root namespace. We do so through the following link command:

```
Link.Create # src -source=dir -watch=dir
```

This means that we want to link the src\ directory into the # namespace, but we also want to make sure that we only pull changes that come from the filesystem. This is because we are editing the code via the WEB document, and we do not want to risk having some intermediate representation that isn't accurate and that doesn't flow the right way; we want all appropriate changes to begin in the WEB document and then, and only then, flow into the session. This also allows us to make some modifications to the code for testing and experimentation inside of the session without consideration for the code outside of the session, and such changes will be removed or forgotten on the next TANGLE command.

To set this up, we also ensure that we begin our work within the root Co-dfns repository directory, as this is where we expect to run the TANGLE and WEAVE commands.

There is unfortunately only a limited range of possibilities for linking in a new directory as we wish to do. The method we choose to use is launching a fresh Dyalog APL session and then using an LX expression from the command line to do the actual linking using the ⎕SE.UCMD functionality. I personally find this to be rather hackish, and I hope that an alternative approach to doing this will show up in the near future. Nonetheless, the arguments that we pass to dyalog.exe look something like this:

```
LX="⎕SE.UCMD'Link.Create # src -source=dir -watch=dir'"
```

If you do not use the LOAD shortcut, you can use the above command to do the linking manually.

10 Chunks

⟨* 7⟩

11 Index

12 GNU AFFERO GPL

Preamble

The GNU Affero General Public License is a free, copyleft license for software and other kinds of works, specifically designed to ensure cooperation with the community in the case of network server software.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, our General Public Licenses are intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

Developers that use our General Public Licenses protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License which gives you legal permission to copy, distribute and/or modify the software.

A secondary benefit of defending all users' freedom is that improvements made in alternate versions of the program, if they receive widespread use, become available for other developers to incorporate. Many developers of free software are heartened and encouraged by the resulting cooperation. However, in the case of software used on network servers, this result may fail to come about. The GNU General Public License permits making a modified version and letting the public access it on a server without ever releasing its source code to the public.

The GNU Affero General Public License is designed specifically to ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

An older license, called the Affero General Public License and published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.
The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU Affero General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

   A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

   The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

   The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   (d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   (a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   (b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   (c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
   (d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
   (e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the

Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Remote Network Interaction; Use with the GNU General Public License.

    Notwithstanding any other provision of this License, if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the

Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the work with which it is combined will remain governed by version 3 of the GNU General Public License.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU Affero General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU Affero General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU Affero General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU Affero General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES

PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear> <name of author>

This program is free software: you can redistribute it and/or
modify it under the terms of the GNU Affero General Public
License as published by the Free Software Foundation, either
version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public
License along with this program. If not, see
<https://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If your software can interact with users remotely through a computer network, you should also make sure that it provides a way for users to get its source. For example, if your program is a web application, its interface could display a “Source” link that leads users to an archive of the code. There are many ways you could offer source, and different solutions will be better for different programs; see section 13 for the specific requirements.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU AGPL, see `https://www.gnu.org/licenses/`.