CPL Language

Table of Contents

Introduction

A while ago, I tried my hand at building a compiler and processor for a language I called TPL/0. I had initially tried to write it in Rust but gave up because I just couldn't get how Rust worked; eventually I wrote it in C++ using YACC and LEX; it wasn't very useful. But I couldn't let Rust go and decided to try again to at least learn some Rust. After going through several sections of the tutorial I began to get an inkling of how the language worked and, wouldn't you know it, I decided to come back to TPL/0 using Rust. This time I called the language CPL (of course! *Carl's Programming Language*). It's a bit different from TPL/0 due to how I'm doing the parsing and I decided to generate my own pseudo machine code that the program would interpret rather than treating the "compiler" as a front-end to Rust or C or C++ code that needed to be compiled. Okay, so this is weird… I admit it. But, well, shit. Why not.

And so we're clear: CPL is "yet another 'C'-like programming language" as opposed to what? Well, maybe Cobol, Fortran or Lisp or some other functional programming language (F++?). But let's save that for another day, shall we?

And, so we're clear: CPL is probably not suitable for any serious production level applications. I wrote it as an exercise and now I know enough Rust to be dangerous.

Running CPL Programs

Note that I'll be referring to the CPL program, the one written in Rust, as the "interpreter" but it's really a compiler that generates interpreted code *and* the interpreter to run it. And I'll refer to the program written in the CPL language as the "CPL program".

Running a CPL program outside of Rust you use:

```
<path to cpl compiler> <path to CPL source> [switches] [<arguments>]
```

Within the context of a Rust development environment, you use:

```
cargo run <path to CPL sourc> [<switches>] [<arguments>]
```

<switches> are all optional and control the behavior of the interpreter and are described in some detail in Appendix II. They are mostly there for debugging anyway.

The *<arguments>* are an optional space separated list of items that are passed into the CPL program program and whose meaning is defined by the CPL program.

Basic Grammar

Most of this you will already know from the many other 'C'-like programming languages but for completeness I describe the language elements in the following sections.

Variables

A variable represents the contents of a location in the pseudo machine's data memory (see *Pseudo Machine*). The data type of a variable is determined by how it is created and updated. Unlike C, Java, C++, etc. etc. a CPL variable is not declared as *having* a particular data type (with one small exception which I'll discuss within the context of parameter passing).

Data Types

A variable and the return value of a function can take on one of the following data types:

Numeric

Internally, all numbers are stored as Rust f64 values. There is a generic data type associated with a numeric variable: int or real. Generally speaking, this feature isn't used as I've left it to Rust to figure out whether a number is an int or a real. There may yet be a use case where this value will be important – haven't found it yet.

String

A CPL string is a sequence of any legal character including tabs, returns and new lines. The underlying representation of a CPL string is a rust String.

Boolean

A CPL Boolean can be either true or false. The underlying value of CPL Boolean is a Rust `bool`.

Array

An array is sequence of CPL variables. An element of an array can be any data type, including an array, an instantiated struct and, I suppose, dictionary. Access to an array element is via an index expression (e.g. `foo[3]`). Accessing elements in an array which is an element can be done using multi-dimensional indices. For example, `foo[i,j]`.

An index expression may be used in either a lvalue or an rvalue. That is, for example:

```
foo[i,j] = 100;
println foo[i,j];
```

Dictionary

A dictionary consists of label/value pairs. Elements are visited using the label as a key (e.g. `foo["key"]`). The key can be either a string or a number (though internally, numbers are translated into strings). The underlying implementation of a dictionary is a rust HashMap (i.e. `HashMap<String,CplVar>`). The value can be any CPL type including, I suppose, both dictionaries and arrays, but I haven't tested this yet. Accessing a dictionary within a dictionary must be done using the `Type()` built-in function. That is, something like this:

```
foo = dict["key1"];
if Type(foo) == "CplDict"{
   print foo["key2"];                  // print stuff from embedded dict
```

```
}
```

Adding an item to or updating an item in a dictionary can be done like this:

```
foo["new key"] = 100;
```

However, as discussed in Appendix IX accessing and updating a "multi-dimensional" dictionary using an index expression is not supported. That is, the following fail:

```
foo["dim1","dim2"] = 100;
print foo["dim1","dim2"];
```

I don't know that this is a bad thing as the "semantics" of a multi-dimensional dictionaries seem squishy to me anyway. And, it gets particularly hard if you want to access a multi-dimensional *array* that is stored in a dictionary element.

As noted above, it is possible to access a dictionary (or an array) within a dictionary using a multi-step process (creating a variable containing the content of an element). When in doubt, this is probably the most reliable way of handling these mixed type collections.

Note, too, that if you say: `foo["x"] = 100;` and "x" exists within the dictionary, whatever was there will be replaced with 100. The Insert built-in function allows you to prevent updating unconditionally.

Identifiers

Identifiers are strings of characters. For example: `foo`, `bar`, `zot`. They are used to declare functions, structs, parameters, and variables. They can contain letters and digits but must start with a letter. The "_" can be used to separate parts of the identifier for readability. References to struct members include the ':' character separating an instantiated struct name and member name (more on this within the context of structs).

Literals

With some exceptions, a literal value can be used wherever an identifier can be used. The exceptions are: function names, the *lvalue* of assignment statements and the source in a `foreach` statement. The following table describes the syntax of literal values:

Type	**Example**
Numeric	1, 100, 1.4
String	"this is a string"
Boolean	true, false
Array	[1,2,3]
Dictionary	{{"key1",1},{"key2",2}}

Operators

CPL supports arithmetic, comparison, logical, assignment and some special operators.

Arithmetic Operators

CPL supports the following arithmetic operators (which can be applied only to numbers) used in expressions:

- Addition (‘+’)
- Subtraction (‘-‘) as a binary operator
- Numeric Negation (‘-‘) as a unary operator (changes the sign of a number)
- Multiplication (‘*’)
- Division (‘/’)
- Modulo (‘%’)
- Bitwise And (‘&’)
- Bitwise Or (‘|’)

Comparison Operators

The following comparison operators which can work on both numbers and strings:

- Compare equal (‘==’)
- Compare not equal (‘!=‘)
- Compare greater than (‘>’)
- Compare less than (‘<’)
- Compare greater than or equal to (‘>=’)
- Compare less than or equal to (‘<=’)

When comparing strings using inequalities (e.g. ‘<’) if the lengths of the two strings are different, the inequality works on the lengths (i.e. a string of length 4 is < a string length 5). If the lengths of the strings are the same, then it uses the collating sequence of the characters (i.e. “twat” is less than “twit”). Strings are equal only if they have the same length and the exactly the same characters. They are not equal if either of these conditions is not met.

Logical Operators

The following logical operators which only work on Boolean values (and expressions that evaluate to Boolean values):

- Logical OR (Lor) (“||”)
- Logical AND (Land) (“&&”)

Assignment Operators

And the following "assignment” operators:

- Addition (‘+=’)
- Array push (‘+=’)
- Subtraction (‘-=‘) as a binary operator
- Multiplication (‘*=’)

- Division ('/=')
- Modulo ('%=')
- Bitwise And ('&=')
- Bitwise Or ('|=')
- Concatenation (".=")

The assignment operator += is overloaded in 3 additional contexts:

- Array: *`array`* += *`scalar`* appends *`scalar`* to *`array`*
- Array: *`array1`* += *`array2`* adds the elements of *`array2`* to *`array1`*
- String: *`string1`* += *`string2`* appends the value of *`string2`* to *`string1`*

Special / Unary Operators

The following are "special" operators:

- Concatenation ('.') – Strings only (equivalent to "+=" in a string context)
- Boolean NOT ('!') – Booleans
- LengthOf('#") – Strings and Arrays

See Appendix VII for information about how CPL deals with mixed types in expressions.

Code Blocks

IF, WHILE, LOOP, FOREACH and EVAL statements (see *Statements* below) conditionally executes a bunch of statements or executes a bunch of statements repeatedly. I refer to these bunches of statements as "code blocks". Code blocks not only define a bunch of statements to execute, they also define a variable scope. That is, variables created within a code block live only as long as the block and are not visible outside the block.

Functions

All CPL executable code exists within functions. Near as I can tell, you can declare as many functions as you want but there must be one (and only one) prefixed with the keyword *`entry`*. All other functions are prefixed with the *`fn`* keyword. The *`entry`* function is where the interpreter begins executing the CPL program (think of this as *`main`* in 'C');

```
entry <name> [([<entry parameter>])]{<body>}
```

declares the entry function. *`<entry parameter>`* is an identifier which is an array of command line arguments and is optional. The data type of these elements is string (see Appendix VII for a discussion about mixed data types). There can be at most one entry parameter. For example, if you execute a CPL program with arguments a b c, then the <entry parameter> will be an

array with 3 string elements, “a”, “b” and “c”. The parenthesis is not required if the <*entry parameter*> is not present. That is both:

```
entry foo(){...}
```

and

```
entry foo {...}
```

are permitted. The following:

```
fn <name> [([<parameter list>])] { <body> }
```

declares all other functions. I’ll refer to these as *signatures*. You should note that the parenthesis surrounding the parameter list are not required if the list is empty. That is both:

```
foo(){...}
```

and

```
foo{...}
```

are legal function declarations.

You can’t declare a function within a function, i.e. a sub-function (at least I don’t think so) – this isn’t Pascal ☺.

Parameters and Arguments

The parameters specified in the signature are place-holders for actual arguments specified in function calls. That is, a reference to a parameter in a function is actually a reference to the argument that it represents. For the *entry* function, as noted, the argument (and there is only one) is the array containing command line arguments.

By default, scalar arguments are passed “by value” and collections are passed by reference. Scalars are numbers, strings and Boolean values. And, within the context of a called function, a parameter representing scalar argument is a “locally declared” variable.

The following are passed “by reference”:

- Instantiated structs
- Arrays
- Dictionaries

That is, the address of the operand is passed rather than the operand itself. If the called function modifies a parameter passed by reference, it is modifying the same operand as seen by the calling function.

A major implication arising from the ability to pass arguments by reference is that there is only one operand stack in existence for the entire life of a CPL program.

For parameters that are scalars, arrays and dictionaries, the identifier is unqualified (e.g. *fn foo(a,b)*). For a function that intends to accept an instantiated struct, the parameter is qualified (e.g. *foo(a,b:my_struct_definition*). See the section below discussing structs.

Structs

A struct in CPL groups a number of variables into a single collection which can be moved, copied or written to a file as a group. The syntax for declaring a struct is:

```
Struct <name> {
   <member list>
}

<member list> ::= <member name> [<initialization>];
<initialization> ::= '=' <value>
<name> ::= ID
<member name> ::= <ID>
```

The <value> must be a number, string or Boolean. In particular, it may not reference other structs and, since this syntax is occurring outside of a function, is unable to reference any variables. Note that each member entry is terminated by a ";". For example:

```
struct foo{
   mem1;
   mem2 = 100;
}
```

Instantiating a Struct

Structs are instantiated using the *new* keyword. For example:

```
struct foo{
   mem1 = 100;
   mem2 = 200;
}

entry main(){
   local = new foo;
   print local:mem1;
}
```

Access to a struct member is:

```
<local variable>:<member name>
```

Where *<local variable>* is the variable which was used to instantiate the struct.

The output of this example will be 100.

Struct members may be treated exactly like any other variable. For example, the following are all legal statements:

```
foo:mem1 = 100;
foo:mem1 += 1;
foo:mem1 = bar:mem2;
etc.
```

CPL will panic if an uninitialized member is referenced. For example, the following fill result in a run time error:

```
struct foo{
   mem1;
   mem2;
}

entry main(){
   local = new foo;
   print local:mem1;
}
```

Passing a Struct to a Function

When passing an instantiated struct to a function you need to use a qualified id as the parameter that will receive it. For example:

```
struct s_foo{a,b,c}
entry entry{
   foo = new s_foo;
   bar(foo);
}
fn bar(a:s_foo){
   print a:a;
}
```

in other words, the parameter of a function that is meant to receive an instantiated struct specifies the local variable and the name of the declared struct. This syntax is required so that the compiler can construct the symbol table for the instantiated struct in the called function. The underlying cause of this is that variables in the operand stack, at run-time, do not have identifier information. When a function is called with an instantiated struct as a parameter, the compiler does not know this at compile time without the use of the qualified id syntax.

Printing Structs

Consider the following:

```
struct foo{
   mem1 = 100;
   mem2 = 200;
}

entry main(){
   local = new foo;
   print local:mem1;
   print local:mem2;
   print local;
}
```

The output from this program will be:

```
100
200
100,200
```

Similarly, if you use *Fwriteln* as in:

```
Fwrite(out_file, local)
```

The output will be as shown above for *print*.

Loading a Struct from a File

If you want to read a delimiter separated file but be able to reference the content by a variable name you can declare a struct which declares fields in a one-to-one correspondence with the fields in delimiter separated list. The do this:

```
Struct delim_fields{
   Date;
   Time;
   Amt;
}
```

Assuming the delimiter separated file has three fields you can:

```
in_file = Fopen("data.csv", "<,");
delim_data = new delim_fields;

delim_data = Fread(in_file);
while !Feof(in_file){
   print delim_data:Date.” “.delim_data:Time.” “.delim_data:Amt;
   delim_data = Fread(in_file);
}
```

That is, the result of *Fread* is an array and, as it turns out, so is an instantiated struct. Tada!!

Unimplemented Features of Structs

There are loads of unimplemented features as of this version of the code but I thought it would be helpful to describe those features specific to structs which one may think should be there but aren’t. Some of these features might become available in the future and some may never be available:

- Associating a struct with an element of an array. This will probably never be implemented. That is, an array of instantiated structs. For example:

```
struct foo{a; b; c;}
entry main(){
   array=[1,2,3];
   array[0] = new foo;
   array[0]:a = 100;
}
```

- Sub-structs and access to “multi-level” structs. I don’t know if the eventual use case for this feature will be significant enough to add this feature to CPL. The question is, is there a way to do what you need to do without it and and how complicated would that be?

- Struct initialization as part of instantiation. As noted this will probably never get implemented (`a = new foo(10,20);`). But what it does do is suggest a syntactic correspondence between functions and structs.
- Struct members containing an array. I thought this would be kind of a slam dunk but it turns out it's more difficult than I originally thought. One problem arose while building the foreach statement having to do with counting code memory locations.

Statements

CPL supports the following statements:

Literal (or Lit)

A literal assigns a value to an id which can then be used in expressions. For example:

```
entry foo(args){
   lit arg1 = 0;
   lit arg2 = 1;
   println "this is argument 1: ".arg1;
   println "this is argument 2: ".arg2;
}
```

Literal values declared with functions have the same scope and lifetime as any other identifier declared in a function.

However, literal values can be declared outside of any function in which case they are global to the entire program. For example:

```
lit foo1 = 1;
lit foo2 = 2;
entry f1{
   println foo2;
   f2()
}
fn f2(){
   println foo1;
}
```

Literals can be used as arguments in function calls.

Literals are "compile time" creatures. That is, no pseudo machine instructions are generated. Their role is to install information in the symbol table which is then used during code generation. There is a slight (tiny) performance improvement of using a literal in an expression instead of a variable. There is no difference between using a literal land using a hard-coded value.

At the moment the literal expression can only be a number, a string or a Boolean value (true or false).

Assignment

Assignment statements create and update variables. You must declare a variable as the target of an assignment statement before it can be used in an expression.

```
<target> <assignment op> <expression>;
```

<target> is an identifier representing the variable that hold the result of the assignment.

The *<assignment op>* can be "=" or an update operator like "+=". The former will create a new variable or replace the value of an existing variable with a new value. For example:

```
a = 10;
```

assigns the value 10 to the variable a. If a doesn't exist in the current block it is created otherwise its original value is replaced.

The latter cannot create a new variable. Rather it is used to update an existing one. For example:

```
a=10;
a +=10;
```

a is created and assigned the value of 10. It is then updated by adding 10 to its original value. This is equivalent to:

```
a = a+10;
```

See the section on *Expressions* for more information on what can appear on the right side of the assignment operator.

Examples:

```
foo = bar * zot + 1;
bar +=1;
a = function(x);
```

NOTE WELL: Assignment statements must be terminated with ';'.

If/Else

If/Else statements are used to conditionally execute blocks of code.

```
If <expression> { body1 }
If <expression> { body1 } else { body2 }
```

The <expression> must evaluate to either true or false. If true then body1 is executed. If false and the else is not present then nothing is executed. If false and the else is present then body2 is executed.

Eval/When/Otherwise

The *Eval* statement acts like (and is semantically equivalent to) a bunch of nested *if/else* statements.

```
Eval <target>{
   When <expression1> { body1 }
   When <expression2> { body2 }
       :
       :
   Otherwise { other body }
}
```

<target> is a variable or literal which against which the expressions are tested.

Each *<expression>* is evaluated and compared with *<target>*. If the comparison is true then the associated body is executed.

In other words, Eval is a case statement on steroids.

[Note: I chose to implement eval rather than a case (switch) statement since it seemed more useful and obviated the need for elseif syntax and offered programmers a way to avoid a bunch of nested if statements]

Example:

```
eval true{
   when a==1 { do something }
   when a==2 { do something }
   otherwise { do something }
}
```

In this case, the variable "a" used in the expression is defined elsewhere in the CPL program prior to its use in the expressions. The first expression to evaluate to true causes its body to execute and the rest of the when blocks are ignores. If none of the expressions evaluate to true then the otherwise block is executed.

```
eval 100{
   when a { do something }
   when b { do something }
   otherwise { do something }
}
```

In this example, each of the variables a, b, etc. is compared to 100 and the first one that evaluates to 100 causes its body to execute.

While

This is the way to do loops based on a condition:

```
while <expression> { body }
```

for example:

```
i=0;
while i<10 {
   print i;
   i += 1;
}
```

Loop

Rather than implementing a *do while* (or *do until*) I decided to follow the language pattern that Rust uses. The loop statement is infinite so CPL implements early exit via *break* and *return*.

```
loop {body}
```

for example:

```
i=0;
loop{
   print i;
   i+=1;
   if I >= 10{
       break;
   }
}
```

Foreach

Iterates through an array (not a dictionary).

```
foreach <targert> <array> { body }
```

For example:

```
array = [1,2,3];
foreach a array{
   print a;
}
```

Note that the following cases are NOT supported:

```
Case 1:  foreach a [1,2,3]{...}
Case 2:  foreach element dict[key] {...}
Case 3:  foreach element array[index] {...}
```

- Case 1: the iteration source is an array literal
- Case 2: the iteration source is the payload of a dictionary entry that is an array
- Case 3: the iteration source is an element of an array that is an array

Break

Provides an early exit out of a loop. For example:

```
array = [1,2,3];
foreach a array{
   print a;
   if a == 2 {
```

```
            break;
        }
    }
```

In this example, the program prints 1 and 3 but skips 2.

break may include a "depth" expression. For example:

```
entry break_test{
   array1 = ["one","two","three"];
   array2 = ["four2","five2", "Done2","More2"];

   foreach target1 array1{
       foreach target2 array2{
                                    if target2 == "Done2"{
                                       println "Breaking inner on
".target2;
                                       break 1;
                                    }
                                    println "target2=".target2;
       }
       println "target1=".target1;
   }
   println "end of pgm";
}
```

In this example, when *target2* receives the value of *array[2]* ("done2") the the entire program ends because the break depth of 1 indicates that the outer loop should exit. If the depth expression evaluates to a number greater than the number of embedded loops, the program will behave as if that number was equal to the number of embedded loops. There is no error or warning generated.

Continue

Provides a skip to the next iteration of a loop. For example:

```
array = [1,2,3];
foreach a array{
   print a;
   if a == 2 {
       continue;
   }
   print "after continue";
}
```

In this example, the program prints:

```
1
after continue
2
3
after continue
```

because the continue bypasses everything after and starts a new iteration. Use caution when using continue in while loops that are dependent on a counter incremented inside the loop. For example the following code will not work correctly:

```
index = 0;
while index < 10{
   if index == 2{
       continue;
   }
   index += 1;
}
```

Return

```
return <expression>;
```

Exits a function and passes a value to the calling function. For example the following illustrates how to use a function call in an expression (*`print foo(a)`*) and how to return a value to the expression (*`return a*2`*)):

```
entry main(){
   array = [1,2,3];
   foreach a array{
       print foo(a);
   }
}

fn foo(a){
   return a*2;
}
```

Return can also be used to exit a function early. For example:

```
entry main(){
   print foo();
}

fn foo(){
   array = [1,2,3];
   foreach a array{
       print a;
       if a==2 {
                                    return "done";
       }
   }
}
```

Which prints only the first 2 values of the array because of the IF statement:

```
1
2
done
```

If return is used in the entry function, the program exits and, if the expression evaluates to anything but 0, prints that at the end. For example:

```
entry main(){
   array = [1,2,3];
   foreach a array{
       print a;
       if a == 2{
```

```
                                        return a;
        }
    }
}
```

Which prints:

```
1
2
Program exit with: 2
```

Additionally, if *<expression>* evaluates to a number, that number is passed to the operating system shell via the rust function: *std::process::exit(<code>)* where *<code>* is the return value. In most operating systems this code can be examined in order to determine what a shell script containing the CPL program should do next.

Exit

Causes execution of a CPL program to complete prior to the completion of the entry function. The syntax is:

```
<exit statement> := exit [<expression>]
```

Where <expression> passes a value to the interpreter's exit code. If <expression> evaluates to a number, this number is passed to the operating system shell as a "process completion code" which the shell can test. If <expression> evaluates to anything else, the interpreter's exit function displays this as a message. If <expression> is omitted the completion code defaults to 1 (which is treated as an abnormal end by typical shell scripts). You can override this by using:

```
exit 0;
```

Function Call

Executes code residing in another function. It can be either expression term or a statement. The example in the discussion of return illustrates how to use a function call in an expression. Here is how it is used as a statement:

```
entry main(){
   foo();
}

fn foo(){
   array = [1,2,3,a];
   foreach a array{
       print a;
   }
}
```

Printing

There are 2 kinds of print statements: *print* and *eprint* and for each of those there two variations: with a new line and without. *print* and *println* writes data to *stdout*. *eprint* and *eprintln* writes data to stderr.

print & println

These two statements output data to the operating system's *stdout*.

```
print <expression>;
println <expression>;
```

println writes data to *stdout* with a new-line control appended. *print* writes to *stdout* without a new-line.

For example:

```
entry main(){
   array = ["foo","bar",3];
   item = 0;
   foreach a array{
       println "item ".item."=".a;
       item += 1;
   }
}
```

Which prints:

```
item 0=foo
item 1=bar
item 2=3
```

If you changed this to:

```
entry main(){
   array = ["foo","bar",3];
   item = 0;
   foreach a array{
       print " item ".item."=".a;
       item += 1;
   }
}
```

You would see:

```
item 0=foo item 1=bar item 2=3
```

eprint & eprintln

These two statement work exactly like *print* & *println* except output is to the operating system's *stderr*.

Include

This is a simple (read: naïve) mechanism for code reuse. In can appear practically anywhere in the source code but it can make your code confusing if you, for example, insert it into the middle of an expression. When the tokenizer encounters this keyword (prior to any syntax or semantics checking), the include file is opened and from there until the EOF all of the characters read that make up tokens come from this included file. For example:

```
include "allcplcode/cpltests/cpltest_include_struct.cpl"
```

```
entry ENTRY(){
   foo=new s_foo;
   foo:member_a = "hello";
   print "in ENTRY -- foo:member_a=".foo:member_a;
   print foo:member_b;
   fun1(foo);
   print "Back at ENTRY -- foo:member_b=".foo:member_b;
   include "allcplcode/cpltests/cpltest_include_junk.cpl"
   goo = foo:member_a;
   print "goo=".goo;
}

fn fun1(s:s_foo){
   print "at fun1 -- s:member_a=".s:member_a;
   s:member_b .= "---goodbye";
}
```

In this example (which is part of the test suite), the file *cpltest_inlcude_struct.cpl* defines a struct called *s_foo*:

```
struct s_foo{
   member_a;
   member_b="foo.member b";
   member_c="foo.member c";
}
```

And the file *cpltest_include_junk.cpl* contains some random bit of code:

```
include_var = 1000;
print "this var was included: ".include_var;
```

NB: The include statement is NOT (I repeat, NOT) terminated with a ‘;’ (semi-colon). Okay, so it’s all not totally parsimonious; I didn’t feel like including another state in the tokenizer to handle the ‘;’ which means that, if you include it, will be treated as the next character in in the normal flow which means that, if it it syntactically okay to have a semi-colon just standing alone, then terminating the include with a semi-colon will not be a problem. But if it is not syntactically legal to have a stand-alone semi-colon, such as outside any function, then you’ll get a syntax error.

Do you think the keyword should be #include? From a parsing perspective, it doesn’t matter since CPL does not have a pre-processor like C/C++ do.

Expressions

In CPL (and in most everything else in programming language design), an expression is a grammatical structure which is, essentially, a set of instructions for computing a single value. The best way to think about an expression is how you use a calculator. You enter a number followed by an arithmetic operations and then maybe more numbers followed by operations, etc. The steps you took to get the result of a calculation if translated into a sequence of symbols would be an expression. For example on a calculator:

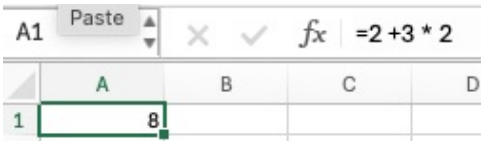
```
Enter 2
```

```
Enter +
Enter 3
Enter *
Enter 2
```

Would produce the value 12. Translating this into symbols you get:

```
2 + 3 * 2
```

Alas, if you were to enter these symbols into, for example, a cell in a spread sheet you would get not 12, but 8.



The reason for this has to do with precedence. Multiplication and division have a higher precedence than addition and subtraction. That is, `3*2` will be executed first and then 2 will be added to that result. The Wikipedia entry:

```
https://en.wikipedia.org/wiki/Order_of_operations
```

does a pretty good job of explaining precedence and how it came to be. Suffice it to say here that I used the precedence defined in the C programming language as a starting point. Things get a little dicey when expressions included things like function calls, literal arrays, etc.

The expression illustrated above is in what's known as "infix". That is the operators are "in the middle" of the operands. It turns out, by converting this to "postfix" the code generator has an easier time of figuring out what opcodes to emit. The postfix version of this expression is:

```
2 3 2 * +
```

And, translating this into opcodes for a stack machine you would get:

```
Push 2
Push 3
Push 2
MUL
ADD
```

The instructions MUL and ADD always operate on the top two elements of the stack. So, the MUL operates on 3 and 2 leaving 6 on the stack. ADD then operates on 2 and 6.

The translation from *infix* to *postfix* is performed in the `itop` module. In the event of a suspected parsing problem, to see how CPL produces the postfix expressions you can add the "-d20" switch which displays the infix and postfix lists for each expression evaluated.

Built-In Functions

A CPL program does most of the real work via the use of built-in functions. There are four categories: I/O, Collections, Strings, Miscellaneous.

I/O Functions

Fopen

Usage:

```
x=Fopen(file_name [, open_mode]);
```

Where:

- *File_name* is the name of the file you want to access
- *Open_mode* is how you want to access the file and a clue as to what might be in the file:
  - "<": opens the file read-only
  - ">": opens the file for write-only; over writes any existing file
  - ">>": opens the file in extend mode; write operations append to existing file
  - "<<delimiter>": opens the file read only AND the file is assumed to be a <delimiter> separated list. See *Fread*/*Freadln* for more on this. The delimiter can be any ASCII character (e.g. "<," specifies a comma separated list and "<|" specifies a list separated by "|" characters.

Returns:

- An index into the *opens* vector or *CplUndefined* if an error occurs

Panics:

- If *open_mode* is unknown

Fread

Usage:

```
Fread(file_handle, array);
```

Where:

- *File_handle* is the thing returned by Fopen
- *array* contains the contents of the entire file. It will be an array of "lines". See NOTE below.

Returns:

- Nothing. *Fread* may not be used as a term in an expression.

Panics:

- If the file not open

- If the *array* argument is not an array

NOTE: This is a little awkward. You must pre-create the array before using it as an argument to this function. To do that you say:

```
array-[];
Fread(file,array);
```

If you don't pre-create the array argument, the parser will complain that you are referencing a variable that has not been added to the symbol table. And, if you use a variable that has been created as a scalar, *Fread* will complain that it can't put the data into it (because it needs an array and *Fread* doesn't know how to turn a scalar into an array).

But, the trade-off is worth it because very large files can be read into memory very quickly.

Also note that *Fread* does will not parse delimited separated files; it ignores the the open mode "<,";

Freadln

Usage:

```
x=Freadln(file_handle);
```

Where:

- *File_handle* is the thing returned by Fopen

Returns:

- If the mode = "<", the next line of the open file or *CplUndefined* if an error occurs or EOF is reached. The call to *Feof* can be used to determine which. The content of the file is assumed to be ASCII.
- If the mode = "<<delimiter>" (e.g. *"<,"*), *Freadln* assumes the line is a <delimiter> separated list and will return an array of items from that list. If the file contains lines *that just happen* to have <delimiters> in them then *Freadln* will assume the commas are separators (*NB: this could really cause you a problem if you don't know ahead of time that the file isn't a csv but has delimiters*). If an item contains <delimiter> but that <delimiter> is "escaped" using the "\" character then it is not treated as a delimiter. If the quote is escaped (e.g. \"), then it is treated as just a regular character. The sequence "\\" means that "\" is part of the element. Also, delimiters inside quoted elements are treated as data and not as delimiters.

Panics:

- If the file is not open

NOTE: When reading a delimited file (e.g. csv), every character (including spaces) between two delimiters and between a delimiter and the end of a line is significant *except* new line characters. For example, if an input line looked like this:

```
A,b    ,  c,d
```

The blanks after "b" until the comma and from the comma until "c" are included in the returned element. The new line following the "d" is not included in the returned element.

Fwrite / Fwriteln

Usage:

```
x=Fwrite(file_handle, var);
x=Fwriteln(file_handle, var);
```

Where:

- *File_handle* is the thing returned by *Fopen*.
- *var* contains the data to write.

Fwrite outputs the var without a new line at the end. *Fwriteln* outputs var with a new line appended.

NOTE: if the file is opened ">" then a new file is created every time. If the file is opened ">>" data is appended to the existing file. In either case, all of the information contained in var is written.

- If var is a scalar, it is converted to a string (unless it already is a string) and then written
- If var is an array, it is written as a comma separated list and, if there is a comma in the element being written the element is written surrounded by quotes.
- If var is an instantiated struct it is treated exactly like an array (a comma separated list)
- Dictionaries are not supported yet and an error will occur if you try to write one.

Returns:

- True if all goes well. But this can be ignored.

Panics:

- If an error occurs while writing (e.g. *var* is a dictionary)
- If the file is not open

Feof

Usage:

```
x=Feof(file_handle);
```

Where:

- *File_handle* is the thing returned by *Fopen*.

Returns:

- True if the last read did not complete because the cursor was at EOF.

Panics:

- If the file not open

Collections

The collections built-in functions operate on arrays and/or dictionaries.

Push

Usage:

```
x=Push(<collection>, <expression>);
```

Where:

- *<collection>* the collection you want to add a value to
- *<expression>* the value (or values) to add.

Returns:

- The new length.

Panics:

- If x is a scaler.

NOTES:

- Push is semantically equivalent to: `x += <expression>` and `x = x + <expression>`. In fact, both of these operations utilize this function. Also, if you don't need the length after the Push you use `x += <expression>.`
- If x is an array and `<expression>` evaluates to an array, then all of the values of `<expression>` are added to x
- If x is a dictionary and `<expression>` evaluates to a dictionary, then all of the key-value pairs in `<expression>` are added to x

Pop

Usage:

```
x=Pop(array);
```

Where:

- `array` the collection you want to add a value to

Returns:

- the popped value was at the end of the array

Panics:

- If x is either a scaler or a dictionary; these data types are not supported in this context.

Contains

Usage:

```
x = Contains(<collection>, <expression>);
```

Where:

- `<dictionary>` is either a dictionary or an array
- `<expression>` the look-up value. For array, the question is, is the index > then the length of the array.

Returns:

- true if `<collection>` contains a key equal to `<expression>`
- false if not

Panics:

- None

Insert

Usage:

```
Insert(<dict>, <key>, <expression>, <update flag>);
```

Where:

- *<dict>* is the dictionary you want to insert something into
- *<key>* the "location" in the dictionary
- *<expression>* the value you want to be at that location
- *<update flag>* allow updates else panic if key exists

Returns:

- Nothing

Panics:

- If *<dict>* is not a dictionary
- If *<key>* already exists and *<update flag>* is false

Delete

Usage:

```
Delete(<collection>, <expression>);
```

Where:

- *<collection>* is the dictionary or array you want to delete something from
- *<expression>* the key value or index for the entry you want to delete

Returns:

- Nothing

Panics:

- If *<collection>* is not a dictionary or an array
- If *<collection>* does not contain *<expression>*

Note: as a rule, unless your CPL program "knows" that the collection contains *<expression>* you should test for this using *Contains()*. Also note that if *<collection>* is an array, the performance can be miserable.

Append

Usage:

```
Append(<collection>, <expression>);
```

Where:

- *<expression>* what you want to append
- *<collection>* is the array or dictionary you want to append something to

Returns:

- Nothing

Panics:

- If *<array>* is not an array or a dictionary
- If *<expression>* is not a collection (i.e. append is for adding arrays to arrays and dictionaries to dictionaries)

NOTES:

- If <collection> is a dictionary then <expression> must be a dictionary
- If <collection> is an array then <expression> may be either an array or a scalar. It may not be a dictionary in this case
- This is semantically equivalent to *<array>+=<expression>*. In other words,
- *+=*, within the context of collections, acts exactly like *Append(x,y)*.

Keys

Usage:

```
x = Keys(<dictionary>);
```

Where:

- *<dictionary>* is the dictionary the keys are in

Returns:

- An array containing all of the keys in *<dictionary>*

Panics:

- If *<dictionary>* is not a dictionary

Note: if *<dictionary>* is empty then x will be empty as well

Sort

Usage:

```
x = Sort(<array>);
```

Where:

- *<array>* is the you want to sort

Returns:

- *<array>*, sorted

Panics:

- If *<array>* is not an array

Strings

Locate

Usage:

```
x=Locate(haystack, lookfor, start);
```

Where:

- *haystack* is a string you want to get a part of
- *lookfor* is a string to locate in *haystack*
- *start* is the location within *haystack* to start looking

Returns:

- the location in the string where *lookfor* starts (a positive number). If *lookfor* is not found then -1 is returned
- returns -2 if *start* is >= length of *haystack*

Panics:

- None

Substr

Usage:

```
x=Substr(haystack, start, length);
```

Where:

- *haystack* is a string you want to get a part of
- *start* is the offset
- *length* is how many characters you want

Returns:

- a new string consisting of the characters from *haystack* starting at *start* for *length* characters

Panics:

- if *start* + *length* > *Length(haystack)*

Regex

Usage:

```
x=Regex(regex);
```

Where:

- *regex* is the regular expression to compile. Note: DO NOT surround the RE with "/" characters as you would for *Replace* or *Split*.

Returns:

- a value that can be used in place of a string in subsequent Match or Capture function calls. It is, actually, just a number which is an index into a table of *compiled regular expressions*.

Panics:

- if the regex is invalid

Match

Usage:

```
x=Match(haystack, needle, start);
```

Where:

- *haystack* is a string containing the things you are looking for
- *needle* is a string consisting of a regular expression for what to find in haystack.; or is the number returned from the Regex function. Note: DO NOT surround the RE with "/" characters as you would for *Replace* or *Split*
- *start* is the location within *haystack* to start looking

Returns:

- an array of strings of the form: *<start>:<end>+1:<item>*. That is, a list of places in the haystack that matches the needle (there can be many). Note you can use Split to get at these fields. You can use *Substr* to extract this information using the result of the split, as in:

```
orig = "this is a foobar at 19:30";
match_list = Match(orig, "19:30", 0);
detail = Split(match_list,":");
fetch = Substr(orig, detail[0], detail[1]-detail[0]);
```

Panics:

- If *start* is >= length of *haystack*

Capture

Usage:

```
x=Capture(haystack, needle, start);
```

Where:

- *haystack* is a string containing the things you are looking for
- *needle* is the regular expression for what to find in haystack; or it is a number returned from the Regex function. Note: DO NOT surround the RE with "/" characters as you would for *Replace* or *Split*.
- *start* is the location within *haystack* to start looking

Returns:

- an array of strings with each element being one of the captured parts of the haystack. For example, if the regex you used or precompiled was:

```
(\w+)(\d+)
```

  There would be 3 items: the zeroth item would be the entire input, item 1 would be all of the characters captured by \w+ and item 2 would be all of the characters captured by \d+.

Panics:

- If *start* is >= length of *haystack*

Split

Usage:

```
x=Split(haystack, delimeter);
```

Where:

- *haystack* is a string containing the things you want to split
- *delimiter* is the expression defining the delimiters in haystack. Note that if the delimiter is surrounded by "/" characters then the delimiter is assumed to be a regular expression as opposed to just a string. See the note for *Replace* for additional information.

Returns:

- an array of substrings

Panics:

- None

Replace

Usage:

```
x=Substr(haystack, needle, newval, start);
```

Where:

- *haystack* is a string you want to get a part of
- *needle* is a string or regex describing what to replace. If you surround the needle expression with "/", *Replace* assumes that what you want to replace is described by a regular expression.
- *newval* is a string you to replace *needle* with
- *start* start looking for *needle* here. This works by removing start characters from the beginning of the haystack before any other look-up activity is performed. While a non-zero value can be used if *needle* is a regex, it is not recommended.

Returns:

- a new string consisting of the original *haystack* with the *needle* replaced

Panics:

- None

Note on the *needle*: if the needle begins and ends with "/" characters then it (less the "/" characters) will be treated as a regular expression. To specify "/" as either the beginning of or ending of the actual needle, that is, to include these characters without the needle being treated as a regular expression, prefix them with "\". For example:

- `x=Replace(haystack, "123", "456", 0)` will look for "123" beginning at character 0 of the haystack and replace all occurrences with "456"
- `x=Replace(haystack, "\/whoops", "/blat", 0)` will replace all occurrences of "/whoops" with "/blat". Note this uses string functions rather than regular expressions.
- `x=Replace(haystack, "/42.\d+/", "42", 0)` replaces any occurrence of "42.<digits>" with "42". The regular expression is "`42.\d+`". There's whole chapter in the Rust book on regular expressions you should look at and, in particular, the replace function. The "needle", between the "/" characters is taken literally and used verbatim, by the Rust regex functions.

Miscellaneous

Length

Usage:

```
x=Length(haystack);
```

Where:

- *haystack* is a variable you want the length of. The content of haystack can be any CPL data type however only strings and arrays make sense. The length of a scalar is always 1. The length of a dictionary is the number of keys which may or may not be useful.

Returns:

- the size of haystack.

Panics:

- None

NOTE: This function is deprecated in favor of the expression term `#<id>` which does the same thing.

Type

Usage:

```
x=Type(var);
```

Where:

- *var* is a variable you want the type of.

Returns:

- a string representing the type of the argument:
  ⇒ "CplNumber"
  ⇒ "CplString"
  ⇒ "CplBool"
  ⇒ "CplArray"
  ⇒ "CplDict"
- if the variable is a reference "&" is appended. For example: "CplString&"

Panics:

- None

Dump

Usage:

Dump("<title>")

Where:

- *<title>* is just a text string.

Returns :

- Nothing (it is a statement)

Panics:

- None

The purpose of the Dump function is to display the content of the operand stack at the point in the execution of the CPL program. It obviously would have little use in an application program but it is useful when the result of an application program is suspicious and I need to understand what's on the stack.

Implementation

In the following sections, I describe various bits of the implementation that seem, to me, to be useful to call out. The details, of course, are in the Rust source code. So I'll start with a description of the development environment and modules used to construct CPL. Then discuss some of the internals and perhaps some of the algorithms used.

Modules

You may want to review how Rust development is structured. I won't bore you with those details except to say that, for my stupid convenience[1], I put all of the source code files in a single directory and pointed the *toml* files at them.

The basic structure is:

```
aa_test_scripts
allsources
allcplcode
   cplpgms
   cpltests
Rust modules
```

[1] I should explain why this is a convenience. It's because I didn't "architect" CPL. That is, I didn't design the structure of the compiler; I grew it. And because it was grown, I had to revisit library code. A lot. Before moving all of the source code into its own directory and before I started naming library files something other than "lib.rs", I couldn't easily tell what source files were open in the IDE (VS Code). So having all of the source code in one place, I could easily hack my way out trouble.

The *allsources* directory contains each of the "lib.rs" files and the "main.rs" file. The Rust modules are a set of directories, each one describing one of the "lib" modules. I won't go into what each of these does but will mention them when it seems to help explain the internals.

The *allcplcode* directory contains a bunch of CPL programs. The *cplpgms* directory contains real programs that I wrote to try to get real work done. The *cpltests* directory contains a bunch of relatively trivial CPL programs which exercise various syntax and semantics of CPL.

The *aa_test_scripts* directory contains "*.sh*" files which invokes all of the CPL programs contained in the *cpltests* directory. To execute these tests, navigate to *cpl* directory and enter:

```
zsh aa_test_scripts/runall.sh
```

For now, the basic flow of CPL is as follows:

- The tokenizer reads the CPL source file and produces an array of Tokens in the order in which they appear.
- The parser reads the array of tokens and, using a kind of state machine, figures out whether the CPL program is syntactically correct. As each correctly formulated syntactic entity is recognized by the parser it adds it to a model. If I was a real compiler writer, this model would probably be some kind of AST but since I'm not and don't know what an AST is, the model is array of syntactic objects (in Rust, *structs*).
- After parsing the CPL program, the *codegen* module reads the model and using whatever hacks I could think of, produces an array of pseudo machine instructions. This is the step where the identifiers used in the CPL program become symbols in the symbol table so that *codegen* , for example, can figure out if the program has properly allocated a variable before it's been used. *codegen*, with the help of the symbol table, is also where variable addresses are computed.
- After all of the pseudo machine instructions have been generated, the exec module is instantiated which looks for the entry function in the pseudo code list and starts interpreting the instructions.
- During execution, function calls creates additional instances of the exec module.

Pseudo Machine

The Rust code that executes a CPL program is a pseudo machine consisting of an instruction processor (i.e. a CPU), Data memory and Code memory. Figure 1 presents a high-level overview of the pseudo machine.

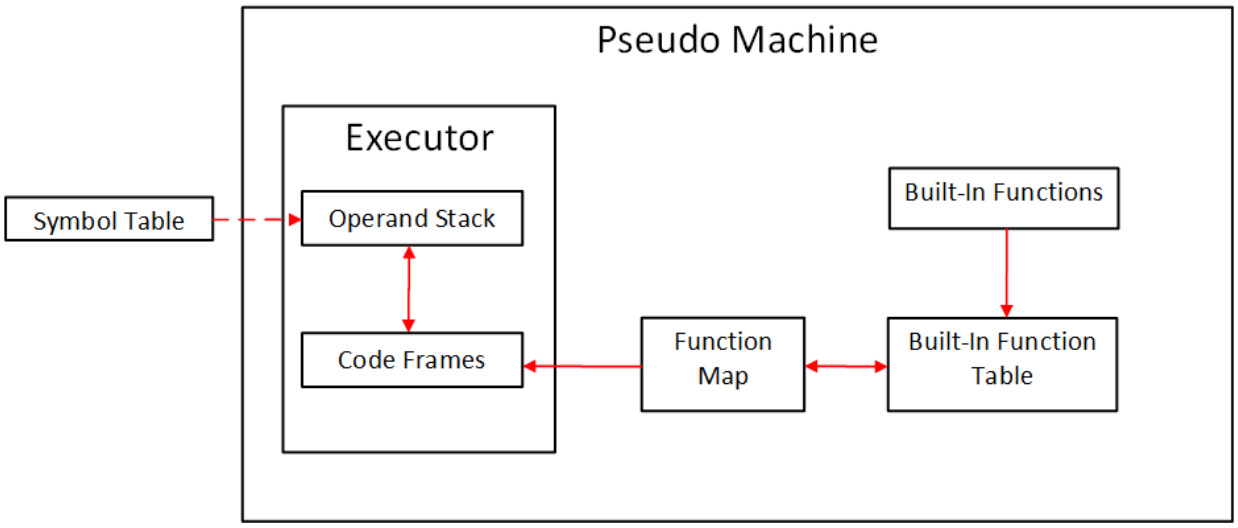


Figure 1 Overview of Pseudo Machine Structure

The executor (`exec()`) consists of two major structures: the operand stack and code frames. The pseudo machine is, essentially, a stack machine and all operations on data take place in the operand stack: variable creation, expression evaluation, argument passing (for built-in functions) and function results. The interpreter itself operates on the code frames structure which contains the opcodes. Function calling is supported via the Function Map which is supported, in turn, by the built-in function table which, in turn, points to the various built-in functions (e.g. `Match`, `Fread`, etc.). Built-in functions are Rust functions which are accessible from within CPL programs. It is relatively easy to add new built-ins functions (look at the `builtins.rs` file starting at approximately line 40. Each entry in the vector there represents a linkage between a CPL program and a built-in function. I show the symbol table here because its structure is intimately related to the structure of the operand stack but is not actively involved during code execution; it exists only for code generation.

Figure 2 shows the structure of data memory along-side of the symbol table to show how symbol table structure mirrors the operand stack.

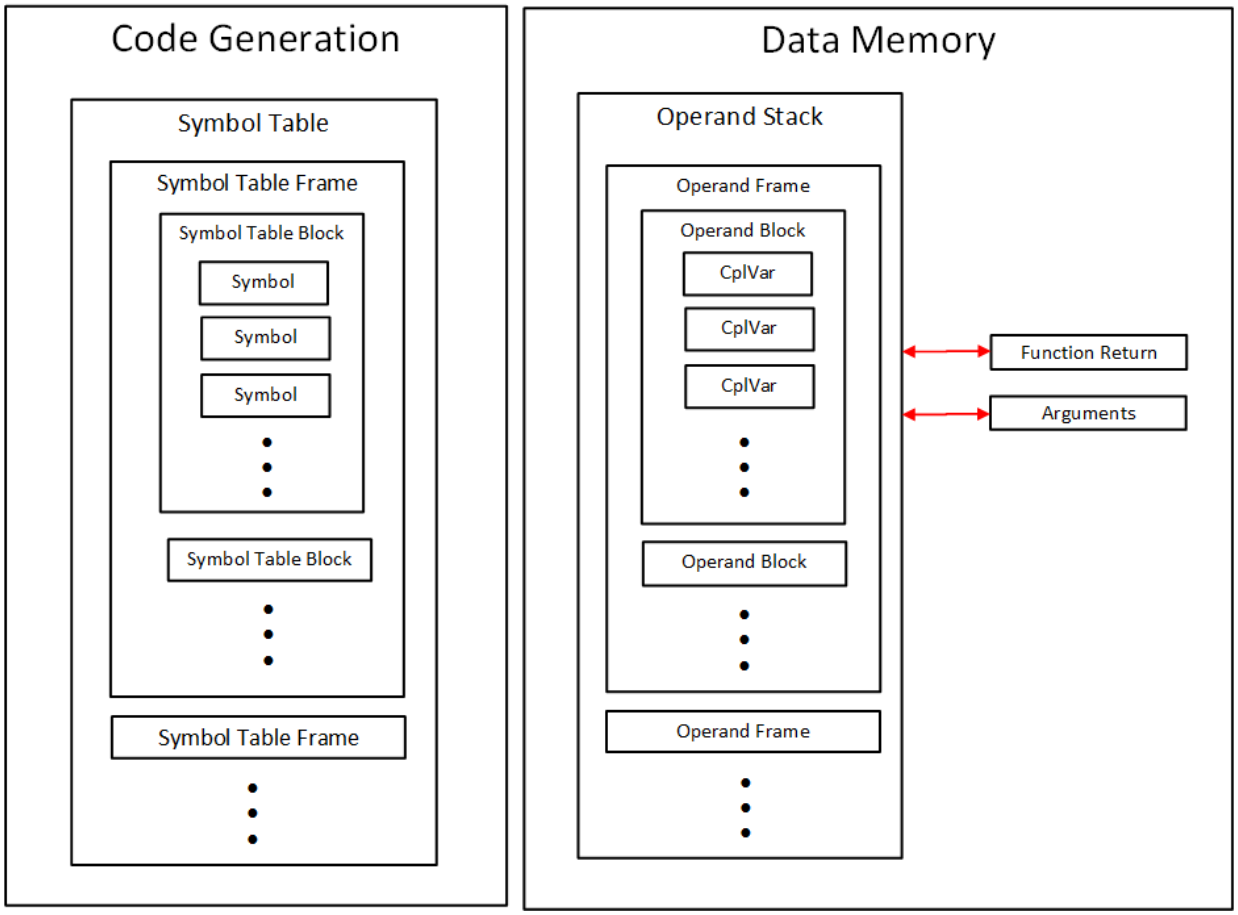


Figure 2 Data Memory Structure

Operand Stack

The primary structure of data memory is the operand stack which consists of two subsidiary structures: frames and blocks. A frame is associated with each active function. That is, if the Entry function calls `function_1` which calls `function_2` then there are three frames: one for the Entry function and one each for `function_1` and `function_2`. All of the variables created during the execution of a function are allocated within its associated frame. When a function returns, its frame is removed from the operand stack (popped).

There is one operand block associated with each code block. Initially there is one block associated with Entry function. Each time a code block is entered due to, for example, an IF statement, a new block is added (pushed). All of the variables created within a code block are allocated in its associated operand block. When a block exits (e.g. at the end of the IF block) its operand block is removed (popped).

There are two *registers* that support function calls and returns by providing a communication mechanism between code frames (see Figure xx):

- Arguments[2]. This is an array of *CplVar* items that the calling function passes to the called function (which, in turn allocates them in the stack according to information provided by the symbol table). Arguments are pushed onto the stack prior to the interpreter executing the function call instruction. Then, the function call instruction pops these items from the stack and adds them to the register. The function call instruction knows nothing about the arguments except how many there are and how many the called routine is expecting. This register may be empty.
- Return Value. A Function always returns a value (a *CplVar*) that is placed in this register. Before stepping to the next instruction, the calling function pushes this value onto the stack.

During code generation (*codegen*) the symbol table maintains variables and their addresses so that when pseudo machine instructions are generated that require variable address (e.g. *alloc*) *codegen* knows what the address is. A symbol table frame comes into existence when *codegen* needs to emit code to set up a function. Symbol table frames exist for the lifetime of *codegen*. A symbol table block comes into existence because the parser has told *codegen* to build a code block. Symbol table blocks are removed (popped) when the parser has told *codegen* to remove a code block. Variables are added to code blocks when the parser has told *codegen* to allocate space for a variable; this happens when the parser recognizes assignment statements and struct instantiations. So, when a symbol (i.e. a variable) is added to a code block its address is:

```
<block-num>,<block-offset>
```

Figure 4 looks inside a symbol table entry. Figure 3 looks inside a *CplVar*.

[2] Theoretically, there isn't a requirement to pass arguments in a register since the frame associated with the calling function is visible to the called function. In fact, when the *CplVar* is of type *CplVarRef*, the interpreter can get the actual values from a different frame. I used this register in a previous version of CPL (when the operand stacks were radically separated) and just never bothered to change the code to allow the called routine to get its arguments directly from the stack. With respect to performance, since CPL never copies collections, there is little or no impact associated with first, moving a variable to the arguments register and then adding it to the local operand stack of the called function.

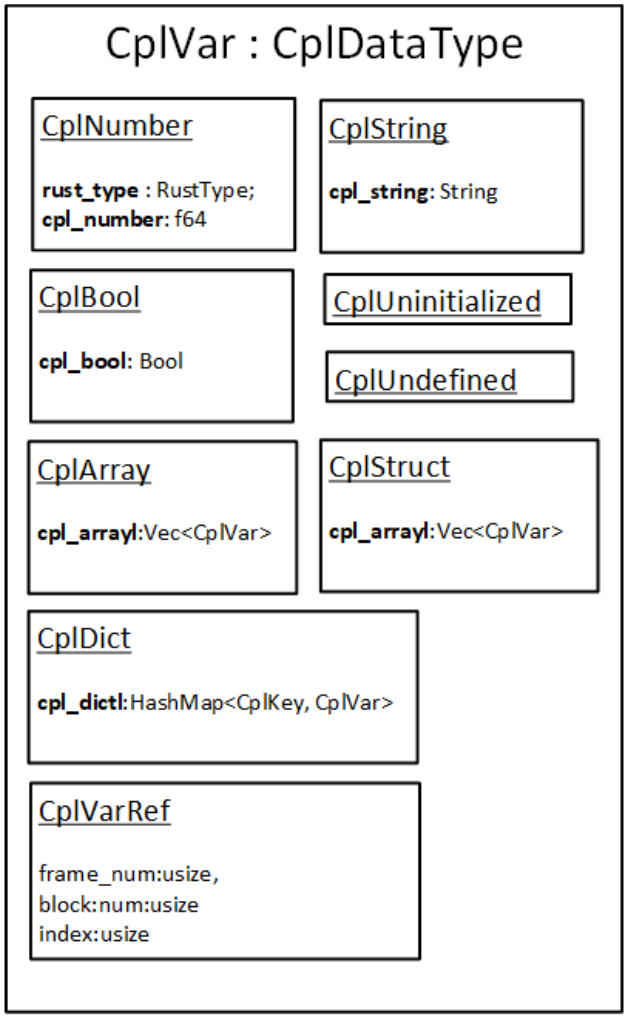


Figure 3 Inside the CplVar

A `CplVar` item is an `enum` called `CplDataType`. Figure 3 shows most of the relevant data types for information stored in the operand stack.

This design takes advantage of a really remarkable feature of Rust that is analogous to a C/C++ union (but a hell of lot safer) and also Java interfaces. You can define an `enum` with content and the content can be any Rust type including a struct. So, for example, here are the first few items in the `CplDataType` enum:

```
    pub enum CplDataType{
CplNumber(CplNumber),
CplString(CplString),
CplBool(CplBool),
    :
    :
   }
```

When a `CplVar` is created its content is created and inserted into the `enum`. When a `CplVar` is fetched, the interpreter can ask, "what `enum` is this?" And from there, it can decide how to handle this data within the context of the particular instruction. The use of this feature obviates

the need to use traits and the concomitant use of downcasting which, evidently, is frowned upon[3] and, in my experience, fraught. The downside is that when the interpreter needs access to the content, it must use the same method as asking what type is it which means that filtering `CplDataType` occurs all over the place – perhaps it's just a fact of life and perhaps there are clever ways to be more intelligent about it.

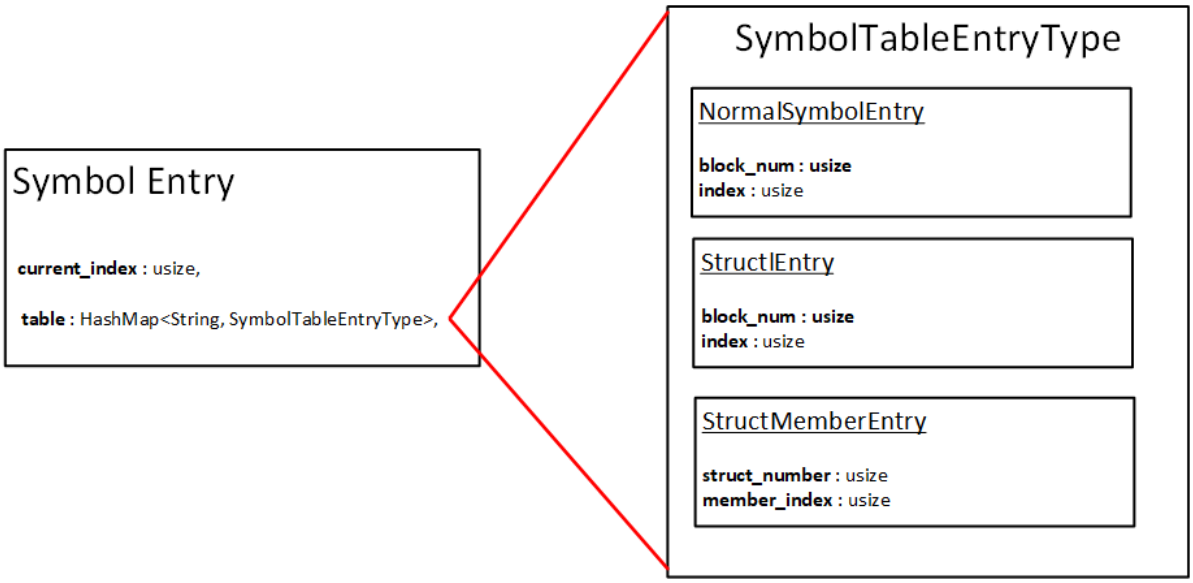


Figure 4 Inside a Symbol Table Entry

A symbol table (for a block) is a Rust HashMap with a String key (symbol name) and an `EntryType`. The `EntryType` is an `enum` and, like the `CplVar` discussed previously, uses the same technique for creating multiple definitions inside the same table. There are three types of symbol table entries: normal, struct and struct member. Each provides a way for `codegen` to emit the right instruction. Normal and struct entries each provide an operand stack block number relative to the current frame, and an index relative to the block number. `StructMember` entries provide an index into an auxiliary table available at execution time called the Struct Table and the index into the array that implements an instantiated struct.

In order to explain a `StructMemberEntry`, I must digress here to explain what a struct is in CPL. *First, and foremost, an instantiated struct is a* `CplArray`. The number of elements in this array is the number of members defined by the struct declaration. For struct instantiation, `codegen` needs to figure out the instructions to emit to create and initialize the array holding the struct members. For struct member access, `codegen` must figure out the instructions to emit to either read or update the element of the array corresponding to the member.

[3] There's an interesting discussion on downcasting here: https://stackoverflow.com/questions/69107401/trait-downcasting.

When *codegen* is instantiated the model is scanned for any struct declarations and, if any were found, they are added to a struct list (*Vec<Struct>*). Then when the struct is instantiated (e.g. *foo = new my_struct*) the following occurs:

- The name of the struct is used to locate the struct declaration in the *struct_list*
- The entry in the *struct_list* links to a list of member names
- Instructions are emitted to create an empty array at the location specified by the symbol table's entry for the *lvalue* of the assignment statement.
- *codegen* then emits instructions to initialize each of the *CplVar* items corresponding to the members of the struct. The initialization value is dependent upon whether or not the struct declaration includes an initial value.
- And for each member, *codegen* emits an instruction to move the initialized member into an array (push).
- And for each member, *codegen* adds a reference to that member in the symbol table using *<lvalue>:<member name>* (e.g. *foo:member_1*).

And thus, we come to the definition of the *StructMemberEntry* in the *StructEntryType enum* which is the point of this little digression. The value of *struct_number* is the index of the declared struct in the *struct_list* and the *member_index* is the index into the array that holds all of the members for the instantiated struct.

When the CPL program then does something like this:

```
foo:member_1 = 300;
```

codegen knows that the *lvalue* is a struct member reference (because the parser told it so) and is able to get the *StructMemberEntry* from the symbol table. From there it can emit instructions to process the expression (*rvalue*) and to update the element in the array corresponding to the struct member. Similarly, when the program does something like this:

```
x = foo:member_1;
```

codegen, during expression processing, knows that the term *foo:member* is a member reference, again, because the parser tells it so, and using the information in the *StructMemberEntry*, knows where to find the member's current value in the array.

Figure 5 shows code memory in more detail.

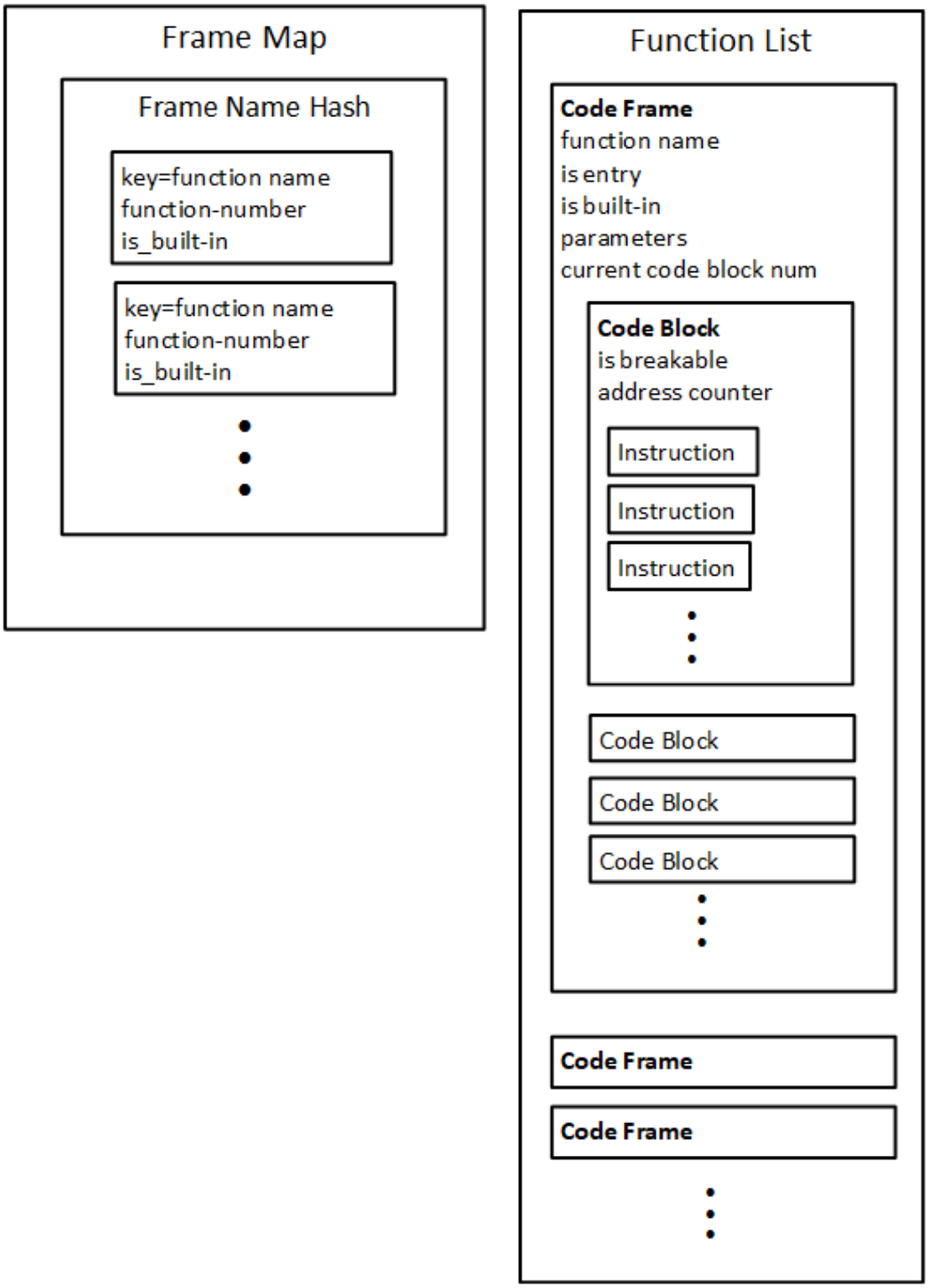


Figure 5 Code Memory

Code Memory

The Frame Map is like a directory of functions. The names of the functions are maintained in a HashMap whose key is the name of the function and whose content points at an item in the Function List. In other words, the Frame Map is your everyday quotidian inversion.

The Function List consists of Code Frames. Each Code Frame corresponds to an entry in the Frame Map. A Code Frame consists of one or more Code Blocks each of which corresponds to blocks of code in a CPL program created by, for example, an IF statement. There is always at least one code block corresponding to the main body of the function. Code blocks consist of the actual pseudo machine instructions which are generated by codegen and interpreted by the pseudo machine instruction processor (sort of like the CPU).

A pseudo machine instruction consists of the following fields:

```
opcode : Opcode,
opcode_mode : OpcodeMode,
function_num : usize,
block_num : usize,
address : usize,
qualifier : Vec<usize>,
literal : Token,
```

Instruction behavior is controlled by: *opcode*, *opcode_mode* and *qualifier*. If an instruction references a specific location in memory the *block_num* and *address* fields indicate where that is and the literal field contains the name of the variable at that location (which is used for diagnostic purposes only).

I don't want to describe all of the opcodes here as that will make this document even more tedious to read than it already is. The list of codes is maintained in an *enum* in the *opcode.rs* file and are, I think, fairly self-explanatory. There are a few, however, for which additional verbiage might be helpful:

- *BlockBegin*: whenever the parser recognizes a new block it signals *codegen* to generate this instruction. It's main job is to push a new operand block onto the stack.
- *BlockEnd*: this undoes what *BlockBegin* does. In particular is pops an operand block off the stack. But it's main job is to return from a code block that was "called" by the *Bl* instruction.
- *Alloc*: This is the instruction that is emitted when the parser recognizes an assignment statement or a struct instantiation. Its primary job is to create an uninitialized variable at a specified location in the stack.
- *Bl* (branch and link): This is a light-weight function call. Every code block recognized by the parser causes the creation of a code block in the Function List item corresponding to the current function. *codegen* emits this transfer of control to the first instruction emitted for the code block. The last instruction executed in a code block is the *BlockEnd* which is analogous to a return (if you trace dump the instructions using the *-d20* switch you may notice in some instances that a block contains multiple *BlockEnd* instructions. This occurs because there may be multiple places where a block can exit.
- *Return*: this instruction is used, not to return from a function, but to move a return value into the Return Value register. A function returns to its caller when there are no more instructions to execute in the top level code block.

- `FunctionCall`: this instruction creates another instance of the exec object (struct) and sets the instruction counter to 0 for that instance. Eventually, instructions are exhausted and the instance simply disappears returning control to the original `FunctionCall`. In effect CPL function call and return utilizes the Rust execution stack.
- `Foreach`: this instruction encodes all of the housekeeping information into a single instruction so that only this is required for: testing an index against the length of an array, exiting the loop when the index is >= the source and keeping the target updated from the array using the index.
- `Push`: You can think of this as analogous to the "load" instruction in other (possibly real) machine architectures. Its job is to push something onto the stack so that can be used in subsequent instructions. It has several "modes":
    - `Lit`: pushes a literal scalar value onto the stack. Lits can be numbers, strings or Boolean values
    - `Var`: pushes a copy of the variable at the address specified in the instruction onto the stack. Note here that if exec figures out that the the "var" to be pushed is a collection it pushes a reference to it, rather than the thing itself.
    - `VarRef`: pushes a reference to the variable at the address specified in the instruction onto the stack. Used when a function call passes a reference to a variable (e.g. `foo(&var)`). The instruction emitted is, for example, `Push &0,0,1 (b)`.
    - `Arg`: pushes an argument from the arguments register onto the stack
    - `Array`: adds an element to the array at the top of the stack. This is used during struct instantiation. You'd think this could be used for array initialization but, alas, it can't for reasons having to do with the relationship between the parser and `codegen`.

Architecture Implementation (Rust)

The pseudo machine (CPU) consists of two primary structs:

```
struct Cpu{...}
struct Executor{...}
```

The CPU consists of:

- A stack of Executor objects, one for each active function
- Runtime statistics repository (used when -d27 or -p switches are used)

After parsing and code generation, the main function instantiates a `Cpu` object and invokes the `run` function. run instantiates an Executor, passing it references to:

- Itself (the Cpu object)
- The frame map
- The arguments register

- The operand stack
- The argument count

And then invokes the exec function which starts executing the instructions from the frame map starting at block 0 of the entry function. The exec function runs until there are no more instructions available in the frame map. As instructions are executed, the runtime statistics counters are updated using the mark_begin and mark_end functions which are methods in the Cpu

Performance Considerations

Initial Thoughts

I was helping a friend find duplicate files on her computer. Over time, and I think we all have done this, we make copies of files and directories ending up with some files being duplicated across multiple directories. I do this when I create versions of things: I'll duplicate a directory and then rename the new one with a date or something. This way I've preserved a snap shot in time in case I need to go back to it.

Anyway, over time, things get out of control and so, I wrote a program in Rust to analyze listings from the OSX `ls` command. I did this rather than having the Rust program access the file metadata directly because my friend's machine is an Intel Mac and mine is an Apple Silicon Mac and I didn't want to go through the bother of installing Rust on her machine and recompiling. She could just run the `ls` command and send me the output.

Essentially the program applied regular expressions to each line of the output to decide whether it was a directory or a file. The result was a hash table keyed by the file name and last modification date with the payload being a list of directories that contain this file. The output was a csv that my friend could load into Excel and use it to figure out where her duplicates were.

The file my friend sent me had over 1 million lines. A lot of these lines I could ignore as they weren't files or directories she was interested in. After scrubbing the data and doing the duplicate analysis, the program discovered approximately 31 thousand files that were duplicated in at least 2 different directories. The program ran in about 32 seconds.

Then, because I'm stupid, I remembered that unless you tell cargo to optimize the code it won't. So I re-ran the program but this time specifying "—release"; the program completed in 1.5 seconds.

So the next step was to rewrite the program in CPL. This is, after all, a song about CPL and not Rust (per se). You'll see this program in the `allcplcode/cplpgms` directory. It is called `dups.cpl`; it is not an exact implementation of the algorithm used in the Rust program but it was similar and it did produce the same result (so at least I got that part right). I suppose the

point of doing this is: could CPL be used for something real and how fast (slow) would it run? I didn't have high hopes since, up to this point, every time I tried do something useful with CPL I found it either lacked some feature or there was a bug getting in the way. But, lo and behold! it worked. Does this mean the CPL has been fully implemented? Probably not, but it appears that I'm close.

Processing the same ls output, `dups.cpl` ran in just under 3 minutes (the statistics for this are shown below).

For grins, I ran this program against a smaller file, containing 200,000 entries. This time it completed in just under 25 seconds. I also ran this same program without the "--release" switch against the larger file. It completed in something like, well, I don't know because after about 10 minutes I just gave up and ctl-c'd out of it.

Line Count	Running Time	Rate
200,000	25 seconds	8K/Second
1.1M	3 minutes	6K/Second

I got curious about where the time was being spent inside CPL's executor (pseudo machine) so I added some counters and a switch to display them at the completion of the CPL program. The command line switch to enable the runtime stats display is "-d27". And, incidentally, if -p<file name> is used, then a CSV file is generated with the raw reduced statistics to enable you, if you are possessed enough, to slice and dice the runtime statistics yourself. The output of this display for the run against the larger file is shown below. The same display for the smaller file was very similar (my hypothesis to explain the difference between the two is that the smaller file had a different mix of directories and files).

The information in the display is as follows:

- The first three columns: opcode, mode and qual identify specific pseudo machine operations. For example, the `FunctionCall` opcode transfers control from one function to the next, the `builtin` mode indicates that the call is to a built-in function and the `qual` indicates which function was called.
- Elapsed is the accumulated elapsed times for each all instances of the code, mode and qual.
- Calls is the number times an instance was executed.
- Average is the average time it takes to execute an instance.
- Percent is how much of the total accumulated elapsed time all of the instances used.

Optimizing the Executor

If I get truly obsessed with this project, I might spend some time reviewing the statistics to determine if there are specific opcodes and modes that, if optimized, could increase the overall performance of the pseudo machine.

Conversions to Inc and Dec

I noticed that there are lot of times when the following statement pattern is used:

```
X += 1;
X -= 1;
```

I modified *codegen* to test for this pattern and converted the statements to *inc* and *dec* respectively. It saved a "push(lit)" which, over a long running loop, does save a noticeable amount of time.

Loop Housekeeping

The foreach statement generated a whole lot of instructions to increment the implied index and keep the target updated from the source via the index:

```
foreach target source{...}
```

All of the information needed to do that is known at code generation time so I figured that a foreach machine instruction could be constructed and implemented so that only a single instruction would be used to do all of this work. Turns out it did save some time in very long running loops: something on the order of 5% to 10%.

So I need to look for other situations where a lot of housekeeping instructions are emitted that could be rolled into a single "kitchen sink" instruction. However, looking at the runtime stats shown below, it's not clear yet where these opportunities exist.

Sample Output from -d27 (Dump Performance Stats) Switch

These lines are from the CPL program itself

```
total lines read from Misc/claire.txt = 1159085
total entries processed = 1159085
total entries analyzed = 1033660
directories= 125425
files duplicated in 2 or more directories = 31296
files skipped = 468717
```

These lines are from stats display (-d27 switch output)

```
Runtime: 1.8mins

*** Internal Runtime Statistics ***

Instruction Count: 270.23M
Total Accumulated Time: 1.2mins
Execution Rate: 3.75M instructions/sec
opcode               | mode     | qual     | elapsed      | calls     | average   | percent
------               | ----     | ----     | -------      | -----     | -------   | -------
FunctionCall         | Builtin  | Capture  | 18.98s       | 908235    | 20.90µs   | 26.3%
FunctionCall         | Builtin  | Match    | 14.91s       | 8870106   | 1.68µs    | 20.6%
Push                 | Var      | Scalar   | 6.43s        | 32349137  | 0.20µs    | 8.9%
Push                 | Var      | VarRef   | 5.88s        | 30827158  | 0.19µs    | 8.1%
Push                 | Lit      |          | 4.23s        | 27682031  | 0.15µs    | 5.8%
Jf                   | Jump     |          | 2.72s        | 21855984  | 0.12µs    | 3.7%
Update               | Update   | Scalar   | 2.57s        | 11018432  | 0.23µs    | 3.5%
LengthOf             | Var      |          | 1.95s        | 20027295  | 0.10µs    | 2.7%
FetchIndexed         |          |          | 1.71s        | 10831699  | 0.16µs    | 2.3%
J                    | Jump     |          | 1.55s        | 20918549  | 0.07µs    | 2.1%
Alloc                | Alloc    |          | 1.47s        | 10893009  | 0.13µs    | 2.0%
BlockBegin           |          |          | 1.33s        | 12376806  | 0.11µs    | 1.8%
```

```
>                  |          |          | 1.10s       | 8926730   | 0.12µs    | 1.5%
BlockEnd           |          |          | 1.13s       | 8815621   | 0.13µs    | 1.5%
<                  |          |          | 1.02s       | 9033244   | 0.11µs    | 1.4%
Bl                 | Bl       |          | 947.71ms    | 11744950  | 0.08µs    | 1.3%
Inc                | Var      |          | 891.50ms    | 11219572  | 0.08µs    | 1.2%
Concat             |          |          | 654.82ms    | 2124734   | 0.31µs    | 0.9%
==                 |          |          | 475.56ms    | 3100980   | 0.15µs    | 0.6%
FunctionCall       | Builtin  | Substr   | 429.78ms    | 1033660   | 0.41µs    | 0.5%
Foreach            |          |          | 280.49ms    | 1215673   | 0.23µs    | 0.3%
FunctionCall       | Builtin  | Insert   | 240.81ms    | 163149    | 1.48µs    | 0.3%
FunctionCall       | Function |          | 286.77ms    | 631855    | 0.45µs    | 0.3%
Push               | Arg      |          | 233.21ms    | 1263710   | 0.18µs    | 0.3%
Continue           |          |          | 184.88ms    | 995948    | 0.18µs    | 0.2%
FunctionCall       | Builtin  | Fread    | 154.37ms    | 1         | 154.37ms  | 0.2%
FunctionCall       | Builtin  | Contains | 87.41ms     | 163137    | 0.54µs    | 0.1%
!                  |          |          | 2.79µs      | 39        | 0.07µs    | 0.0%
AddEq              | Update   |          | 48.82ms     | 137884    | 0.35µs    | 0.0%
Append             |          |          | 22.06ms     | 119181    | 0.18µs    | 0.0%
Return             |          |          | 39.99ms     | 631855    | 0.06µs    | 0.0%
Eprintln           |          |          | 32.38µs     | 6         | 5.40µs    | 0.0%
FunctionCall       | Builtin  | Split    | 38.29ms     | 31296     | 1.22µs    | 0.0%
FunctionCall       | Builtin  | Sort     | 9.18ms      | 1         | 9.18ms    | 0.0%
FunctionCall       | Builtin  | Regex    | 2.05ms      | 38        | 53.87µs   | 0.0%
FunctionCall       | Builtin  | Keys     | 17.08ms     | 1         | 17.08ms   | 0.0%
FunctionCall       | Builtin  | Fwriteln | 13.98ms     | 31296     | 0.45µs    | 0.0%
FunctionCall       | Builtin  | Freadln  | 27.80µs     | 38        | 0.73µs    | 0.0%
FunctionCall       | Builtin  | Fopen    | 401.33µs    | 3         | 133.78µs  | 0.0%
FunctionCall       | Builtin  | Feof     | 8.83µs      | 39        | 0.23µs    | 0.0%
Pop                |          |          | 16.09ms     | 194449    | 0.08µs    | 0.0%
PushNewCollection  | Dict     |          | 0.79µs      | 2         | 0.40µs    | 0.0%
PushNewCollection  | Array    |          | 7.60ms      | 87886     | 0.09µs    | 0.0%
carl@McTreehouse cpl %
```

Appendix I. Syntax Oddities

There are some subtle syntax differences between CPL and other C-Like languages that we should note up front.

Predicates are not Parenthetic

The grammar for logical expressions in IF and WHILE statements do not need to be surrounded by parenthesis. So instead of:

```
if (foo > 10) { ... }
```

CPL would do it this way:

```
if foo > 10 { ... }
```

Foreach Lacks Syntactical Sugar

Where you've probably seen something like:

```
for x in y{...}
```

CPL does away with the "in". It is simply:

```
foreach x y{ ... }
```

Variable Life Times are Local

CPL is more like C/C++ in that a variable lives only as long as the block within which it is declared lives. Thus a variable constructed inside and IF/ELSE block would not be still available after the bock ends. And, as you would expect, variables declared in a "parent" block are visible in the "children".

Increment/Decrement not supported

If this were a proper "C-Like" language then the operators ++ and – would be supported. But alas they are not. Well, not exactly. The executor will execute them but the parser doesn't recognize them. `codegen` uses the ++ internally for the `foreach` statement and codgen replaces statements of the form `<id>+=1` and `<id>-=1` with `<id>++` and `<id>--` respectively.

Perhaps someday I'll revisit this and let the parser recognize them. The problem I ran into was when they are used in expressions like `while i++ > 10` versus `while ++i > 10`. The rule is: post increment (or decrement) means use the value of the variable before changing it and pre increment (or decrement) means use the value of the variable after changing it. My expression processor just wasn't up to it.

Appendix II. Rust Command Line Switches

If you run CPL with "-h" CPL will show you all command line options:

```
WELCOME TO CARL'S PROGRAMMING LANGAUGE (CPL)
```

```
cpl <source> <switches> [<arguments>]

    <source>      ::= file containing CPL source code
    <switches>    ::= [-<sw>[<sw_parameter>]]

         <sw>     := 'd<debug bit>[+<debug bit>]' (debug)
                   | 'w' (Warn runtime errors)
                   | 'h' (help/usage))
                   | 'o'<file> (output file)
                   | 'p'<file> (performance stats csv file)

   <arguments>    ::= arguments passed to ENTRY function

         -d values:
            TRACE_TOKENIZER = 0
            TRACE_PARSER_NEXT_TOKEN = 1
            TRACE_PARSER_STATES = 2
            TRACE_PARSE_LOOP = 3
            TRACE_MODEL_ADD = 4
            TRACE_INFIX_TO_POSTFIX = 5
            TRACE_INFIX_TO_POSTFIX_DRIVER= 6
            TRACE_CODE_GEN = 7
            TRACE_CODE_GEN_ADD_INSTRUCTION = 8
            TRACE_EXEC = 9
            TRACE_EXEC_DISPATCHER= 10
            TRACE_STATEMENT_ADD = 11

            DISPLAY_RUNTIME = 12

            DUMP_POSTFIX_EXPRESSION = 20
            DUMP_OPERATOR_STACK = 21
            DUMP_GEN_CODE = 22
            DUMP_SYMBOL_TABLE = 23
            DUMP_OPERANDS = 24
            DUMP_OPERANDS_DISPATCH = 25
            DUMP_STRUCTS = 26
            DUMP_PERFORMANCE_STATS = 27

            SET_BACKTRACE = 30
            INSERT_DIAG_COMMENTS = 31


         -d values can be 'added' together to specify multiple
            switches; e.g. -d1+4 will set bits 1 and 4
         -w switch tells the executor to print a warning message
            instead of abend if possible


         NOTE: the source file may appear anywhere in the argument
         list but must be the first non-switch

         <switches can appear anywhere on the command line

         NOTE: no space between switch name and its parameter (e.g.
         use -ojunk instead of -o junk)
```

Appendix III. Thinking About Regular Expressions

In my experience, for what that's worth, the two most important concepts when hacking up text are dictionaries and regular expressions. Apart from all the other amazing hacks available using Perl, these two things made Perl really good at it. So, dictionaries have been

implemented (or at least most of it). Perl made regular expressions part of the language. For simplicity, I used built-ins:

```
x = Match(haystack, needle);
x = Replace(haystack, needle, new_value);
x = Split(haystack, delimiter);
```

Match accepts a string or array or a dictionary and asks: does this var (*haystack*) contain *needle*. *Needle* is a string. The result is an array of locations in *haystack* that match *needle*. If the array is empty (i.e. *Length(x) == 0*) then no match was found. If *haystack* is a collection then the returned array is a list of elements that had at least one match. The CPL program can then get the sub expressions in each of the elements by applying the *needle* to each of the resulting elements.

Haystack can be by value.

Match will panic if *haystack* is not a string or, in the case of a collection, any element of the collection is not a string.

Replace replaces, in *haystack*, all occurrences of *needle* with *new_value*. *needle* is a regular expression that exposes substrings to be replaced through the use of groups (named or not). *Replace* returns an array containing the locations in haystack that were replaced.

Appendix IV. CPL Data Types

CPL is a "typeless" language. That is, variables do not have a declared type when they are created. The basic data type, then is something called *CplVar*. The definition of *CplVar* is:

```
pub struct CplVar{
   pub var : CplDataType,
}
```

where *CplDataType* is an *enum* whose variants contain instantiated structs which define each of the actual data types that *CplVar* can host. Some examples are shown in the Pseudo Machine section.

The technique for determining the underlying data is essentially the same. Assume a Rust variable, called foo, was bound to a *CplVar*, the following Rust code will determine what's in it:

```
Match foo.var{
   CplDataType::CplNumber(n) => { do something with n }
   CplDataType::CplString(s) => { do something with s }
       :
   _ => { if not what we're looking for panic! }
}
```

Alas, you see code like this everywhere as it's kind of an inevitable consequence of how *enums* like this work. You will see variants of this used to access functions in the implementation blocks of the various underlying data types. This is one of the trickier aspects of Rust because

of borrowing. You will probably look at this code and laugh at my naivete as a real Rust programmer would know how to simplify this sort of idiom. Mine is an idiotic idiom.

Appendix V. Thinking About Structs

Please review the section describing how structs are specified and accessed in CPL program in the section above: *Structs*. I'll start with this: *a struct is just an array whose elements are named*. This suggests treating a struct like a dictionary. But we don't want or need to do that because at execution time we don't want to access the data in a struct member by its name; we want to access it by its offset. So at exec time, *there is no difference between an array and a struct*. Except, of course, you can't add more elements to a struct at run time (or, rather, if you tried the program will crash).

To do this, `codegen` must translate the specifications of a struct into an array and the symbol table must support translating a struct member into the array offset.

Model Support

The model will have the following functions:

```
add_struct
add_struct_member
```

This will work, more or less, like adding an IF/ELSE. `add_struct` returns `struct_model context` (i.e. it's location it the statement list) and `add_struct_member` will update the struct using the `struct_model context`.

Symbol Table Support

Within the context of struct declaration, the symbol table is not used. Instead a bespoke hash table called `struct_map` is used to capture:

- The index in the model struct list for the specific struct being declared
- The index within the synthesized array that implements a struct.

At the start of `codegen`, all of the structs and their members are added to `struct_map`.

Within the context of instantiation, all of the information contained, related to the struct being instantiated, is copied (more or less) into the symbol table. As noted previously, an entry in the symbol table consists of a key (String) and a value. The type of the value is an Enum, `SymbolTableEntryType` of which there are two values `NormalSymbolEntry` and `StructMemberEntry`. The latter is used for struct members, the former for everything else and, in particular, the array that implements the instantiated struct.

Instantiation

Recall that a struct is instantiated by assigning it to a variable:

```
struct foo{a; b; c;}
entry main(){
   a = new foo;
}
```

We use the new keyword here to differentiate a struct from a local variable which might have the same name and, alas, it is a bit of a syntactic que to help *codegen* figure out what pseudo code to emit.

Because the parser has determined that an assignment statement is being declared, *codegen* starts off by asking two questions: is the *target_expression_list* length > 0? If so then *codegen* knows it must emit code to update an array element whose index is specified by the *target_expression_list*. The second question is, does the name of the target include a ':'? If so, *codegen* knows that a member of a struct is being updated. If neither of these questions is answered in the affirmative then *codegen* emits code that will update a scalar value.

If *codegen* is building an assignment statement that updates the member of a struct, it has to first get the address of the instantiated struct from the symbol table (this came from the *a=new foo* statement). Offsets with the struct array of the individual members of a struct are kept in the symbol table keyed by *<instantiated struct name>:<member name>*.

Passing Instantiated Structs to a Function

And, herein lies the problem that could collapse the entire house of cards out of which Structs are built and which highlights a performance issue which really needs addressing. Please look at Appendix VI for an explanation of the performance issue.

Using an instantiated struct as an argument to a function is, more or less, the same thing as using an array. The problem here is that the called function has no way of knowing that it's being passed an instantiated struct and thus can access its members only as items in an array. To fix this problem, if a CPL function wants to accept a struct as a parameter, the syntax is:

```
//  this is the definition of a struct
struct foo{
   var1 = 200;
   var2 = 100;
}

entry main(){
   local = new foo;
   bar(local);                                    // call bar with
instantiated struct
}

// the parmeter expects to receive an instantiation of
// the struct foo
fn bar(x:foo){
   print x:var1;
}
```

Appendix VI. Passing By Reference

In Appendix V I alluded to a performance issue associated with using arrays as arguments in function calls. Suppose a CPL program read 50,000 lines of text into an array. Then, for some reason, needed to pass that array to a function. If arrays are passed "by value" then the content of the array would be copied to the top of the arguments register and then to the top of the operand stack. That's 100,000 moves. Now, suppose, for some reason, the CPL program

called the function in a loop with, say, 50,000 iterations. The arithmetic hurts my head and the performance of the CPL program would be hideous.

The way to deal with this is for CPL to pass collections by reference.

Consider the following CPL program (*cpltest_pass_by_reference*):

```
entry foo{
   array = [1,2,3];
   b = 5;
   bar(array,5);
   print array;
}

fn bar(a,b){
   i=0;
   while i<Length(a){
       a[i] += b;
       i+=1;
   }
}
```

And here are the emitted instructions (this'll be a bit tedious so I'll annotate as best as I can):

```
Generated Code for Function foo
Block Number: 0
    0: BlockBegin
    1: Alloc %0,0,0 (array)

// create a new array and add values 1,2,3 to it
    2: PushNewCollection [0,0,0,0 ()
    3: Push $0,0,0,0(1)
    4: Update [0,0,0,0 ()
    5: Push $0,0,0,0(2)
    6: Update [0,0,0,0 ()
    7: Push $0,0,0,0(3)
    8: Update [0,0,0,0 ()
    9: Update #0,0,0 (array)

// create a variable b and set it to 5
    10: Alloc %0,0,1 (b)
    11: Push $0,0,0,0(5)
    12: Update #0,0,1 (b)

// push a reference to the array onto the stack
    13: Push @0,0,0 (array)

// push a literal number 5 onto the stack
    14: Push $0,0,0,0(5)

// call function "bar", passing 2 parameters:  reference
// to array and literal 5
    15: FunctionCall (bar) arg count=2 is_statement=1
    16: Pop

// print the new values of the array
    17: Push @0,0,0 (array)
    18: Print
    19: Push $0,0,0,0("$$Synthetic$$")
    20: Return
Generated Code for Function bar
```

```
Block Number: 0
    0: BlockBegin

// grab arguments (ref to array and literal 5) from arguments
// register
    1: Push ^1,0,0
    2: Push ^1,0,1

// create index variable I and set it to 0
    3: Alloc %1,0,2 (i)
    4: Push $1,0,0,0(0)
    5: Update #1,0,2 (i)

// compare the index to the Length of the array and if
// the loop is at the end exit the function otherwise
// do what's in the if block
    6: Push @1,0,2 (i)
    7: Push @1,0,0 (a)
    8: FunctionCall (Length) arg count=1 is_statement=0
    9: <
    10: Jf *0,13
    11: Bl rtn=0:12 targ=1
    12: J *0,6
    13: Push $1,0,0,0("$$Synthetic$$")
    14: Return
Block Number: 1
// We're inside the IF block here
    0: BlockBegin

// get the index and the new value
    1: Push @1,0,2 (i)
    2: Push @1,0,1 (b)

// Key to the whole shootin match:  +=
    3: AddEq ##1,0,0 (a)

// increment the index and return to the length comparison
    4: Push $1,0,0,0(1)
    5: AddEq #1,0,2 (i)
    6: BlockEnd
```

I'll draw your attention to two instructions:

- **13 in block 0 of foo**. This push instruction figures out that what it has to push onto the stack is an array and, instead of pushing the entire array onto the stack, creates a reference variable, containing the absolute address (frame, block and address) of the array.
- **3 in block 1 of bar**. The "##" is the instruction's *opcode_mode* and means "operate on an element of an array" and when *exec* does this it can figure out that the value at address 0,0 is actually a reference and does the dereferencing necessary to update the element at index "i".

Appendix VII. Mixed Type Expressions

CPL takes some unnatural liberties with mixed type expressions. For example, consider the following:

```
a = 10;
b = "10";
```

note the results of the following expressions:

```
a+b = 20
a==b = true
```

In other words, if an operator is *usually thought of as* an arithmetic operator and the operands are mixed numbers and strings, then CPL will attempt to convert the string into a number. If it can't then the operation will fail or, in the case of the comparison, the result will be false.

And, consider this:

```
a = "10";
b = "10";
```

note the results of the following expression:

```
a+b = 20
```

In other words, if the operator is *usually thought of as* and arithmetic operator and the operands are both strings, the CPL attempts to convert both strings to numbers and then perform the operation. If the either string is unable to be converted, then CPL will treat the operation as a string operation but only ".", concatenation, can be used. That is:

```
b="10"
a="x10";
```

`a+b` will fail because "x10" can't be converted into a number.

In these cases "failure" means a Rust panic! which will stop execution of the CPL program.

There are some cases where an operator is overloaded. For example:

```
a=[1,2,3];
a+=4;
print a;
```

The result will be:

```
[1,2,3,4]
```

In other words, when the lvalue is an array, the "+=" acts like "push".

Appendix VIII Module Notes: Hacking My Way to Hell

In this section, I discuss some specific "features" (now there is a polite word for it) of the modules and other parts of the CPL structure.

Symbol Table and Names

I recently learned about using Symbol Interners. The idea is that you can put all of the symbols you care about into a table and whenever you need to reference that symbol you can use the

index of the that table. In particular, it's a way of providing a connection between a symbol and how it's being used in the executor (module: `exec`) without having the symbol table a part of the executor itself.

The symbol table (module: `symboltable`) is responsible for, in effect, modelling a runtime operand stack. Every symbol that is added to the symbol table is given an "address" which consists of a block number and offset within that block (i.e. address). When `codegen` emits an instruction that has to know the address of a variable it consults the symbol table.

Every symbol is also added to the "names" list.

These instructions (e.g. `Alloc`, `Push/Arg`, etc.) include the index of the variables names so that, in particular, when the instruction adds a variable to the operand stack, the item in that stack not only has a block number and address, it also has the variable's name (or, rather, its index in the names list).

Module: `cplvar`

This is one of those modules that you never like to have because it's too big. It defines the operand stack and all of the data types assigned to a `CplVar`. As noted above, one of the properties of a `CplVar` is a reference to a name. Again, as noted, when the executor, acting on behalf of a pseudo machine instruction, creates a new variable, the name index added to the instruction by `codegen`, and is propagated to the `CplVar` itself.

This is a debugging aid. With the reference to the name list, I can dump the contents of the operand stack (see the `Dump` built-in function) and see the name of the variable along with its address.

It is not straight forward. What I first tried to do was create the Names struct and simply pass it to all of the various modules who needed to create or use a name. Inevitably I hit a wall because of the borrow checker: adding a reference to the Names struct to the Exec and `OperandStack` constructors blew things up. So here, I was able to simply pass the array containing the names so that if either of these modules needed the actual name it would have access to it via the index. The important objective was to avoid having to pass a copy of the name list to the executor because that would impact the performance of function calls. So the executor got a reference to the vector containing the names.

Appendix IX. Rethinking Dictionaries

I've been able to implement both structs and multi-dimensional arrays as lvalues. That is, for example:

```
foo:bar:zot = 100;
foo[1,2] = 100;
```

and I've been able to implement a single dimension dictionary as a leval: for example:

```
foo["this is a key"] = 100;
```

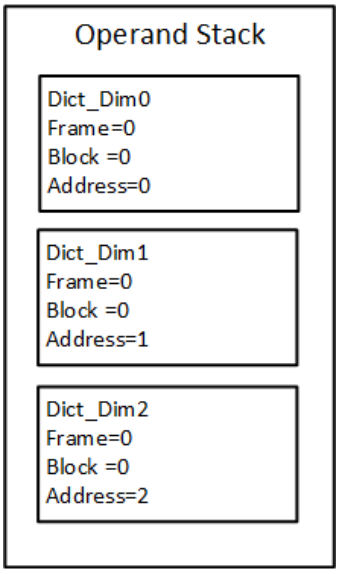
But I hit a wall when I tried to implement a multi-dimensional dictionary, e.g:

```
Foo["key1","key2"] = 100;
```

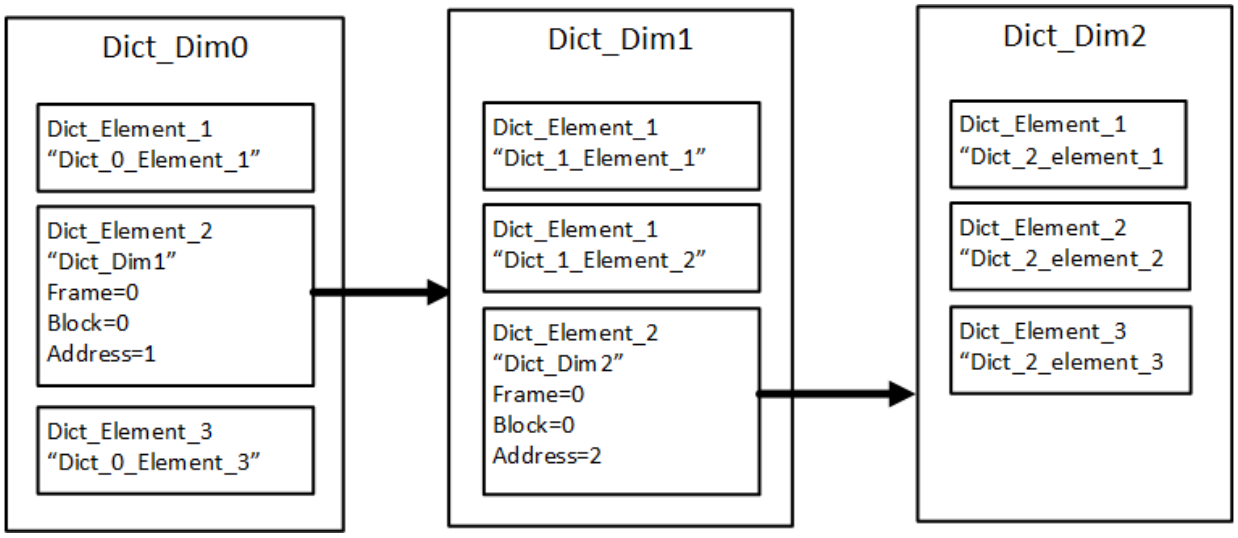
I kept running into the dreaded: "you can only have one mutable reference of self" error. This arose because I had to visit, mutably, the dictionary recursively.

There is a solution but I think I have to re-think how dictionaries are implemented.

I think the operand stack would look like this:



When a dictionary is instantiated in the operand stack, additional dimensions become individually addressable variables (each with its own address). That is, dimension 0 is at address 0, dimension 1 is at address 1, and so on. How does the code find these dimensions? Here's an attempt:



So we're still implementing dimensions as "dictionary within a dictionary" except that we don't *insert* the dimension into an element of its "parent" we insert the *address* of the dimension.

This will solve the multiple mutability problem because we don't have to mutability borrow an element of the parent; instead we address the dimension via its address.

Appendix X. Thinking About Futures

In this appendix, I muse about future enhancements and optimizations. Warning: you shouldn't take much (or any of it, for that matter) seriously. It's just a place for me to jot down some thoughts that wake me up at 3AM.

Generalized Access to the Operand Stack

In the executor, there are a gazillion points where access to the operand stack is required and I think it would better to have some kind of generalized macro for doing that. This macro would need to encode all of the use cases emerging from the many different opcodes the executor has implemented.

Here is the list that I've been able to determine:

- Fetch a scalar
- Fetch an element of an array
- Fetch an entry in a dictionary
- Update a scalar
- Update an element of an array
- Update an entry in a dictionary
- Add a new scalar
- Add a new Array
- Add a new Dictionary
- Pop a variable
- Push a variable

The addressing modes appear to be:

- Top of Stack (TOS)
- Local (via block number and address)
- Global (via frame number, block number and address)
- Indirect (via a CplVarRef)

Setting aside the TOS mode, a variable is located via frame, offset within the frame (block) and and offset within the block (address).

<end this madness here>

I've spent hours reading about macros and I still haven't gotten my head around them. It is probably not impossible to do something like this but it's way beyond my current level of understanding.

Extend Literal Syntax

Support the use of array literals, dictionary literals and any arbitrary expression whose terms are literals. For example:

```
Lit secs_per_hour = 60;
Lit hours_per_day = 24;
Lit secs_per_day = secs_per_hour * hours_per_day;
```

Essentially, this is evaluating a postfix expressions in *codegen* instead of the *executor*.

Multi-Level Structs.

A use case for multi-level structs which may make it easier to justify more thinking about it is XML.

An XML document is, more or less, equivalent to a struct:

```
<tag1>
   <subtag1>some data</subtag1>
   <subtag2>some more data</subtag2>
</tag1>
<tag2>even more data</tag2>
```

Is:

```
struct xml_exmple{
   struct tag1{
       subtag1 = "some data";
       subtag2 = "some more data";
   }
   tag2 = "even more data";
}
```

And it would be analogous in JSON.

So I think what I'm really describing here is built-in support for mapping XML to CPL and CPL to XML; in other words serializing an deserializing CPL structs.

There are, evidently, two forms: named sub-structs and struct references. For example:

```
struct multi{
   struct level1_a{
       sublevel1 = "some data";
       sublevel2 = "some more data";
   }
   level1_b = "even more data";
}
```

and

```
struct multi{
   foo = "some data"
```

```
   zot = mutli_2;
   bar = "some more data"
}

struct multi_2{
   subzot = "even more data";
}
```

My inclination is to prioritize name sub-structs over struct references since it looks more like what I need for XML. So I won't discuss the latter any more here.

Access to a multi-level struct would be (using the example above):

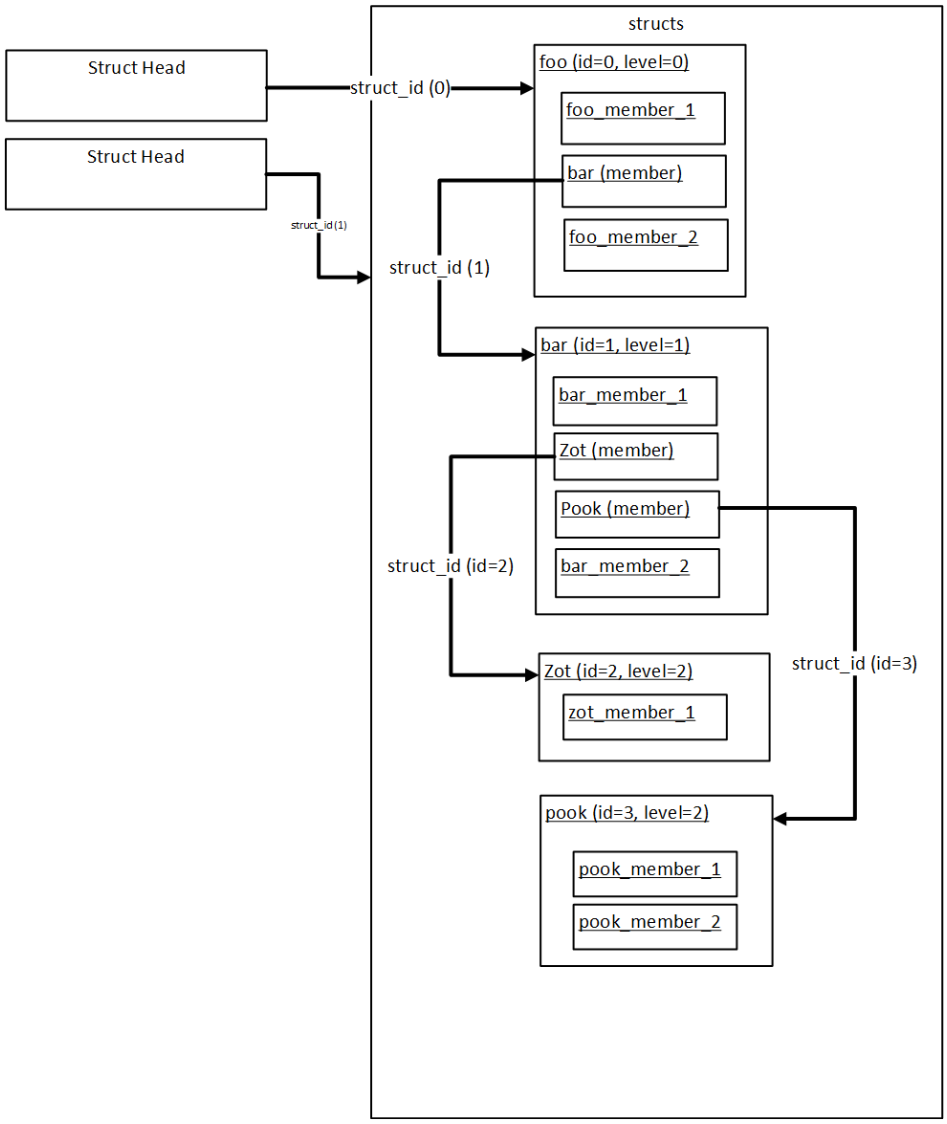
```
foo = new multi;
print foo:level1_a:sublevel1;
```

(which would print: "some data").

Recognizing Multi-level Structs

Thinking a bit more, here is another example of a multi-level struct with a diagram:

```
struct foo{
   foo_member_1="foo:member_1";
   struct bar{
       bar_member_1="bar:member_1";
       struct zot{
           zot_member_1="zot:member_1";
       }
       struct pook{
           pook_member_1 = "pook:member_1";
           pook_member_2 = "pook:member_2";
       }
       bar_member_2="bar:member_2";
   }
   struct baz{
       baz_member_1="bas:member_1";
   }
   foo_member_2="foo.member_2";
}
```



There are two attributes in the *Program* struct: *struct_headers* and *structs*. *struct_headers* contains a list of indices; each one a "pointer" into *structs*. *structs* is a vector of type *Struct* (defined in *structmodel.rs*). When the parser recognizes a top-level struct name (e.g. *struct foo*), a *Struct* object is added to *structs* and its location in *structs* is added to *struct_headers*.

When the parser recognizes a field (e.g. *foo_member_1*) it follows the index captured when the struct was first recognized and added and adds a *Field* object to the member list.

When the parser recognizes a sub-struct (e.g. *struct bar*) it adds a Struct object to *structs* and adds a Substruct object to the parent struct's member list.

Instantiating Multi-Level Structs

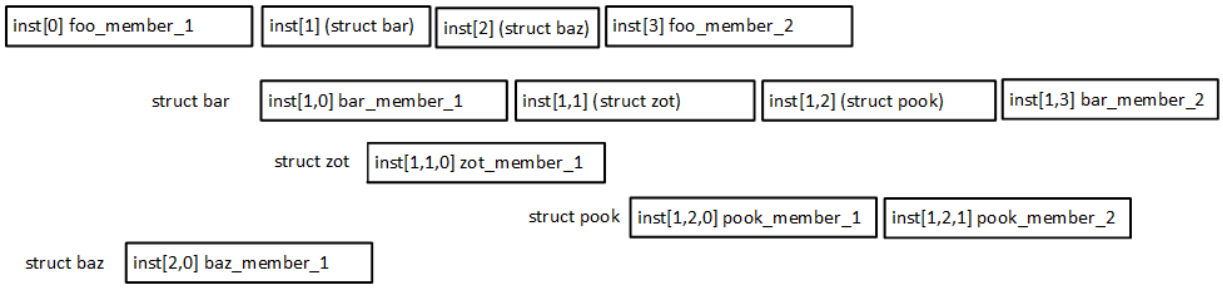
The goal, ultimately, is to be able to resolve a struct reference into an element of the underlying array that implements a struct. Sub-structs are elements of the underlying array which are, themselves, arrays.

```
entry multi_level{
   inst = new foo;
   println inst:bar:zot:zot_member_1;
}
```

Using the struct defined above, the following program will (or at least, should) print:

```
zot:member_1
```

The implemented struct will look like this (sort of):



The array inst has 4 elements. The first and last are members. The middle 2 are arrays: bar and baz. That is:

- inst[0] is a field (foo_member_1)
- inst[1] is a sub-struct (bar)
- inst[2] is a sub-struct (baz)
- inst[3] is a field (foo_member_2)

The array in inst[1] has 4 elements whose indices are 1,0; 1,1; 1,2; and 1,3:

- inst[1,0] is a field (bar_member_1)
- inst[1,1] is a sub-struct (zot)
- inst[1,2] is a sub-struct (pook)

- inst[1,3] is a field (bar_member_2)

The array in inst[1,1] has 1 element: 1,1,0; zot_member_1

The array in inst[1,2] has 2 elements: 1,2,0; 1,2,1; pook_member_1 and pook_member_2

The array in inst[2] has 1 element: 2,0; baz_member_1

The trick here is for codegen to emit pseudo machine instructions to build this data structure at run-time. Codegen must locate the entry in Program.struct_headers corresponding to the struct *`foo`*.

The print statement must access:

```
inst[1,1,0]
```

`struct_list` and *`struct_map`*. *`struct_list`* is a vector of type *`Struct`* and *`struct_map`* is a hash table which acts as kind of symbol table for struct declarations. The key to *`struct_map`* is the name of the struct and the payload is the index of this struct stored in *`struct_list`*.

So there is a bit of re-architecting Struct Map and the implementation of struct instantiation:

The struct map has a head "pointer" which will point at the top level struct (*`struct_id = 0`*). And each child has a "member" in the parent which points to it. These "pointers" aren't really pointers but simply indices. Implementing all of this with actual references will be borrowing / ownership hell and I'd rather stay in purgatory.

With the exception of how to get to the top level struct, there is no difference between the top level struct and the sub-structs. Each struct that contains a sub-struct has a member which is the id of the sub-struct (that is, the index into the structs table).

Adding a struct to the map:

- Create the struct
- Create a member
- Push the struct onto the structs vector storing its ID in the member
- Push the member onto the parent's members vector

Assuming the parser triggers this in a strictly linear sequence, store the id of each struct in a vector as it's created. The "parent" will always be the id at the top of this stack.

Structs and I/O: Reading and Writing XML

Currently, when a struct is printed or written to a file CPL will format the output as a simple comma separated list. As suggested previously, once CPL supports named multi-level structs it should be relatively easy to serialize them to XML and load them from XML (I suppose I'd have to introduce a minimal SAX processor).

Optional Semi-Colons

I tried to make the semi-colon optional for the last statement in a block but that crashed and burned and I gave up. It seems a little trivial now so I probably won't return to this.

Arrays of Structs

Let's pretend the following syntax was available (it isn't now):

```
foo = [];                                                    // creates an
empty array
foo[0] = new my_struct;            // instantiate my_struct
   :
   : statements populateing foo[0]
   :
Foo[1] = new my_struct;
   :
   : statements populating foor[1]
etc.
```

The idea is to instantiate a struct in an element of an array (pursuant to the the discussion below on reading and writing structs). How would a CPL program access a member of one of these entries. How about this…

```
foo[0]:mem1 = 100;
foo[0]:mem1 += 1;
foo[0]:mem1 = foo[1]:mem1[2];
foo[0]:mem1 = bar:mem2;

foreach f foo{
   f:mem1 = 100;
   f:mem2 += 5;
}
```

This doesn't feel exactly right but if the parser could figure out that the *lvalue* in this case is both an indexed variable and a struct member reference, then it might work. The syntax inside the foreach loop might work easily enough.

Struct Members Containing Arrays as Rvalues

Consider the following:

```
struct my_struct{
   mem1;
   }

entry fun{
```

```
    foo = new my_struct;
    foo:mem1 = [1,2,3];
    i=0;
    while i < length(foo:mem1){
        print foo:mem1[i];
        i+=1;
    }
}
```

I already have implemented multi-dimensional arrays (e.g. `foo[i,j]`), so `foo:mem1[i]` shouldn't be that different since the underlying data structure is an array. In other words, `foo:mem[i]` should be equivalent to `foo[0,i]`. I just haven't done the code yet. Unclear yet what changes would be necessary to do this.

However, first, I think I would like to generalize the two functions in `cplvars`: `fetch_array_indexed_from_operand_stack` and `fetch_dict_indexed_from_operand_stack` as they are almost identical except for the underlying data types: numeric indices compared to keys and `CplArray` compared to `CplDict` respectively. This might be my first genuine use case for generic functions or another use for macros. Watch this space.

Struct Members and Array Elements Containing Arrays as Lvalues

TBD

Generalizing Indexed Access

Currently, multi-dimensional *arrays* are supported, but multi-dimensional mixed typed collections are not. So you can get to an element of a multi-dimensional array using: `foo[i,j]` where both i and j are numbers. And, if `bar` is a dictionary whose elements contain dictionaries, you can do `bar["key1","key2"]`. But if `zot` is a dictionary whose element contains an array, you can't do `zot["key1",2]` to get at the 2nd element of the array contained with the element of zot whose key is "key1". And, if `pook` is an array whose 2nd element contained a dictionary with an element whose key is "key1", you can't do `pook[2,"key1"]`.

I'm unsure of the use cases for this and there is the problem of foreknowledge. That is, in order for any of this to work, assuming it was actually implemented, the CPL programmer would have to know ahead of time that there was, for example, a dictionary contained in the 2nd element of an array.

In other words in a mixed-collection configuration, I'm wondering if it were the best practice to use the `Type()` built-in function to figure out what to with an element of a dictionary or an array if that element contained a collection. This would negate the foreknowledge problem and would make the CPL programs encountering data structures like this to be more general.

I/O Enhancements

Reading and Writing Dictionaries

I'm not sure yet what the requirements are for this. I suspect the result of reading a dictionary file would be a single dictionary with many entries and writing a dictionary would produce the same kind of file. The specific structure of this file is TBD.

Reading and Writing Structs

I discussed this above: serializing and deserializing structs.

TCP/IP

If the Fopen built-in recognized the file name as a TCP/IP address then may it could support and open mode of "<>" which would mean: I am a TCP/IP client talking to a TCP/IP server. If the name were an octet or V6 address then it would be relatively easy to translate all of the I/O into sockets. However, it would be difficult to tell the difference between foo.org, an address and foo.org, a file name. So, I think it's up to the open mode to provide the hint. I've already implemented "<," to indicate reading from a file containing a list of comma separated records. Perhaps just using "<>" as the open mode is sufficient. However, I think we want to also use this for inter-process communication (i.e. prompt-response). So perhaps "<>|" for IPC and "<>/" for TCP/IP ("|" and "/" are, more or less arbitrary).

Instantiated Structs as Elements of Arrays and Dictionaries

It occurred to me that if we want to support reading (and writing) structs, we'll need a way to store instantiated structs in a collection. Let's say a file had the following in it:

```
100,"blat",200;
200,"zot",300;
300,"poop",10000;
```

There are 3 records in this file. And, suppose the PCL program declared a struct:

```
Struct zort{
   Size;
   Id;
   Distance;
}
```

Where a typical size might be 100, the id is just some string, and a typical distance might be 400. The struct *zort* is the structure of a record in the file.

The read loop that will build an array of zort records might look something like this:

```
file = fopen("blat","<");
array = FreadStruct(file, zort);
```

Evidently, *FreadStruct* reads records from a file structured like this and adds them to an array whose address is returned to the calling program at EOF and assigned to the variable *array*. The question here is can the built-in function allocate space in the operand stack? A CPL program that needs to use each record in the file as a driver might look like this;

```
file = fopen("blat","<");
while !feof(f){
   foo = FreadStructNext(file, zort);
   process_a_foo_record(foo);
}
```

Here the function *FreadStructNext* returns the next *zort* record from the file.

I think writing an array of structs looks like this:

```
file = fopen("blat",">");
foreach z zort_array{
   Fwrite(file, z);
}
```

Or the entire array as:

```
file = fopen("blat",">");
Fwrite(file, zort_array);
```

In either case, *Fwrite* analyzes the specific content of what it has to write and, if it is an instantiated struct, will write the data in the format described above.

Accessing SQL

I'm guessing this will be a set of functions that allow the CPL program to build SQL statements and access the underlying SQL server (e.g. Sqlite or MySql). Probably an ORM is overkill.

Hacking Binary Files

I'm not entirely sure what the requirements for this are or how to implement it. But it may be worth determining the various use cases for dealing with binary files. For example, MP3 files.

Overloading Operators

The "+=" operator is a candidate for overloading. I've implemented it within some additional contexts (see the Operators section in this document)

There are several other operators that could be interpreted to have meaning in cases where the operands aren't scalars:

- "-=" could mean the same thing as Delete(). This is tricky because "delete" or "pop" wants to return what was deleted or popped and the the overloaded operator wouldn't know how to do that.
- The supported binary operators could be applied to arrays. For example, *array+10*, would add 10 to every element in the array and *array1*array2* could builds a cross product.

Stand-Alone Compiled CPL Programs

This is probably something not possible given the many inter-module dependencies arising from the, albeit amateurish, design. But I periodically wonder if it were possible to generate a "compiled" version of a CPL program and use a stand-alone interpreter to run it.

Libraries and Code Reuse

At the moment CPL only supports single code files which may source additional code using the INCLUDE keyword. More complex mechanisms, such as compiled code libraries, are probably a bridge to far. Evidently, this would involve a new keyword, say USE, which would copy the instructions generated by compiling another bit of CPL code into the code frame of the current program. Or perhaps it could only work at the function level, i.e. the USE keyword would copy the instructions into a new frame that could be called via the normal function call mechanism. I think I'll let this one go for the time being unless it becomes obvious that there are use cases where it would be vital to have such a mechanism.

Symbolic Literals (i.e. Constants)

When indexing into an array where the indices are "well-known" (relative to the context) it would be helpful (for readability) to have those indices named. So, for example:

```
literal x = 10;
:
:
A = foo[x];
```

codegen would generate push(lit) of 10 for index expression. Unclear yet how comprehensive this needs to be. If the syntax for this is:

```
<symbolic literal> ::= literal <id> = <expression>
```

what would be legal terms of <expression>? I think <expression> would have to evaluate to a primitive (i.e. number, string or Boolean) because otherwise the value wouldn't be known until run-time and, in that case, the symbolic literal would be simply another variable on the operand stack. So the syntax must be:

```
<symbolic literal> ::= literal <id> = <lit_value>
<lit_value> ::= <number> | <quoted string> | boolan
```

I'd have to make modifications to the symbol table to support items that have values instead of addresses.

Process Management: Process Creation and IPC

I was thinking about Perl and how it can be used instead of sh or awk. I don't think CPL will ever get to that completely but one thing might useful: process launching, control and communication. I think the way to do this is to extend the *Fopen*, *Fwrite* and *Fread* functions to support processes (pipes?). For example (using the open mode discussed previously):

```
process = fopen("/me/pgm","<>|");
fwrite(process, prompt);
rsp = freadln(process);
```

I made this up, so it's kind of a first approximation. The idea is that the open mode, "<>|", indicates that the file is an IPC "channel" and that the file name is the executable file to use to create the process in the first place. *Fopen* creates a process and assigns a *CplProcess* data type to an open file slot. *Fwrite* and *Freadln* will use the information in the *CplProcess*

struct to figure out how to talk to the process. A `Fwrite` consisting of a zero length string will cause the `CplProcess` data type to kill the process.

Protocols and Roles

This approach implies a protocol between processes that have specific roles to play: one process acts in the role of "server" and the other in the role of "client". The protocol is "prompt-response". The protocol works like this:

- When the server starts up it enters a wait loop
- When the client starts up it attempts to send a "prompt" to the server
- The server receives the prompt, does a computation, returns a response and then returns to its wait loop
- The client receives the the response, does something with it and then sends another prompt

The content and format of the prompt and response is completely defined by the specific application implementing the protocol. Within the context of CPL, I would think that these definitions would take the form of structs and that the data sent and received would be serializations of these structs. The structs would be defined in "include" files and the "wire format" (i.e. what the serialization would look like) is more than likely comma separated lists that CPL knows how to map onto the structs. If, on the other hand, I enhance CPL to support XML, this format might be useful for serialization as well. I haven't mentioned it, but clearly JSON is another protocol that might come in handy (though I would think that once CPL could deal with XML it wouldn't take much to implement support for JSON).

Motivation: Use Cases

Why would anybody want to do this using CPL? I think there are two use cases:

1) Server side computation
2) Client side applications that don't run in a browser

Server Side

Currently, languages like PHP, Python and Perl are used by servers to implement complex computations in a backend. CPL could help here as well. Since my goal in developing CPL was to create a very simple language for text hacking, CPL programs could take the place of code written in these other languages in applications whose requirements are simple.

Client Side

Here I envision a requirement for a command line program (or one that runs in the background using a schedular) to query web sites for information. This application would be in two pieces: a TCP/IP client that knew how to format HTML, send HTML requests and receive HTML responses; and an application process that didn't care about TCP/IP or HTML but would send prompts to the TCP/IP client and receive whatever that client got back from the world.

The TCP/IP client could be written in CPL (after CPL has been enhanced to know how to do HTML over TCP/IP) but more likely be written in Rust or C or something that already knows how to do HTML over TCP/IP. The application client would, of course, be written in CPL.

Why Reinvent the Wheel

What I've been describing has is a wheel that has been evolving for the past 50 years and probably needs no additional enhancement. Why bother?

The simple answer is this: why not?

The whole point of my work developing CPL has been to learn Rust and a little bit about compilers and programming languages. It is a "student" exercise. If I had expertise in carpentry or HVAC or electrical work I would build houses for Habitat for Humanity. If I were a doctor I'd work at Doctors Without Borders/Médecins Sans Frontières (MSF). Whether or not I can learn enough Rust to be useful to the world remains to be seen but I see this effort as a way to maybe help when knowledge of Rust would be helpful. In order to implement this and other future enhancements to CPL must, as a matter of necessity, involve learning more about Rust.

So that's it. Basically anything I do to enhance CPL is another way to learn Rust and learning Rust could become helpful.

Whether or not anybody ever uses CPL itself is almost beside the point.

Async Programming

Rust supports Async programming: https://rust-lang.github.io/async-book/intro.html and so it might be helpful to expose these capabilities to CPL programmers or at least simplified abstractions.

I think this would involve extending the language as opposed to providing yet more built-in functions (though there might be a need for that as well).

In my mind, I would add a new prefix to the function signature:

```
async foo(<parameters>){...}
```

and the following verbs:

```
start <function name>
stop <function name>
wait <funtion name list>
```

So the first approximation might be:

```
async a_foo1(p1){
   do something with p1
   return something1
}

async a_foo2(p1){
   do something with p1
   return something
```

```
{

Entry main{
   x = start a_foo1(something);
   y = start a_foo2(something);

   loop{
   rsp = wait a_foo1, a_foo2;
   if rsp:complete{
       exit "done with program"
   }
   eval rsp:val {
       when 0 {do something with x;}
       when 1 {do something with y;}
   }
   }
}
```

Or something like that. *rsp* is an internally defined struct that has two fields: *complete* is a Boolean that indicates that whether or not a response has been received from every async function listed in the wait and *val* contains some kind of indicator as to which response was received.

Essentially, every async function is assigned a handle, these handles are listed in the wait list and the handle is returned as the value of the *rsp:val* field.

The first question that must be asked is, can the operand stack support concurrency? That is, can two or more async functions operate on the operand stack at the same time without totally destroying it?

If each async function creates its own set of blocks (i.e. the block for the function and any blocks created for code blocks) can one function find its variables when the other function is popping its blocks. The immediate problem is variables at blocks further down the stack will change their addresses as blocks above them are popped.

At the moment, the operand stack would explode if this were to occur.

The changes required are TBD.