"The idea that you can do a programming language in 100 000 lines of code or you might be able to do it in a page and there are advantages to the page for sure" (Kay 2009)

1 Definitional Architecture Metaprogramming

Definitional-Interpreters (DI)– convey the ideas and semantics of a language with brevity and clarity; the advantage with defining a language with this method over a specification document is that the result of the **DI** is executable, so there is no question of the meaning of an expression in a context.

Architecture– Different cpus need different executables if they use different instruction-sets, and even if they do use the same **ISA**(instruction-set-architecture), they may have new Extensions that may extend an **ISA**, to let older programs run on newer cpus.

Metaprogramming– technique of programs manipulating other programs as data.

Architecture Data-Structure "`C`" – For compiling/interpreting any language; `C` is an excellent target architecture to use as a stepping stone to actual compiled assembly code. "`damcc`" is not just a **DI** for `C` it is also to serve as documentation for the different **ISA**s. Users should be able to understand an architecture quickly and easily by reading the code for the supported **ISA**s.(as well as elf files and static/dynamic linking) *"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures."* –Alan Perlis

1.1 Metaprogramming with OMeta

See (Warth 2009) for what OMeta Is, this section is about how it works.

(Piumarta 2007) – *"The system that tries to find coverings of those trees pieces of structure within the trees, is itself simple and you can write 2 functions that do it.*

```
reduce(tree, startSymbol) =
    foreach rule in startSymbol.startSets
        if match(tree, rule.pattern)
            rule.action()
            return startSymbol
    return false
match(tree, pattern) =
    if (pattern.isSymbol()) return reduce(tree, pattern)
    if (tree.first != pattern.first) return false
    foreach treeElement, patternElement in tree.tail, pattern.tail
        unless match(treeElement, patternElement)
            return false
    return true
```

***Reduce** a particular piece of tree to a quantity: a register in the machine, a memory location, a literal in the machine, or a `void` just a statement that returns no value.*

***Match** a particular piece of tree against the pattern in the list of rules. These things are recursive so that structuring the trees can have sub structure etc. and what falls out at the end is a yes or no answer.*

It's the inverse of traditional parsing if you like where a normal parser takes unstructured free text and creates a structured representation. What this bottom-up system does is it takes a structured representation and tries to find "an optimal unstructured equivalent" which in our case are machine instructions, for the for the target machine.

One of the interesting things that we've now done is we've described code generation as a tree rewriting process. . . . Various phases of the implementation of a programming system can be described in terms of tree like structures and rewriting-rules applied to them. We can hope in the end to create a single small easily understandable tree rewriting system in which the whole of the language implementation problem is described. . . . Then everything else the the transformation rules for taking structures applying all the functional rules to them to create executable things within fed in and from that point on the whole system takes off exponentially because of the simplicity the expressiveness that is built into those 2 mutually supportive models within it."

1.2 (ELF Format) Relocatable Object "`.o`" Files –

Relocatable object files are files that are in the "elf" executable and linkable format. To Dynamically link to each foreign function call requires parsing header files to see which header provides each function that is used in the current file, to properly call the function and throw an error if a function isn't available. "`.o`" files are binary data assembly with symbols packed in, based on a specification. This is the format for linux executables and libraries, These files are the raw inputs to "linkers". Translating from "`C`" to "`.o`" linkable binary files is the goal here and the definition of compiling.

Stage 1 "explicitly enumerate the nuances of `C`" by "removing them" Define what the `C` language is using "Ometa" to convert from preprocessed `C` to `C` or our small subset of it as an Intermediate Representation(IR) The goal of using this subset of `C` is that it is more easily translatable to machine code.

- files referenced by `#include` will have their functions scanned and added to a table of functions listing the file and the functions and types so that when a external function is called the appropriate "`symbol()`" call of the elf file can be made.
- even though we can replace function calls with gotos, they need to be left in to represent where we will leave symbol references for the "`.o`" files even executables are formatted this way the `main()` function is called.
- all paramaters to functions that are not passed by reference are to be converted to paramaters that are passed by reference, and that variable is never to be referenced,

except when it is copied to another location at the begining of the function and that copy is to be used in its place in all of it's occurences.

- The meaning of all loops translated into gotos
- nested if statements translated into guard clause non-nested if statements
- arithmetic-assignment-operators (`+=`, `-=`, `*=`, `/=`, `%=`) need to be converted to a combination of the arithmetic operation of the correct type and the assignment.
- Each arithmetic-operator-function (`+`, `-`, `*`, `/`, `%`), and each unary-operator-function (`++`, `--`), and each bitwise-operator-function (`&`, `|`, `<<`, `>>`, `~`, `^`) and each boolean-logical-operator (`&&`, `||`, `!`) as well as relational operators (`<`, `<=`, `>`, `>=`, `==`, `!=`) must be converted to a function call named after the type of the inputs.
- `sizeof()` tells the number of bytes that type or array occupies, and `sizeof()` is known at compile time, and most of the time this number is fed into arithmetic functions that can also be calculated at compile time; which can allow optimizations here.
- no more Arrays, Types, Struct, Union, Enum, or any other data-types not manipulatable with assembly; just pointer arithmetic.
- **Memory–** all of variables **AND** pointers Each function or block `{}` is to have all memory explicitly allocated/deallocated with appropriate calls to `malloc()` `realloc()` `calloc()` `aligned_alloc()` `free()`

There is no limit to the depth of the pointer to pointer to pointer to pointer type.

deallocation also happens whenever a goto jumps out of a block, including nested blocks

There may be more caveats I am missing this is a draft document.

`C99` doesn't support function overloading . . . but we could perhaps add that as a command arg to stage1 by renaming overloaded functions to have the names of parameter types in part of the function name.

But wait there's more to go over on memory! Storage and Types

storage classes - auto, static, extern, and register only one storage class can be specified comes first in declaration

```
* storage duration - static or automatic
* scope - block or file
* linkage - external, internal, or no linkage
```

auto(by default), static(), extern(says that this variable will be defined somewhere else, likely another file)

type qualifier - const(val can't be changed abort with error if this is attempted) and volatile(value may change outside our control)

type specifier - struct, enum, union need to be converted to regular types(`uintN_t` `intN_t` `floatN_t`) or their pointers with typedef

"Start at the variable name (or innermost construct if no identifier is present. Look right ***without jumping over a right parenthesis****; say what you see. Look left again* ***without jumping over a parenthesis****; say what you see. Jump out a level of parentheses if any. Look right; say what you see. Look left; say what you see. Continue in this manner until you say the variable type or return type."* –Terrence Parr

Force `anytype_t` to be `sometype_t` with: `*(sometype_t*)&(anytype_t code)` Every operation or function to be made strongly typed with the above, should enlighten users why the following 2 prints aren't equal.

```
float64_t rydb = 123456789543211;// hex code for rydb with typedef
printf("hex value of \"rydb\" is %"PRIX64"\n", rydb);
printf("hex value of \"rydb\" is %"PRIX64"\n", *(uint64_t*)&rydb);
```

If there isn't a function available that accepts the given type abort with error here. **Statically typed** means that types are checked at compile time, **Dynamically typed** means that types are attached to values at run time, `C` is softly typed . .

Metaprogramming Example translator Control flow operators? maybe all we need is "`goto`" read this code:

```
while (condition) statement
//the above while-loop is equivalent to:
begin:
if (condition) {
    statement;
    goto begin;
}

do [loop body statement] while ([condition]);
//the above do-while-loop is equivalent to:
begin:
loop body statement;
if (condition)
    goto begin;

for (variable-declaration;
     condition;
     variable-update)
     statement
//the above for-loop is equivalent to:
variable-declaration;
```

```
begin:
    if (!condition)
        goto end;
    statement;
    variable-update;
    goto begin;
end:
    // end loop
#                          Switch                                 #
switch (expression) {
    case const-expr1: statement1
    case const-expr2: statement2
    case const-expr3: statement3
    default: statements
}
//the above switch is equivalent to:
type value = expression
if (value == const-expr1)
    statement1
else if (value == const-expr2)
    statement2
else if (value == const-expr3)
    statement3
else
    statement0

//Which is also equivalent to:
type value = expression
if !(value == const-expr1) goto alternate2;
    statement1; goto end;
alternate2:
if !(value == const-expr2) goto alternate3;
    statement2; goto end;
alternate3:
if !(value == const-expr3) goto alternate0;
    statement3; goto end;
alternate0:
    statement0
end:
#                                                                #
#                                                                #
#                                                                #
#                                                                    #
#                                                                    #
#                                                                    #
if (n > 0) { //nested if
    if (a > b)
        z = a;
}
else
    z = b;

//guard clause version of nested if
if !(n > 0) goto alternate;
if !(a > b) goto alternate;
    z = a;
    goto end;
alternate:
    z = b;
end:
```

***"Clean"* and convoluted–** after Stage1 is complete there will be no need to:

- parse the `sizeof()` command
- parse types of variables
- parse creation or destruction of any variables – the `C` code will be requesting memory for it's own variables and pointers the appropriate function calls (`malloc()` etc.).
- create or manage Arrays, Types, Struct, Union, Enum
- parse control flow statements switches loops etc.
- destroy any variables – the c code should be explicitly calling `free()` on any variables that are to cease existance after they fall out of scope including global variables at the end of the `main()` scope.

"I came to realize that if I'm not writing a program, I shouldn't use a programming language. People confuse programming with coding, coding is to programming what typing is to writing. It's something that involves mental effort, what you're thinking about, what you're going to say, the words have some importance; but in some sense that even they are secondary to the ideas. In the same way programs are built on ideas, they have to do something and what they're supposed to do, is like what writing is supposed to convey. If people are trying to learn programming by being taught to code, well they're being taught writing by being taught how to type, and that doesn't make much sense."
–Leslie Lamport https://youtu.be/rkZzg7Vowao

"C" Compiler (`damcc`) as the DI of the "C" language – Rarely is a language as well put together as `C`, Mixing `C` and assembly can often be as simple as not translating the assembly parts of the `C` code. A lot of compromises were made to do this; `C` is not so much a language so much as it is basically assembly and just as dangerous. `C` is a notoriously "Dumb" language, in that it will do exactly what **you** tell it to do; Including the set of all known computer *bugs*: memory leaks, stack overflows, disk destruction, or worse.

Stage 2 "Register allocation" – determining which values should be placed into which memory hierarchy and at what time during the execution of the program.

The different storage devices on modern computers: hard disk, ram, cpu cache, & cpu registers all have different latencies, and `C` programs are written as if there are only 2 kinds of memory: main memory and disk. The Compiler is responsible for adding logic into the program to move data between the different memory hierarchies: memory, cpu-cache, and cpu-registers. The compiler is not responsible for the hard disk since it is accessed by function calls to the system or kernel.

"Register allocation phase of the compiler stands between the optimization phase and the final code assembly and emission phase. When the intermediate or internal language (IL) enters register allocation, it is written assuming a hypothetical target machine having an unlimited number of high-speed general-purpose CPU registers. It is the responsibility of the optimization phase to eliminate references to storage by keeping data in these registers, as much as possible." –(Chaitin 1982)

58:57 - 60:54 into (Click 2020) video overview of how C2 works described in (Paleczny, Vick, and Click 2001)

*"**Outline how The architecture description files, in how in general made the compiler suitable for many platforms all at once:** . . . There is an architecture file that is used to describe a CPU architecture including the set of the kind of instructions and a mapping from the sea to ideal nodes to machine equivalents and the registers that are allowed and the encoding for the machine code encodings for them is sort of the common set of things there's a bunch of the things that are in there that are all these fine-grained details but the big pieces are:*

Here's a mapping from idealized nodes which is what the eye are mostly runs on to get under code gen to hardware instructions and the registers that are allowed both on each individual hardware instruction input and overall what are those set of arc registers are allowed in the system and there is a tool which reads this file which is enough funny in its own little format and spits out C code that is in compiled in with the rest of the system.

Builds what's called a burrs pattern matching which does a optimal for some definition of optimal mapping of Hardware instructions to ideal nodes by doing like an overlapping tree like like usually a hardware node covers two to five ideal nodes and so there's some tiling operation that covers them all including if it eight this region then *those edges in between don't have to register allocated but all the edges on the outside do and then what all the registers that are involved and the registers turn into basically a bit set for every input and output of allowed registers and the register allocator Grox that bit set and uses it to pick the right registers so that gets into registrar in pretty heavily right away because that's how you describe inputs to the register allocator"*

there are similar files in LLVM that describe CPU architectures.

Intermediate Representation (IR)

LLVM target descriptions are typically written in LLVM's own assembly-like language, also known as the LLVM **IR**. These files, called LLVM target descriptions, specify the **ISA** of the target machine, register sets, code generation options, and constraints on the code generation. The LLVM target descriptions are used by the LLVM code generator to produce machine code for a specific target architecture.

In the LLVM source code, the target descriptions are typically located in the "lib/Target" directory. The ".td" extension target description files are usually found in subdirectories, for example "lib/Target/X86/X86.td".

At each stage of compilation/translation `damcc` data will remain in C until there is nothing left but assembly.

Inline functions – The instructions of a function have there own place in memory and a jump to that section is performed followed by a jump back to wherever the function was called from. Short function calls that can be "inlined" can take less cpu clocks to run than jumping to some other section of memory. **Inline functions** mean the whole code of the function is written verbatim whereever it is called without any jump instructions. In certain situations some compilers and may decide not to honor the "`inline`" primitive.

```
// Static Inline function for each possible compare or jump
// for the current architecture in assembly
static inline int foo(){
    return 2;
}
```

pointers – "address of" operator: "`&`" - returns address in memory of a variable "dereference" operator: "`*`" - Grab data of type of the argument at the given memory location "ffi"-foreign function interface "efi"-external function interface

advanced Inline operator functions – the ternary operator (`?:`) are to be converted to if statements, and function calls to the appropriate boolean function. assignment operator =

goto – If an active block is exited using a goto statement, any local variables are destroyed when control is transferred from that block. You cannot use a goto statement to jump over initializations. –ibm goto

---

Executable Linkers are basically just home theater setups- Executable Linkers are basically just home theater setups CS361 Chris Kanich https://youtu.be/eQ0KOT_J8Sk?list=PLhy

in certain situations with low memory budgets other techniques might be used – https://github.com/avrdudes/avr-libc

external function interface To allow assembly functions in our compiled code to to be called from another program, we have tor create an external function interface.

https://stackoverflow.com/questions/13901261/calling-assembly-function-from-c

---

```
e1 e2   sequencing
e1 | e2 prioritized choice
e*      zero or more repetitions
e+      one or more repetitions (not essential)
 e     negation
<p>     production application
'x'     matches the character x
```

```
meta Calc (vars) {
  __init__ ::= <empty> => [self vars: [IdentityDictionary new]];
  space    ::= ' ';
  var      ::= <letter>:x <space>* => x;
  num      ::= <num>:n <digit>:d => [[n * '10] + [d - '$0]]
             | <digit>:d <space>* => [d - '$0];
  priExpr  ::= <var>:x => [[self vars] at: x]
             | <num>:n => n
             | '(' <space>* <expr>:r ')' <space>* => r;
  mulExpr  ::= <mulExpr>:x '*' <space>* <priExpr>:y => [x * y]
             | <mulExpr>:x '/' <space>* <priExpr>:y => [x / y]
             | <priExpr>;
  addExpr  ::= <addExpr>:x '+' <space>* <mulExpr>:y => [x + y]
             | <addExpr>:x '-' <space>* <mulExpr>:y => [x - y]
             | <mulExpr>;
  expr     ::= <var>:x '=' <space>* <expr>:r => [[self vars] at: x put: r
             | <addExpr>;
  rep      ::= <space>* <expr>:r '\n' => (println r);
}
```

1.3 Appendix

Malloc() basically wraps both brk and mmap2 syscalls, but it also keeps internal information about the memory that has been allocated for use in later malloc calls or free calls. how this is implemented falls outside the scope of a compiler: Having to move around countless copies of the malloc() function in each and every executable in the operating system is going to be slower than Dynamically linking to malloc(). Executable .o files aren't just a bunch of assembly code, there needs to be dynamically linkable functions at execution time even if we do our own linking.

1.3.1 Events

Probably should write a library for events, hopefully this can be done in a library.

The sublist method of event management, maintains a list of subscribers for each possible event, and when that event is published it is sent out to each of them. Currently this section is in java, but I should translate it into C soon implemented as a C library as well, so we're actually going to describe the operations in java, to avoid any confusion added from also having to juggle pointers and address in addition to the events.

```
import java.util.*;//import of java.util.event

interface ThrowListener { public void Catch(); } //Declaration event inte
// OR import of the interface, OR declared somewhere else in the package

/*____________________________________________________________*/class Th
//list of catchers & corresponding function to add/remove them in the lis
    List<ThrowListener> listeners = new ArrayList<ThrowListener>();
    public void addThrowListener(ThrowListener toAdd){ listeners.add(toAd
    //Set of functions that Throw Events.
        public void Throw(){ for (ThrowListener hl : listeners) hl.Catch(
            System.out.println("Something thrown");
        }
////Optional: 2 things to send events to a class that is a member of the
. . . go to Thrower.java file see this code . . .
}

/*_____________________________________________________________*/class
implements ThrowListener {//implement added to class
//Set of @Override functions that Catch Events
    @Override public void Catch() {
        System.out.println("I caught something!!");
    }
////Optional: 2 things to receive events from a class that is a member of
. . . go to Catcher.java file see this code . . .
}
```

```
#                                                                    #
#                                                                    #
#                                                                    #
```

---

as long as a pointer is not using a pointer to a pointer. `void func(int* p, int** pr){p++; (*pr)++;}`

---

overview of how C2 works(Paleczny, Vick, and Click 2001)
Open, extensible composition models(Piumarta 2011b)
Association-based model of dynamic behaviour(Piumarta 2011a)
PEG-based transformer provides stages in a compiler(Piumarta 2010)
Open, Extensible Object Models(Piumarta and Warth 2008)
OMeta(Warth and Piumarta 2007)
Virtual processor: dynamic code generation(Piumarta 2004)
Sebastian Falbesoner 2014 Implementing a Global Register Allocator for TCC

Imperative & Declarative Programming (Programming as Planning, Executable Specifications)(Samimi 2009)

i'm sure you have seen an imperative as well as declarative methodologies to do programming. imperative is how implementations are done and declarative is we've seen prolog language you know it's meanings they're compact um they're very understandable but we need search in order to to execute them so that they're often not practical.

part of what you're interested here to save codes as well as to add understandability to problems is; is there a way to kind of combine both at the same time? Which is not very common today. This is a famous quote that i made recently that the natural combination of declaritive and imperative is missing. A couple of methodologies that we kind of studied in this direction:

basically if you add a a a layer of logic on top of your your your languages to support search basically your heuristic search you can do imperative programming with a little bit of overhead if you always willing to add optimizations so in the in the presence of multiple methods if there's a if there's an optimization that always tells you based on the state which method is the is a good method to take so you always do this check to to find out the different alternatives then you have this little bit overhead of checking to do your normal imperative but when you do want when you do have the you know um leverage to do kind of search you have the option to say well explore possibilities and find the best scenario to to do so this is what this this project is all about you know allows you to kind of separate the optimizations and heuristics from from the actual implementations the actions that you see here to satisfy goals.

The need for avoiding complexity is more important in the context of **DI** than the need for optimization.

References

Chaitin, G. J. 1982. "Register Allocation &Amp; Spilling via Graph Coloring." In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, 98–105. SIGPLAN '82. New York, NY, USA: Association for Computing Machinery. `https://doi.org/10.1145/800230.806984`.

Click, Cliff. 2020. "The Sea of Nodes and the HotSpot JIT." *YouTube*. `https://youtu.be/9epgZ-e6DUU?t=3535`.

Kay, Alan. 2009. "How Complex Is "Personal Computing"?" *YouTube*. `https://youtu.be/HAT4iewOHDs`.

Paleczny, Michael, Christopher Vick, and Cliff Click. 2001. "The Java HotspotTM Server Compiler." In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, 1. JVM'01. USA: USENIX Association.

Piumarta, Ian. 2004. "The Virtual Processor: Fast, Architecture-Neutral Dynamic Code Generation." In *Proceedings of the 3rd Conference on Virtual Machine Research and Technology Symposium - Volume 3*, 8. VM'04. USA: USENIX Association.

———. 2007. "Building Your Own Dynamic Language." *YouTube*. `https://youtu.be/cn7kTPbW6QQ?t=2736`.

———. 2010. "PEG-Based Transformer Provides Front-, Middle-and Back-End Stages in a Simple Compiler." In *Workshop on Self-Sustaining Systems*, 10–20. S3 '10. New York, NY, USA: Association for Computing Machinery. `https://doi.org/10.1145/1942793.1942796`.

———. 2011a. "An Association-Based Model of Dynamic Behaviour." In *Proceedings of the 1st International Workshop on Free Composition*. FREECO '11. New York, NY, USA: Association for Computing Machinery. `https://doi.org/10.1145/2068776.2068779`.

———. 2011b. "Open, Extensible Composition Models." In *Proceedings of the 1st International Workshop on Free Composition*. FREECO '11. New York, NY, USA: Association for Computing Machinery. `https://doi.org/10.1145/2068776.2068778`.

Piumarta, Ian, and Alessandro Warth. 2008. "Open, Extensible Object Models." In *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16,* *2008 Revised Selected Papers*, 1–30. Berlin, Heidelberg: Springer-Verlag. https://doi.org/10.1007/978-3-540-89275-5_1.

Samimi, Hesam. 2009. "Imperative & Declarative Programming (Programming as Planning, Executable Specifications)." *YouTube*. https://youtu.be/HAT4iewOHDs?t=3578.

Warth, Alessandro. 2009. "Domain-Specific Languages (OMeta)." *YouTube*. https://youtu.be/HAT4iewOHDs?t=2060.

Warth, Alessandro, and Ian Piumarta. 2007. "OMeta: An Object-Oriented Language for Pattern Matching." In *Proceedings of the 2007 Symposium on Dynamic Languages*, 11–19. DLS '07. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1297081.1297086.